

Developing RESTful web APIs in Java

Restlet IN ACTION



Jérôme Louvel
Thierry Templier
Thierry Boileau

FOREWORD BY Brian Sletten

MANNING

Restlet in Action

Restlet in Action

DEVELOPING RESTFUL WEB APIs IN JAVA

JÉRÔME LOUVEL
THIERRY TEMPLIER
THIERRY BOILEAU



MANNING
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964
Email: orders@manning.com

©2013 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.



Manning Publications Co.
20 Baldwin Road
PO Box 261
Shelter Island, NY 11964

Development editor: Jeff Bleiel
Copyeditor: Corbin Collins
Proofreaders: Elizabeth Martin, Melody Dolab
Typesetter: Dennis Dalinnik
Cover designer: Marija Tudor

ISBN: 9781935182344

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – MAL – 18 17 16 15 14 13 12

*To my father, Guy Louvel,
for his love of life and people
and for inspiring my passion for computers*

—J.L.

brief contents

PART 1	GETTING STARTED	1
1	■ Introducing the Restlet Framework	3
2	■ Beginning a Restlet application	13
3	■ Deploying a Restlet application	46
PART 2	GETTING READY TO ROLL OUT	79
4	■ Producing and consuming Restlet representations	81
5	■ Securing a Restlet application	121
6	■ Documenting and versioning a Restlet application	151
7	■ Enhancing a Restlet application with recipes and best practices	165
PART 3	FURTHER USE POSSIBILITIES	201
8	■ Using Restlet with cloud platforms	203
9	■ Using Restlet in browsers and mobile devices	242
10	■ Embracing hypermedia and the Semantic Web	274
11	■ The future of Restlet	294

contents

<i>foreword</i>	xvii
<i>preface</i>	xix
<i>acknowledgments</i>	xxi
<i>about this book</i>	xxiii
<i>about the cover illustration</i>	xxvii

PART 1 GETTING STARTED.1

1 *Introducing the Restlet Framework* 3

1.1 “Hello World” with Restlet	5
<i>Coding a ServerResource subclass</i>	5
<i>Running the server</i>	6
<i>Using the ClientResource class</i>	7
1.2 Overview of the Restlet Framework	8
<i>Main benefits of the Restlet API</i>	9
<i>Overall design of the Restlet Framework</i>	10
<i>Available editions and deployment targets</i>	11
1.3 Summary	12

2 *Beginning a Restlet application* 13

2.1 The purpose of Restlet applications	14
2.2 The structure of Restlet applications	15

2.3	Setting up a Restlet application	17
	<i>Creating an Application subclass</i>	17
	<i>Setting application properties</i>	19
	<i>Exploring the application context</i>	20
	<i>Configuring common services</i>	22
2.4	The Restlet routing system	24
	<i>Preprocessing and postprocessing calls with a Filter</i>	24
	<i>Using a router to dispatch calls based on URIs</i>	27
2.5	Using Restlet resources in an application	30
	<i>Resource, the base of all resources</i>	30
	<i>Using ServerResource as target of calls</i>	31
	<i>Using ClientResource as source of calls</i>	35
	<i>Higher-level resources with Java annotations</i>	38
	<i>Updating the example mail application</i>	42
2.6	Summary	45

3

3	Deploying a Restlet application	46
3.1	The purpose of Restlet components	47
3.2	The structure of Restlet components	48
3.3	Standalone deployment with Java SE	50
	<i>Creating a Component subclass</i>	50
	<i>Adding server and client connectors</i>	53
	<i>Setting up virtual hosting</i>	57
	<i>Configuring common services</i>	61
3.4	Declarative configuration in XML	62
	<i>XML configuration with Component</i>	62
	<i>XML configuration with Spring Framework</i>	63
3.5	Deployment in an existing Java EE server	67
	<i>The Servlet extension</i>	67
	<i>Servlet engine as a connector for a Restlet component</i>	68
	<i>Servlet engine as a container of Restlet applications</i>	70
	<i>The Oracle XML DB extension</i>	71
	<i>Restlet Framework as a library inside Servlet applications</i>	72
	<i>Dynamic deployment in OSGi environments</i>	73
3.6	Summary	76

PART 2 GETTING READY TO ROLL OUT.....79

4

4	Producing and consuming Restlet representations	81
4.1	Overview of representations	82
	<i>The Variant and RepresentationInfo base classes</i>	82
	<i>The Representation class and its common subclasses</i>	83

4.2	Producing and consuming XML representations	87
	<i>The org.restlet.ext.xml.XmlRepresentation class</i>	88
	<i>Using the DOM API</i>	89
	<i>Using the SAX API</i>	92
	<i>Evaluating XPath expressions</i>	94
	<i>Handling XML namespaces</i>	95
	<i>Validating against XML schemas</i>	97
	<i>Applying XSLT transformations</i>	99
	<i>Using the JAXB extension</i>	102
	<i>Alternative XML binding extensions</i>	104
4.3	Producing and consuming JSON representations	105
	<i>Using the JSON.org extension</i>	106
	<i>Using the Jackson extension</i>	107
4.4	Applying template representations	108
	<i>Using the FreeMarker extension</i>	109
	<i>Using the Velocity extension</i>	111
4.5	Content negotiation	113
	<i>Introducing HTTP content negotiation</i>	113
	<i>Declaring resource variants</i>	115
	<i>Configuring client preferences</i>	116
	<i>Combining annotated interfaces and the converter service</i>	117
4.6	Summary	119

5

Securing a Restlet application 121

5.1	Ensuring transport confidentiality and integrity	122
	<i>Understanding TLS and SSL</i>	122
	<i>Storing keys and certificates</i>	124
	<i>Generating a self-signed certificate</i>	125
	<i>Generating a certificate request</i>	125
	<i>Importing a trusted certificate</i>	126
	<i>Enabling HTTPS in Restlet</i>	126
	<i>Providing a custom SSL context</i>	128
5.2	Authenticating users	129
	<i>Providing authentication credentials on the client side</i>	130
	<i>The org.restlet.security.Authenticator class</i>	134
	<i>Challenge-based authentication</i>	135
	<i>Verifying user credentials</i>	136
	<i>Certificate-based authentication</i>	139
5.3	Assigning roles to authenticated users	141
	<i>Request principals</i>	141
	<i>The org.restlet.security.Enroler interface</i>	142
	<i>Organizations, users, and groups</i>	142
	<i>The default enroler and verifier</i>	143
5.4	Authorizing user actions	143
	<i>The org.restlet.security.Authorizer class</i>	143
	<i>The role authorizer</i>	144
	<i>The method authorizer</i>	145
	<i>Fine-grained authorization</i>	145
	<i>Using Java security manager</i>	146

5.5	Ensuring end-to-end integrity of data	147
	<i>Ensuring representation integrity</i>	148
	<i>Representation digesting</i>	148
	<i>Digesting without losing content</i>	149
5.6	Summary	150

6 Documenting and versioning a Restlet application 151

6.1	The purpose of documentation and versioning	152
	<i>Use cases</i>	152
	<i>Pitfalls</i>	152
	<i>Recommendations</i>	153
6.2	Introducing WADL	154
6.3	The WadlApplication class	155
6.4	The WadlServerResource class	156
	<i>Overview of properties and methods</i>	156
	<i>Improving description of existing server resources</i>	158
	<i>Describing a single resource</i>	162
6.5	Automatic conversion to HTML	163
6.6	Summary	164

7 Enhancing a Restlet application with recipes and best practices 165

7.1	Handling common web elements	166
	<i>Managing forms</i>	166
	<i>Managing cookies</i>	169
	<i>Serving file directories</i>	172
	<i>Customizing error pages</i>	174
	<i>Handling file uploads</i>	176
7.2	Dealing with Atom and RSS feeds	178
	<i>Exposing web feeds</i>	178
	<i>Consuming web feeds</i>	180
7.3	Redirecting client calls	180
	<i>Manual redirection</i>	181
	<i>The org.restlet.Redirector class</i>	182
7.4	Improving performances	185
	<i>Streaming representations</i>	185
	<i>Compressing representations</i>	186
	<i>Partial representations</i>	187
	<i>Setting cache information</i>	188
	<i>Conditional methods</i>	190
	<i>Removing server-side session state</i>	191
7.5	Modularizing large applications	192
	<i>Server dispatcher</i>	192
	<i>RIAP pseudoprotocol</i>	194
	<i>Private applications</i>	196

7.6	Persisting resources state	197
	<i>The JDBC extension</i>	197
	<i>The Lucene extension</i>	198
	<i>Best design practices</i>	199
7.7	Summary	199

PART 3 FURTHER USE POSSIBILITIES201

8

Using Restlet with cloud platforms 203

8.1	Restlet main benefits in the cloud	204
	<i>Better SaaS portability</i>	204
	<i>Easy client access to services from the cloud</i>	206
8.2	Deployment in Google App Engine	207
	<i>What is GAE?</i>	207
	<i>Deploying Restlet applications in GAE</i>	208
	<i>Using Google Accounts authentication</i>	210
8.3	Deployment in Amazon Elastic Beanstalk	211
	<i>What is Elastic Beanstalk?</i>	211
	<i>Deploying Restlet applications</i>	212
8.4	Deployment in Windows Azure	214
	<i>What is Azure?</i>	214
	<i>Deploying Restlet applications</i>	215
8.5	Accessing web APIs from GAE	221
	<i>GAE restrictions and URL fetch</i>	221
	<i>Using Restlet to access RESTful applications</i>	221
8.6	Accessing OData services	221
	<i>What is OData?</i>	222
	<i>Generating classes for access using Restlet</i>	226
	<i>Calling OData services</i>	228
8.7	Accessing Amazon S3 resources	230
	<i>Configuring a bucket</i>	230
	<i>Accessing a resource with the bucket</i>	230
8.8	Accessing Azure services	233
	<i>Configuring storage accounts</i>	233
	<i>Using table service</i>	233
8.9	Accessing intranet resources with Restlet's SDC extension	235
	<i>Secure Data Connector overview</i>	235
	<i>Installing SDC agent</i>	236
	<i>Using the Restlet SDC connector</i>	238
	<i>Restlet SDC support in GAE edition</i>	240
8.10	Summary	240

9 *Using Restlet in browsers and mobile devices* 242

- 9.1 Understanding GWT 243
 - GWT overview* 243 ■ *Installing and using GWT* 244
 - GWT and REST* 246

- 9.2 The Restlet edition for GWT 247
 - The RequestBuilder class of GWT* 247 ■ *Restlet port to GWT* 248 ■ *Communicating with a REST API* 249
 - Handling cross-domain requests on the client side* 257

- 9.3 Server-side GWT extension 259
 - Working along with GWT-RPC* 260 ■ *Handling cross-domain requests on the server side* 261

- 9.4 Understanding Android 262
 - Android overview* 263 ■ *Installing Android and Eclipse plug-ins* 263

- 9.5 The Restlet edition for Android 267
 - Restlet port to Android* 267 ■ *Client-side support* 268
 - Server-side support* 271

- 9.6 Summary 272

10 *Embracing hypermedia and the Semantic Web* 274

- 10.1 Hypermedia as the engine of RESTful web APIs 275
 - The HATEOAS principle* 275 ■ *What are hypermedia and hypertext?* 276 ■ *Hypertext support in Restlet* 276
 - The new hyperdata trend* 278

- 10.2 The Semantic Web with Linked Data 280
 - REST and the Semantic Web* 280 ■ *Using RDF in representations* 281

- 10.3 Exposing and consuming Linked Data with Restlet 286
 - Exposing RDF resources* 286 ■ *Consuming linked data with Restlet* 291

- 10.4 Summary 293

11 *The future of Restlet* 294

- 11.1 Evolution of HTTP and the rise of SPDY 295
 - HTTP history so far* 295 ■ *Refactoring with HTTP/1.1 bis* 296
 - The rise of alternatives* 296

11.2	The Restlet roadmap	299
	<i>Connectors for SPDY, HTTP and SIP</i>	299
	<i>▪ Enhancements to the Restlet API</i>	299
	<i>▪ Editions for JavaScript and Dart</i>	301
	<i>Restlet Forge</i>	302
	<i>▪ Restlet Platform</i>	304
	<i>▪ APISpark, the online platform for web APIs</i>	306
11.3	Restlet community	307
	<i>Third-party projects</i>	308
	<i>▪ Contributing to Restlet</i>	310
11.4	Summary	310
APPENDIXES		313
<i>appendix A</i>	<i>Overview of the Restlet Framework</i>	315
<i>appendix B</i>	<i>Installing the Restlet Framework</i>	328
<i>appendix C</i>	<i>Introducing the REST architecture style</i>	359
<i>appendix D</i>	<i>Designing a RESTful web API</i>	371
<i>appendix E</i>	<i>Mapping REST, HTTP, and the Restlet API</i>	397
<i>appendix F</i>	<i>Getting additional help</i>	413
	<i>references</i>	415
	<i>index</i>	417

foreword

News flash: the web is kind of a Big Deal.

It is difficult to consider its full impact on technology, society, commerce, education, governance, and entertainment without resorting to grand language that has been stated many times before. It is a Big Deal and we will never be the same because of it.

But, here's the thing. If we sat down and tried to rebuild the web today, knowing what we know from 20 years of experience with it, we would probably fail. The problem is that as software developers, we generally think in terms of software constructs: objects, services, methods, etc. While useful from a solution space perspective, they can induce coupling and coupling does not scale.

The web works because in its design, Sir Tim Berners-Lee and his cohorts embraced the notion of change. The thing we forget is that they were not trying to build the web that we know; they were trying to build a system that worked for a dynamic organization such as CERN. Logically named resources could be requested and manipulated with no regard to how back-end systems worked. New shapes of information could be negotiated over time without disrupting deployed systems. Clients and servers could evolve independently.

The REpresentational State Transfer (REST) architectural style embraced these ideas and began to describe how to build flexible, scalable, change-tolerant systems. The primary shift was away from implementation details and toward the information that flows through the infrastructure. Clients were less cognizant of what to expect and more reactive to what they were told. The focus was on the properties induced by the architectural choices, not the technologies used to implement the solutions.

This is an important point because most REST frameworks built in languages such as Java ignore these distinctions. Their choices reflect a desire to bend REST to a privileged Java world view, not the other way around. What makes REST special and interesting is lost in the process, which is why I think most of them ultimately fail. Jérôme's work on Restlet struck me as having the appropriate perspective: how can we conveniently surface the ideas of REST in a language like Java without debasing the goals of the architectural style.

He tackled the problem from both the client and the server perspectives. He introduced objects that stood for a Resource itself. He turned concepts such as content negotiation and metadata management into registered services, outside of the concerns of these resources. He embraced the idea of logically named protocols to extend the idea of REST into the framework with tools like the Restlet Internal Access Protocol (RIAP) and the Class Loader Access Protocol (CLAP). He understood the value of getting the APIs to a consistent, uniform place before declaring success. The resulting framework is cleaner, more flexible, and more true to the spirit of REST than any other language-based approach I have seen.

While the Restlet documentation has always been adequate, as I introduced the API in my talks and courses, I longed for a solid book on the subject. I even toyed with the idea of writing one. Thankfully, now, as I look at the manuscript in front of me, I do not have to.

Restlet in Action by Jérôme et al. is exactly the broad, deep, and example-driven book I had hoped for. The list of topics they tackle is impressive. The authors provide thorough but not overwhelming coverage of REST itself, security, and performance. Beyond that, they also discuss issues of growing interest such as mobile applications, the cloud, the Semantic Web, Linked Data, and the future of HTTP. The whole endeavor is grounded in practical advice on how to use Restlet toward those ends. These ideas are a Big Deal and this book will help you understand why and how to benefit from them.

I have already preordered *Restlet in Action* for many of my clients and students and anticipate it being a staple in my courses for years to come.

BRIAN SLETTEN
PRESIDENT
BOSATSU CONSULTING, INC.

****preface****

When I had a chance to use the Mosaic web browser way back in 1994, I fell in love with the web at first sight and became interested in HTML and the way the W3C was driving the growth of the web along with the IETF. A year later, I discovered Java by reading Sun's white paper and was convinced that it would lead to a great future. I started using it professionally to write a web load-testing tool using CORBA and an HTTP proxy.

In 2001, while reading *Weaving the Web* by Tim Berners-Lee, I was hooked by his grand vision of a read-write Semantic Web and started to think about the best way to help it come about. In 2004, I built a website in my spare time called Semalink which bridged the classic web of documents with the semantic web of data. As I wanted to stay true to the principles of the web, I read more and more about REST and the core HTTP and URI standards and realized that the Servlet API had too large a gap applying those principles. That's when Restlet emerged as a higher-level Java API derived directly from REST and HTTP. This was very helpful, so I thought about sharing it with others. I believed that it could radically change the way we develop web applications, in the same way that REST was radically changing the way I was thinking about the web.

After announcing the Restlet Framework on December 2005 in an article on TheServerSide website, I hoped that this open source project, the first REST framework for Java, would contribute to the success of REST in the Java world. I wasn't sure how the Java community would welcome it since the industry was strongly behind Java EE (including Servlet, JSP, and Spring MVC) and WS-* (based on the SOAP protocol) technologies.

Feedback was quick and mostly positive, suggesting features and leading me to dedicate more time to this project than I had initially planned. While REST was gaining wider support, from early adopters to the whole industry, Restlet matured by broadening its scope of features and by growing its community. In 2007, my friend Thierry Boileau joined me fulltime as a core developer, and one year later we formally created a company to provide professional services to the Restlet community, ensuring that we could dedicate even more time to Restlet.

Early on, users of the framework asked for better and more complete documentation and it became clear that we needed to make serious efforts on this front. At the end of 2008, we started to look for a publisher who would support our idea for a Restlet book. When Guillaume Laforge, head of Groovy development, told us that Manning was looking for exciting new technology to add to its list, it was clear that a “Restlet in Action” book would be ideal, especially with Manning’s Early Access Program (MEAP) that would give the community access to the electronic version of the draft manuscript, and provide us continuous and valuable feedback during the writing process.

Philippe Mougin, a web services expert, joined us for a few months and contributed important content on REST, including proper URI design and a comparison with the RPC style. Also, Bruno Harbulot, a PhD from the University of Manchester who had been instrumental during the design of the Restlet security API, contributed the initial content for the chapter on security.

Later in 2010, Thierry Templier, a Java EE expert and author of several books published by Manning, started to collaborate with us on the Restlet project and became the third coauthor, contributing two chapters on the cloud, GWT, and Android, as well as content on Spring integration, OSGi deployment, and security. He is now part of our team, focusing on the development of our new APISpark Platform as a Service (PaaS) and on editions of the Restlet Framework for JavaScript and OSGi.

In addition, Tim Peierls, coauthor of *Java Concurrency in Practice* and a key contributor in the Restlet community, provided valuable technical feedback on our draft manuscript. He also contributed an introduction to chapter 1 and worked hard to help us improve the quality of the English in the manuscript.

Finally, after three years of intense effort, *Restlet in Action* is ready for a new life in the world of bookstores and libraries. Speaking for all the coauthors and all the contributors to this book, I hope that you will enjoy reading it and developing RESTful web APIs using the Restlet Framework.

JÉRÔME LOUVEL

acknowledgments

We are sincerely and humbly indebted to Roy T. Fielding for his PhD dissertation that described REST and created a radical shift in the way we envisioned the web and its core standards. Without his work on REST and the HTTP protocol, Restlet wouldn't exist. We would also like to thank all Restlet Framework contributors, including past and present committers, as well as users submitting issues, helping with the documentation, and answering questions of other users.

Thanks to Jim Alateras, John D. Mitchell, and Steve Loughran for their help during the book's inception phase, especially Guillaume Laforge who introduced us to Manning.

Our sincere thanks to the staff at Manning, including Marjan Bace, Michael Stephens, Mary Piergies, Maureen Spencer, Christina Rudloff, Corbin Collins, Elizabeth Martin, Ozren Harlovic, and Melody Dolab. Special thanks to our editor Jeff Bleiel for his patience and support during the book's development phase.

Manning rounded up a great group of reviewers, whom we thank for helping to transform our drafts into the book you are now reading through incremental enhancements and some refactoring ideas. The reviewers include Adam Taft, Aron Roberts, Brian Sletten, Bryan Hunt, Colin Yates, Dave Nicolette, Dave Pawson, Doug Warren, Dustin Jenkins, Fabián Mandelbaum, Gabriel Ciuloaica, Gordon Dickens, James Ferrans, Jeff Thomas, Jeroen Nouws, Jim Alateras, Johannes Kirschnick, John Logsdon, Kristoffer Gronowski, Marcelo Ochoa, Rhett Sutphin, Richard E. Brewster, Stephen Koops, Tal Liron, Vincent Nonnenmacher, and Rob Heittman.

Special thanks to Tim Peierls and Nick Watts, our technical proofreaders, for their careful review, just before the book went into production, of the chapters and the

appendices respectively. Thank you as well to all our MEAP readers who posted feedback messages on Manning’s Author Online forum; we tried to take all of them into account. We also thank Brian Sletten for penning the foreword.

Finally, our special thanks go to Benoit Maujean and Stève Sfartz for their continuous support through the years and to our advisory team members, Didier Girard, Jean-Paul Figer, and Frederic Renouard, for sharing their experience and for helping us at both the technical and business levels.

JÉRÔME LOUVEL

First, I’d like to thank Lance Taturo who assured me that I could indeed write a technical book in English. Thanks also to my entire family and to my closest friends for their support. Finally, I’d like to thank my wife, Sandrine, and my two-year-old daughter, Clara, for their patience during the writing process and the numerous evenings and weekends spent working on the book. I am grateful for your support and your love.

THIERRY TEMPLIER

I would like to thank my wife, Séverine, and our lovely little boy, Maël, for being by my side in life. I also want to thank Jérôme and Thierry for bringing me to this project, and thanks to the Manning team for their confidence, support, and professionalism. Thanks finally to everyone who helps me move forward in life and be the best that I can be.

THIERRY BOILEAU

I am deeply grateful to Jérôme Louvel for bringing me to the Restlet project. I would also like to thank the Restlet community and its active contributors like Tim Peierls, Kristoffer Gronowski, Martin Svensson, Bryan Hunt, Wolfgang Werner, Shaun Elliott, and many others, for their time and dedication. Thanks to Benoît Roblin for reviewing the first chapters. Thanks to Didier Arnachellum for happily playing with the framework. In conclusion, and in French, *un petit clin d’œil à mon frère Fabrice*.

about this book

We wrote this book to help readers efficiently develop RESTful web APIs based on the Restlet Framework. It is also an answer to the open source Restlet community's request for a comprehensive guide to this technology. The book introduces you to the world of REST and HTTP through the use of the Restlet Framework, which directly derives from those standards.

We've tried to stay very practical throughout the book by providing many source code listings and illustrative figures, introducing Restlet concepts and fundamentals along the way. In addition, an example RESTful mail system serves as a conductor all along the book.

Six appendixes provide additional details related to the Restlet technology as well as a generic presentation of the REST architecture style and of the ROA/D design methodology that are both valuable beyond the Restlet Framework.

Audience

Our main audience is Java developers who are interested in the web standards such as HTTP and REST, as well as their usage to expose and consume web APIs. No prior knowledge of Restlet is required.

Readers should ideally be familiar with the Java EE ecosystem including technologies such as Servlet, OSGi, and Spring Framework to make the most of the book, but this isn't a prerequisite to reading the book.

The secondary audience is web API project managers and architects who want to understand how to design and develop a RESTful web API in a controlled manner, using the Restlet Framework or alternative technologies for the implementation.

Roadmap

Part 1 gets you acquainted with the Restlet Framework, quickly looking at the code while introducing important Restlet concepts such as editions, applications, routing, resources, components, and available deployment options.

Chapter 1 gets you started with the Restlet Framework by showing you how to write your first client and server programs. It reviews the main features and benefits of Restlet and gives you an overview of the Restlet Framework, an open source REST framework for Java.

Chapter 2 starts with background information on Restlet applications. It explains how to set up a Restlet application and how the filtering and dispatching of calls works with Restlet's routing system. Finally, it covers using client-side and server-side Restlet resources including both method-overriding and annotation-based resource implementation.

Chapter 3 starts with background information on Restlet components. It explains how to deploy in standalone Java SE virtual machines, configuring virtual hosts and log and status services using either Java-based or declarative XML configuration. It also covers the deployment in Java EE application servers and OSGi environments.

Part 2 gets you to the next level of knowledge with more advanced topics such as security, documentation, and versioning or optimization. Those topics will become essential as you move your Restlet application closer to a deployment in production.

Chapter 4 covers producing and consuming XML and JSON representations, as well as producing HTML using template representations. It explains how HTTP content negotiation is supported by Restlet Framework and describes how to simplify representation handling with the converter service.

Chapter 5 covers how to secure a Restlet application at various levels. It starts with the use of SSL/TLS to secure communication, then explains how to authenticate remote users, assigning roles in order to authorize them to perform actions on the system. Finally, it describes how to ensure end-to-end integrity of the data.

Chapter 6 explains why you need to document and version your web API and describes the main pitfalls and recommendations for doing so. It introduces the Web Application Description Language (WADL) and its support in Restlet Framework.

Chapter 7 covers handling common web artifacts, such as forms and cookies. It explains how servers can redirect clients and how to handle file uploads on the client and server side. It also provides guidance on how to improve performance of Restlet applications and how to split a large Restlet application into several modules.

Part 3 looks at further Restlet usage possibilities, such as with cloud platforms, browsers, and mobile devices. It also explains how to embrace hypermedia and the Semantic Web in your Restlet projects and what the future of Restlet looks like.

Chapter 8 covers how to use Restlet in the cloud. It starts with deploying Restlet applications to cloud platforms such as Google App Engine, AWS Elastic Beanstalk, and Microsoft Azure. It explains how to access RESTful applications from the cloud using OData, AWS S3, and Azure services and it describes how to securely access intranet resources from public cloud platforms with Restlet, thanks to the Secure Data Connector protocol.

Chapter 9 starts with a description of Restlet editions for GWT and Android. Then it explains how to use REST within GWT applications and within Android-based mobile devices with the Restlet Framework.

Chapter 10 explains why hypermedia is important for RESTful web APIs and describes how hypertext and hyperdata are supported in Restlet to drive applications. In addition, it discusses the relationship between REST and the Semantic Web. Finally, it covers how Restlet can expose and consume Linked Data in RDF.

Chapter 11 starts by covering the state of the HTTP protocol and alternatives. It introduces the SPDY protocol and discusses its impact on REST, HTTP, and Restlet. An overview of the Restlet roadmap covers planned enhancements to Restlet API, extensions, and editions; the planned Restlet Studio tool for Eclipse; and the planned online Restlet Cloud service. It concludes the book with the new “API Spark” platform offering Restlet as a service, as well as a review of Restlet community driven projects.

The book also contains six appendixes. The first gives an overview of the Restlet Framework, its Java API and engine, its available extensions for each edition, and its versioning scheme. The second explains how to install the Restlet Framework using the various distributions provided and the major IDEs available. Appendix C introduces the REST architecture style and how it became an alternative to RPC. Appendix D explains how to design a RESTful web API following the iterative and agile ROA/D methodology in order to succeed in your projects. Appendix E provides reference materials, mapping REST and HTTP to the Java-based Restlet API. The last appendix gives some pointers on where to get additional help beyond this book.

Online and printed references mentioned in the book are listed at the end of the book.

Code conventions and downloads

All source code in the book is in a fixed-width font like this, which sets it off from the surrounding text. In many listings, the code is annotated to point out key concepts, and numbered bullets are sometimes used in the text to provide additional information about the code.

The source code for the examples in the book is available for download from the publisher’s website at www.manning.com/RestletInAction. It is also available for download as part of regular Restlet distributions in the `src/org.restlet.example.org.restlet.example.book.restlet` root package. See the respective chapters for details about the dependencies required to compile and run the examples.

All the source code in the examples has been written for and tested with Restlet Framework version 2.1 and should mostly work with version 2.0 as well, except for features added in version 2.1. The source code should continue to work well with future 2.x releases.

Visit www.restlet.org to download the latest Restlet Framework release. See both chapter 1 and appendix B for more details on how to get started. The only requirement is to have a Java Development Kit (JDK) version 1.5 or above installed on your computer.

Author Online

Purchase of *Restlet in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the authors and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/RestletinAction. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the authors

Jerome Louvel is cofounder and CEO of Restlet Inc. and Restlet SAS, and the creator of the Restlet Framework. Thierry Templier is an R&D architect at Restlet SAS, and a core developer of the Restlet Framework. Thierry Boileau is cofounder of Restlet SAS, an open source community manager, and core developer of the Restlet Framework.

about the cover illustration

The figure on the cover of *Restlet in Action* is captioned “An Arbanas man, from Zadar, Dalmatia, Croatia.” The illustration is taken from a reproduction of an album of Croatian traditional costumes from the mid-nineteenth century by Nikola Arsenovic, published by the Ethnographic Museum in Split, Croatia, in 2003. The illustrations were obtained from a helpful librarian at the Ethnographic Museum in Split, itself situated in the Roman core of the medieval center of the town: the ruins of Emperor Diocletian’s retirement palace from around AD 304. The book includes finely colored illustrations of figures from different regions of Croatia, accompanied by descriptions of the costumes and of everyday life.

Arbanas is an old Croatian family name associated with the town of Zadar, a seaport on the Adriatic coast in central Dalmatia, an area rich in Roman and Venetian history. The first settlement in this location dates to the Stone Age and the town became an important stop on maritime trading routes for Romans and Venetians, subsequently becoming part of the Austrian Empire, Italy, and Yugoslavia, before Dalmatia became part of an independent Croatian state. Today the ancient walled city is a popular tourist destination with its many churches, cultural attractions, and seaside location.

The figure on the cover is wearing blue woolen trousers and, over a white linen shirt, a red vest which is richly trimmed with the colorful embroidery typical for this region. A green embroidered jacket is slung over his shoulder and a red cap and leather moccasins complete the outfit.

Dress codes and lifestyles have changed over the last 200 years, and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants

of different continents, let alone of different hamlets or towns separated by only a few miles. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by illustrations from old books and collections like this one.

Part 1

Getting started

D

o you want to blend your web services, websites, and web clients into unified web applications, exposing and consuming RESTful web APIs? The three chapters in part 1 show how this is possible using the open source Restlet Framework and its unique Java API available in six consistent editions for Java SE, Java EE, OSGi, GAE, Android, and GWT!

The Restlet Framework supports the web in all its forms, in a simple and unified way. You can use it to both expose and consume web APIs, including web pages and web services. One benefit is that it supports all core HTTP features, making a radical difference when you stop adapting your applications to the web and instead naturally use its full power.

Chapter 1 gets you started immediately with the Restlet Framework. You'll develop your first Restlet "Hello World" program right away and then find out about the main features and benefits of this framework.

In chapter 2 we show you how to create fully featured Restlet applications, implementing an example mail system that you'll develop throughout the book. You'll learn how to put together major Restlet building blocks such as client-side and server-side resources, filters, and routers.

Because one of the strengths of the Restlet Framework is its ability to work in various technical environments, chapter 3 covers how to deploy Restlet applications on your own computers, using a simple Java SE virtual machine or a full-blown Java EE server—or even a dynamic OSGi container. You'll also see how to declaratively configure Restlet applications using XML and the Spring Framework.

That's a lot of exciting content, so let's get started!

Introducing the Restlet Framework



This chapter covers

- Writing your first Restlet client and server programs
- Restlet's features and benefits
- Overview of the Restlet Framework, an open source REST framework for Java

Let's say you headed a team that built a new kind of email service. It was a web application written in Java that made heavy use of servlets, and although initially it worked fine, it didn't scale well to larger loads when launched in the cloud. Your team had trouble responding to requests to expose the service to clients other than a browser using a SOAP-based web services stack. As a result, your service lost ground to more scalable and interoperable systems; your team was disbanded; and you were let go.

You're okay—you found a similar position at a better company with a shorter commute. Only one problem: the first thing they want you to do is head a team building...a new kind of email service! You don't want to repeat the same mistakes this time around.

What were those mistakes, anyway? You're pretty sure it wasn't the choice of programming language. You needed the type-safety, the rich built-in library, and the

access to powerful open source libraries that Java developers enjoy. Besides, Java is your core expertise, and it was easy to find competent Java programmers for your team. It wasn't that servlets didn't perform as promised, and it wasn't due to bad programming practices. In fact, it's hard to pinpoint any specific mistakes.

In the time between jobs, you read about REST and web API designs (see the Resources section near the end of the book, for more information about these). At first you think it's a step backward, working directly with HTTP, which seems like a lower-level layer, and having no easy access to anything like the session state you were used to with servlets. You're also understandably leery of buzzwords that can cause religious wars between purists. As you face this new design challenge, a few things about REST begin to click for you:

- *It's not fair to think of HTTP as a lower-level layer*—HTTP is a mature application-level protocol with support for advanced features like caching and conditional requests, compression, and negotiation between multiple languages and media types; the web depends on it directly for its existence. Although adding these features to your old service would have been prohibitively expensive, a RESTful design, done correctly, would be able to take advantage of them for free.
- *It's not fair to complain about the absence of session state in HTTP*—One of the things hampering the scalability of your old service was the need to share session state between servers in a cluster, which caused high latency as users' application states were shunted from server to server. Had the design been more RESTful, application state would have been encoded into the URLs and maintained by the client, leaving the servers dedicated to server-side state.

Maybe you're not entirely convinced, but at least you're willing to look further. You want to try a framework for building RESTful services and clients in Java that doesn't shut out other non-Java clients. You want to be able to deploy these services and clients in a variety of environments, including lightweight ones that don't involve Java EE. Even if your story isn't exactly like this (and we hope it isn't), the punch line is the same: you want Restlet.

An open source project for Java developers, Restlet Framework makes it as easy as possible for you to take advantage of REST and the web in Java. It provides an extensive set of classes and routines that you can call or extend, saving you from writing a great deal of code you would otherwise need to write and allowing you to focus instead on your domain requirements. Restlet helps you use the rich set of HTTP features such as content negotiation, caching, conditional processing, and secure authentication. It can open the doors of the web to you in all its forms—from the classic web to the semantic web, from web services to rich web clients and websites, from the mobile web to cloud computing.

In this chapter we show you some sample Restlet code based on the traditional “Hello World” on both the client and server sides. Then, we explain what the Restlet Framework is and why it's valuable. By the end of this opening chapter, you'll have a good overview of the framework, its design, and its capabilities.

Which version of Restlet Framework do I need?

The minimum version necessary to run all examples in the book is 2.0, but we recommend using version 2.1 or later for new developments. To get started, see appendix B for installation and IDE configuration instructions.

Where can I download the source code for the book examples?

The Restlet Framework distributions since version 2.0 contain the complete source code for the examples as well as all the dependencies necessary to run them. Once you have installed a copy of this distribution (choose the Java SE or Java EE edition of the framework to get started), you'll find the source code in the following directory:

```
/src/org.restlet.example/org/restlet/example/book/restlet/
```

Please note that the printed examples have been made as compact as possible for clarity of illustration. Real-life code would be made more robust by following typical programming practices.

1.1 “Hello World” with Restlet

In this section coding a “Hello World” program in Restlet will show you how *resources* are materialized in Restlet, on both the client and server side. We don't explain all the details right now. Our goal is to show you how basic HTTP clients and servers are developed with Restlet.

At this point it's important to note that *resources* is in italics in this chapter to clarify that we are talking about REST *resources*, the domain concepts identified by URIs and manipulated through HTTP standard method and representations such as HTML documents and JSON data. If you don't feel familiar enough with REST concepts, we encourage you to read appendix C, which briefly introduces the architectural style of the web.

1.1.1 Coding a ServerResource subclass

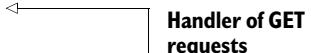
First, how does Restlet enable you to implement your own *resources* on the server side? In a few lines of code, listing 1.1 implements one that returns the “hello, world” string. It creates a subclass of the org.restlet.resource.ServerResource class.

Next, you add a represent() Java method, then use a Restlet-specific @Get annotation imported from the same org.restlet.resource package to expose it as a GET method defined in HTTP.

Listing 1.1 Creating a server-side resource

```
import org.restlet.resource.Get;
import org.restlet.resource.ServerResource;
```

```
public class HelloServerResource extends ServerResource {
    @Get
    public String represent() {
        return "hello, world";
    }
}
```



In this case the `@Get` annotation means “Here is the code to be run when a client wants to retrieve a representation of a *resource* implemented by the `HelloServerResource` class.” The name of the annotated Java method is flexible for your convenience. We used `represent()` here, but you can choose any other name as long as it doesn’t override a method of `ServerResource` or one of its supertypes, such as `get()`.

Note that the signature of an annotated Java method is important and requires you to understand the meaning of the associated HTTP method. An application receiving an HTTP GET request is required to provide a representation of the *resource* identified in the request by the URI. This is translated into the Java world by a call to a method that returns a value. Here the representation is a simple string, but you’ll see in chapter 4 how to return more complex representations such as XML or JSON documents.

Later in the book, we look at the entire set of existing annotations, how to extend it, and alternative ways to define behavior in `ServerResource` subclasses that don’t rely on annotations but on overriding regular Java methods.

1.1.2 *Running the server*

Once written, your Hello *resource* needs to be exposed and served. The `main()` method in listing 1.2 creates an HTTP server connector, which is an instance of the `Server` class available in the `org.restlet` package. It’s configured with the protocol, the listening socket port, and the target Restlet that will handle the requests—in this case, the `HelloServerResource` class.

After launching this Java program, you can point your web browser to the `http://localhost:8111` URI and get this line of text: “hello, world.”

Listing 1.2 Serving the Hello server resource

```
import org.restlet.Server;
import org.restlet.data.Protocol;

public class HelloServer {
    public static void main(String[] args) throws Exception {
        Server helloServer = new Server(Protocol.HTTP, 8111,
            HelloServerResource.class);
        helloServer.start();
    }
}
```

Passing the `HelloServerResource` class to the `Server` constructor might suggest that a kind of factory is used. That’s true. Each incoming request is handled by a new instance of `HelloServerResource`. It lets you focus on the behavior and ensure that

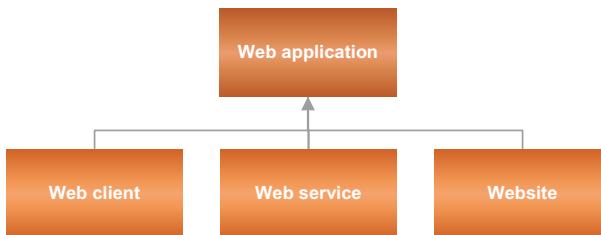


Figure 1.1 We use the term **web applications** to refer to web services, websites, and web clients.

each *resource* instance starts with a clean slate in a way that mirrors the statelessness of REST. As an additional benefit, it simplifies concurrent programming.

By design, a `Server` instance can't run without a `Protocol`. The listening port is optional, because each protocol defines a default port (for example, port number 80 for HTTP, 443 for HTTPS). Creating and starting a `Server` object is a way to associate a listening port to an object that will handle incoming requests. Such objects can be `ServerResource` subclasses, as illustrated earlier, but can also be instances of the `org.restlet.Restlet` class that I present in the next section. Focus your attention on the fact that a `Server` instance has a set of methods dedicated to its lifecycle—a `start()` and a `stop()` method.

Even though this program works fine, many features are lacking to make it a fully featured web application. All URIs, as an example, will be mapped to the same *resource* class, which isn't what you would do in a real application. You'll see in chapter 3 how you can map more specific URIs and how Restlet supports routing and filtering in complex situations such as virtual hosting.

At this point it seems important to clarify what we mean by a *web application*. In this terminology, as illustrated in figure 1.1, it covers web services in the sense of programmatic interactions over HTTP, static and dynamic websites, and even web clients in the sense of browser-hosted or standalone programs.

It's common for a web application to mix those types—something can be a web service and a web client at the same time. Let's now switch to the client side.

1.1.3 Using the `ClientResource` class

As mentioned earlier, Restlet not only is a server-side framework, but also gives you the ability to write clients that consume and manipulate *resources* exposed on the web. Let's illustrate this by accessing a sample server. If you run the following code, either in your IDE or your shell, the “hello, world” text will be displayed in the console.

Listing 1.3 Using a client-side resource

```

import org.restlet.resource.ClientResource;

public class HelloClient {

    public static void main(String[] args) throws Exception {
        ClientResource helloClientResource =
            new ClientResource("http://localhost:8111/");
    }
}
  
```

1 Create local proxy to resource

```

        helloClientResource.get().write(System.out);
    }
}

```

Print resource's representation to console

The `ClientResource` class is analogous to a `ServerResource` but located on the client side. This aspect is enforced by the fact that instances of this class are created with the URI of the *resource* ①. You can consider a `ClientResource` as a local proxy of the remote *resource*. The communication aspects are handled in an intuitive way, with Java methods that are simple transpositions of HTTP methods such as `get()` for GET and `delete()` for DELETE.

Note that the `ClientResource` class allows a series of methods to be invoked on the same target *resource*, automatically follows redirections, and can retry *idempotent* requests (requests that produce the same result when executed several times) when network errors occur.

Figure 1.2 illustrates the decomposition of *resources* between client side and server side. This separation is based on the uniform interface defined by REST and concretized by HTTP (see appendix C for details), which means the client side can interact with any server-side *resource* on the basis of predefined rules.

1.2 Overview of the Restlet Framework

The Restlet Framework, the main topic of this book, has been available since its launch in 2005 as an open source project at www.restlet.org. It's free, meaning you can use it without charge for in-house, commercial, and open source projects, subject to the relevant license agreement. It's also mature, meaning it has been under active development since its creation. It's well supported thanks to an extremely active community; questions asked on the mailing list are usually answered quickly.

In this section we briefly describe this framework, its main features and benefits, its overall design, and the target environments supported.

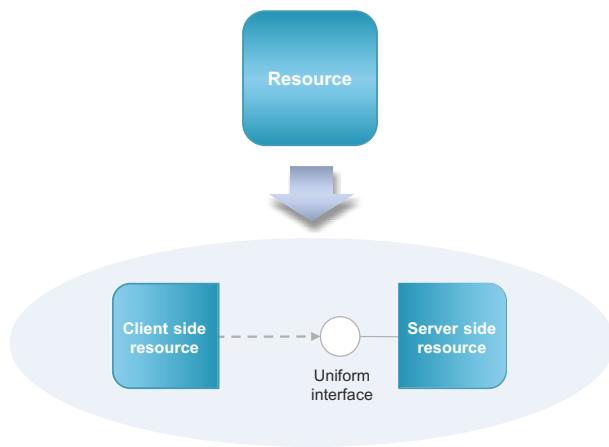


Figure 1.2 Decomposition of an abstract *resource* into Restlet artifacts

1.2.1 Main benefits of the Restlet API

The main feature of the Restlet Framework is its Restlet API, a compact and portable Java API located in an `org.restlet` package that embodies major REST concepts such as:

- *Uniform interface*—Standard way to interact with *resources* via requests and responses
- *Components*—Logical containers for Restlet applications
- *Connectors*—Enable communication between REST components using a protocol
- *Representations*—Expose the state of a REST *resource*

The Restlet API also abstracts the main features of the HTTP application protocol without requiring deep knowledge of HTTP methods, headers, and statuses:

- *Content negotiation*—Selects the best representation variant based on format (media type), language, or encoding, taking into account client capabilities as well as server preferences
- *Automatic compression*—Reduces the size of representations sent and expands compressed representations received using built-in HTTP capabilities
- *Partial representations*—Retrieves only the part you need via ranged requests, for download resumption or partial-update scenarios
- *Conditional processing*—Executes an HTTP method only if a given condition is met, such as a change in the signature (E-Tag) of the representation you want to retrieve
- *Caching*—Gives hints to clients about the way they should cache and update retrieved representations

The Restlet API is also thread-safe and designed for high concurrency and scalable deployment environments.

As illustrated in figure 1.3, this API has built-in support for routing that is both comprehensive and dynamic (in contrast to the Servlet API, which relies on static XML configuration or annotations for those aspects). It even supports virtual hosting in a way comparable to Apache httpd server but even more flexible. In addition, it comes with a complete web server for both static and dynamic web pages, blurring the distinction between web services and websites which merge into web applications exposing web APIs.

In addition, the Restlet API offers comprehensive support for security based on HTTP features including authentication, authorization (both coarse- and fine-grained), confidentiality (HTTPS, TLS/SSL, Google SDC), and access logging. This built-in feature reduces the complexity of your web projects, removing the need to select and learn a third-party security API for common security needs. But it's easy to integrate this API with other libraries using standard extensions for JAAS, OAuth, and OpenID, and there is support for authentication schemes such as Azure Shared Key and Amazon Web Services.

In conjunction with a large set of extensions, this single Java API lets you use several protocols such as POP3, SMTP, and pseudoprotocols such as FILE and WAR in a

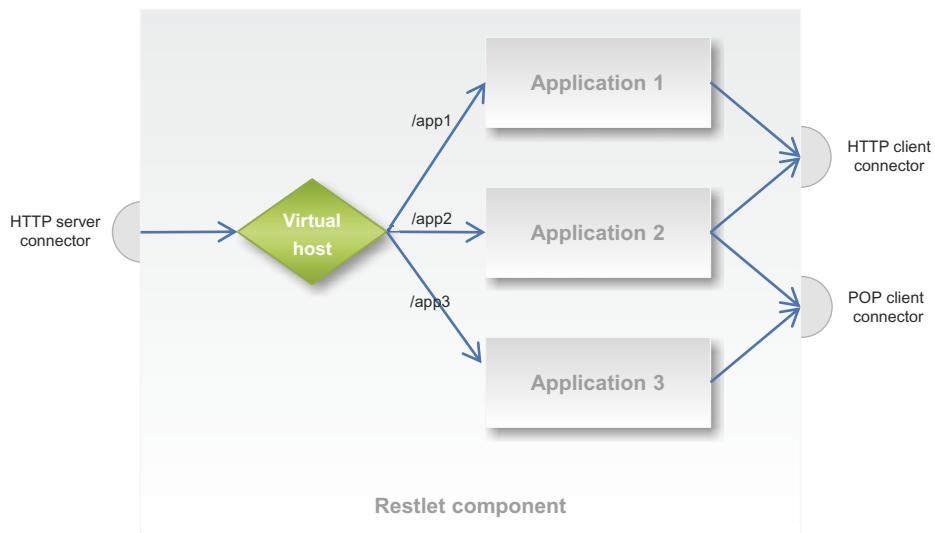


Figure 1.3 Example of one use of the comprehensive and modular Restlet architecture

REST-like way. Numerous representation types are supported for XML, JSON, and many more media types.

Finally, its unifying Java API is unique from many points of view in being usable both for server-side and client-side interactions—or even both at the same time, as with mash-up scenarios. This usability reduces the learning curve and increases productivity.

1.2.2 Overall design of the Restlet Framework

The Restlet Framework is composed of a lightweight core and a growing set of extensions. The core is distributed as a single `org.restlet.jar` file with a size of around 500 KB and which contains both the Restlet API and the Restlet Engine. Users typically write programs against the API, indirectly using the engine. But it's also possible to directly use the engine or extend its behavior by registering new plug-ins, such as connectors for new protocols or helpers for new authentication schemes.

As illustrated in figure 1.4, Restlet user projects can use a growing number of extensions adding standards support (like Atom, JAX-RS, JSON, RDF, and WADL), pluggable connectors (like POP3, SMTP, and FTP), or third-party integrations (like Apache FileUpload, FreeMarker, Jackson, JAXB, Spring, Velocity, and XStream).

As a result, the Restlet Framework offers a comprehensive solution thanks to its numerous extensions while keeping a simple and compact core. For additional details on those extensions, see appendix A, sections A.1–A.3, as well as the project Javadocs. We'll now describe the target environments supported by this framework.

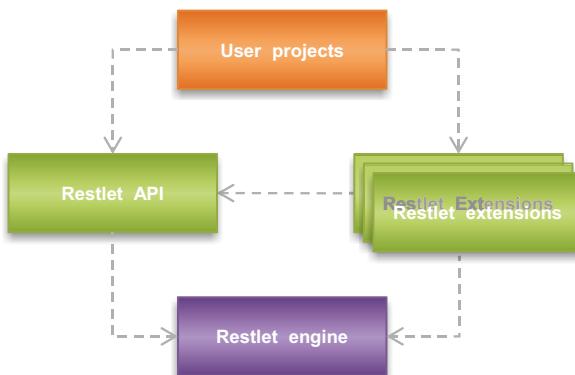


Figure 1.4 Overall Restlet design

1.2.3 Available editions and deployment targets

The Restlet Framework is Java-based software supporting REST and the all-embracing web. In addition to being usable on both client and server sides, with the HTTP protocol and others like POP3, SMTP, and FTP, the framework is available and supported on several Java-based platforms.

As shown in figure 1.5, those platforms are Java Standard Edition (Java SE) version 5.0 and above, Java Enterprise Edition (Java EE) version 5.0 and above, OSGi version 4.2 and above, Google App Engine (GAE), Android version 1.0 and above, and Google Web Toolkit (GWT) version 1.7 and above. You may not be familiar with all these platforms, so we'll introduce them next.

Java SE is the classic Java edition that you use when you install a JDK or a JRE. It's composed of a set of runtime libraries on which Java applications can be built and run. Java



Figure 1.5 Platforms supported by Restlet

EE builds on top of Java SE and adds a standard set of APIs such as the Servlet, JavaMail, JPA, and EJB APIs. We cover Restlet editions for Java SE and Java EE in parts 1 (chapters 1-3) and 2 (chapters 4-7) of this book. OSGi is a dynamic module system for Java made popular by its support in the Eclipse IDE, by Apache Felix, and by its use as a kernel of application servers such as JBoss.

GAE is a cloud computing service that lets you deploy your application on Google's own infrastructure, with virtually unlimited CPU power, network bandwidth, and storage capacities. We cover the Restlet edition for GAE in chapter 8.

GWT is an open source technology that lets you develop Rich Internet Applications for web browsers with no additional plug-in. You write and debug using the Java language, but a special compiler produces optimized JavaScript code for execution in a web browser. We cover the Restlet edition for GWT in chapter 9.

Finally there is Android, an open source OS based on Linux and Dalvik, a special virtual machine that can run Java-based programs on mobile devices. It's supported by Google and a large consortium called the Open Handset Alliance. We cover the Restlet edition for Android in chapter 9 as well.

As promised, this was a brief overview. If you want to get more details at this stage, we encourage you to read sections A.4 and A.5 of appendix A, which presents the various editions of the Restlet Framework, including the matrix of available extensions and the logical versioning scheme of the project.

Now that you've had a glance at Restlet programming, you're ready to move forward to more realistic Restlet application development in chapter 2.

1.3 **Summary**

In the context of the growing success of RESTful web APIs, the goal of the Restlet Framework is simple: making it as easy as possible for you to take advantage of REST and the web in Java. It provides an object-oriented framework that helps you use the rich set of HTTP features such as content negotiation, caching, conditional processing, and authentication.

This chapter gave you an overview of the Restlet API, a Java API abstracting REST and HTTP concepts and features and enriched by a growing number of Restlet extensions.

The Restlet Framework helps you build first-class RESTful systems, because it was designed from the start to support the REST style. You can use it to create RESTful web services, websites, or client programs, and deploy on a number of platforms including Java SE, Java EE, OSGi, GAE, GWT, and Android, thanks to its multiple editions.

Let's now continue our exploration of the Restlet Framework with chapter 2 and begin creating Restlet applications.



Beginning a Restlet application

This chapter covers

- Setting up a Restlet application
- Filtering and dispatching calls with Restlet's routing system
- Using client-side and server-side Restlet resources
- Method-overriding and annotation-based resource implementation

In chapter 1 we introduced the Restlet Framework with a basic demonstration of a single resource implementation. In this chapter you'll see the first structuring features of the framework with Restlet applications.

After learning the purpose of Restlet applications, you'll see how to set them up, how to filter and dispatch calls with the Restlet routing system, and how to use Restlet resources as targets or sources of calls. We also show how to implement resources using annotations as an alternative to overriding methods.

We want to begin with a solid foundation for developing Restlet applications, so we won't go into full details now; we cover aspects such as Restlet representations, security, and documentation later in the book. As in real projects, we'll need to iterate

several times before we end up with a fully featured Restlet application. For now, let's start with background information about Restlet applications.

2.1 The purpose of Restlet applications

The Restlet Framework can be used as an embeddable toolkit, as a library where you pick a few features that interest you (like the client connectors or the URI template support), or as a comprehensive framework that will fully take care of your web applications. In this chapter we explore the latter option, which is the most powerful one.

Restlet applications provide a way to implement a RESTful web API (commonly shortened to REST API) by grouping RESTful resource classes that share common data and services. Such applications can use third-party libraries and frameworks, such as Hibernate and EclipseLink for persistence and FreeMarker and Velocity for template representations, either directly or via one of the Restlet extensions.

Restlet applications aren't necessarily the largest elements of a Restlet solution; they can be part of Restlet *components* for deployment, as we explain in chapter 3. Figure 2.1 illustrates that Restlet applications are containers that provide a way to organize server and/or client resources and to route calls to/from them.

Note that we're talking about both the usual notion of web application and the particular concept of *application* in the Restlet Framework. The realization of the concept of web application as a Restlet class allows application code to be independent of these deployment aspects:

- Technical platform such as Java SE, Java EE, OSGi, or Google App Engine (GAE)
- Packaging solution such as JAR file, WAR file, or cloud deployment
- Domain name, root URI, and security context

Restlet applications also provide a way to modularize large RESTful systems into several pieces hosted in the same JVM or in separate ones. They're the main units of reusability for Restlet code.

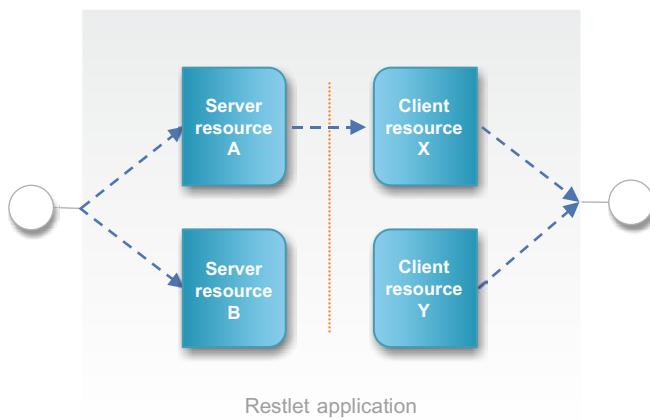


Figure 2.1 Restlet applications are containers of server resources and/or client

We're intentionally discussing server and client usage of Restlet at the same time. Most web frameworks and HTTP libraries are specific to the client side or the server side, but Restlet is unique in the way that it blurs this artificial separation. Consider that most web server applications also need to consume other web resources, becoming web client applications! With Restlet, when we talk about Restlet applications, you know they can be server side, both client and server side at the same time, or even client side.

In the next section, we get more concrete and explain how Restlet applications are structured into layers by the Restlet Framework.

2.2 The structure of Restlet applications

Restlet applications have multiple important purposes, so it's essential to understand how they're structured and used at runtime in Restlet projects. Figure 2.2 illustrates their overall design as three concentric layers, from the most generic and reusable on the edge to the most specific in the center. As mentioned, an application can handle both inbound server calls and outbound client calls, which are depicted using arrows.

An inbound call first goes through a service filtering layer, which is common to all resources contained in the application. This layer can handle things such as automatic decoding of compressed representations and support for partial representations. Then the call goes through a user routing layer, where you can do your own filtering (such as for authentication purposes) and you can route the request to a target resource, typically based on its URI. Finally, the call can reach the resource handling layer, where the target resource will handle the request and reply with a response that will return to the original client, following the same path. In addition, client calls can be initiated by applications, typically while handling server calls inside server resources, by creating client resources. A client resource sends a request outward through the layers, where it reaches some target resource (typically on a remote host,

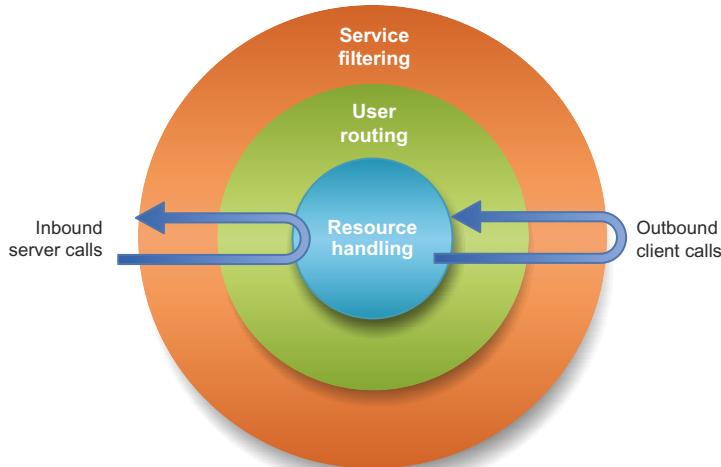


Figure 2.2 Restlet applications are structured into three concentric layers, processing inbound

and not necessarily a Restlet resource) and finally comes back with a response. The user routing layer will have a chance to filter or dispatch the request—for example, to handle automatic authentication—whereas the upper service filtering layer will be able to provide services such as automatic decompression of representations received in responses.

Now that you have a good understanding of the purpose of Restlet applications and their overall structure, let's get into the details of the design, based on figures 2.3 and 2.4. In each figure, three columns correspond to the three layers introduced earlier.

The service filtering column includes one or more filters, allowing the application to provide common features to all handled calls and all contained resources. We discuss those services in section 2.3.4.

The resource handling column is based on what you learned in chapter 1 when we created a simple subclass of `ServerResource` and used a `ClientResource`. This is the core of a RESTful application, and we cover it extensively in section 2.5.

The central user routing column is where application developers are given a chance to filter and route calls (perhaps for security purposes) and to dispatch them to target server resources based on URI templates. This is different from the filtering done in the service filtering column, which applies to all calls and target resources. Section 2.4 covers in detail how to route calls in Restlet.

For now, we would like to highlight again that user routing is primarily done on the server side but can also be useful on the client side, as illustrated in figure 2.4. For this purpose, we rely on two special elements to attach this routing logic: the *inbound root* and the *outbound root*, depicted as small circles in figure 2.3 and 2.4. They define

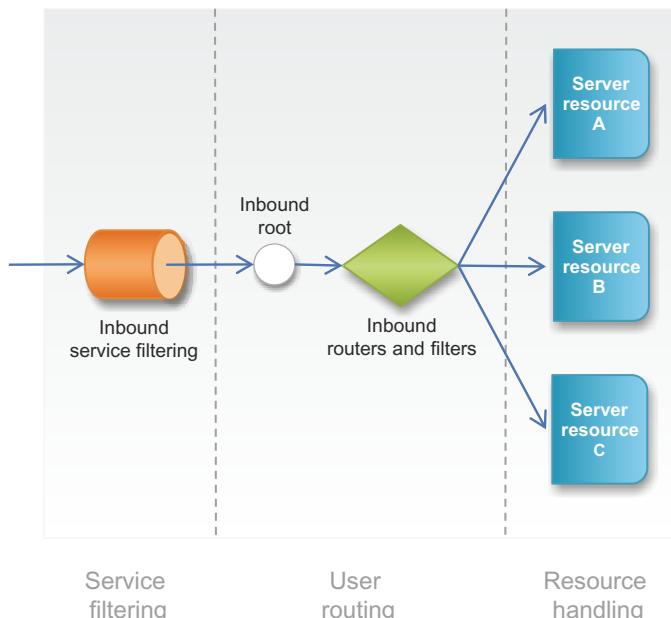


Figure 2.3 Server calls enter a Restlet application through the service filtering layer, continue into the user routing layer via the inbound root, then reach the target server resources.

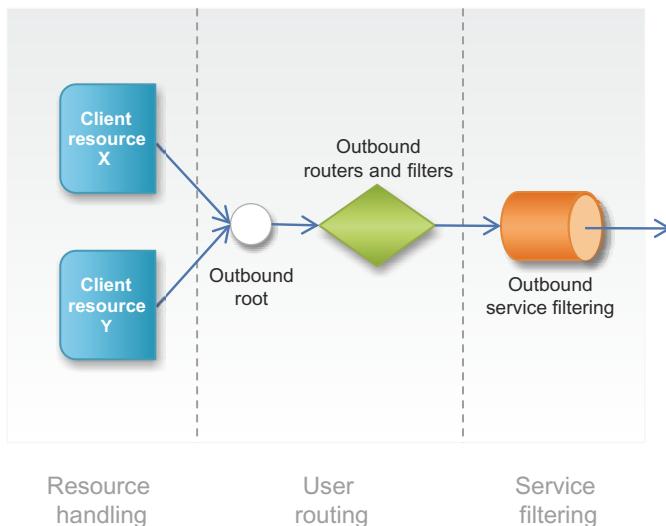


Figure 2.4 Client calls start from client resources, enter the user routing layer via the outbound root, and leave the application after going through the service filtering layer.

the starting points for user routing of calls received by server connectors in figure 2.3, and of calls sent to client connectors in figure 2.4.

At this point, you should have enough background information to understand the purpose and structure of Restlet applications. Let's now start coding your first Restlet application, extending the `org.restlet.Application` parent class.

2.3 Setting up a Restlet application

In this section we explain how Restlet applications are built by extending `org.restlet.Application`. We also present application context, properties, and common services and describe how to configure them. To illustrate what you'll learn, you'll implement a RESTful mail system.

Book example: RESTful mail system

In this book we'll progressively implement a RESTful mail system. For a more detailed description of this system, its requirements, and how to design its RESTful web API, see appendix D.

2.3.1 Creating an Application subclass

Going back to the RESTful mail system, let's start the implementation by creating a shell application for the RESTful mail server. As illustrated in the following listing, the first step is to create an `org.restlet.Application` subclass and add a static `main` method that creates an HTTP server connector, listening on port 8111 and transferring all calls to an instance of your application.

Listing 2.1 Providing the inbound root Restlet for the application

```

import org.restlet.Application;
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.Restlet;
import org.restlet.Server;
import org.restlet.data.MediaType;
import org.restlet.data.Protocol;

public class MailServerApplication extends Application {
    public static void main(String[] args) throws Exception {
        Server mailServer = new Server(Protocol.HTTP, 8111);
        mailServer.setNext(new MailServerApplication());
        mailServer.start();
    }

    @Override
    public Restlet createInboundRoot() {
        return new Restlet() {
            @Override
            public void handle(Request request, Response response) {
                String entity = "Method      : " + request.getMethod()
                    + "\nResource URI : "
                    + request.getResourceRef()
                    + "\nIP address   : "
                    + request.getClientInfo().getAddress()
                    + "\nAgent name   : "
                    + request.getClientInfo().getAgentName()
                    + "\nAgent version: "
                    + request.getClientInfo().getAgentVersion();
                response.setEntity(entity, MediaType.TEXT_PLAIN);
            }
        };
    }
}

```

Launch application with HTTP server

Create root Restlet to trace requests

The second method, overriding `createInboundRoot()`, is a factory method called by the framework when the application starts. It's in charge of creating the inbound root Restlet (shown as a small circle in figure 2.3). For now, you attach a simple Restlet that returns a string formatted from the method name, the target resource URI, and information on the user agent. If you run this example and point your browser to `http://localhost:8111/test/abcd`, a result page like this will be displayed:

```

Method      : GET
Resource URI : http://localhost:8111/test/abcd
IP address   : 127.0.0.1
Agent name   : Firefox
Agent version: 12

```

Now that you have a basic application working, it's time to explore what else Restlet applications have to offer, starting with their main properties.

2.3.2 Setting application properties

In this subsection we look more closely at the `org.restlet.Application` class and describe its main properties and methods. For visual reference, the UML class diagram in figure 2.5 displays the `Application` class below its parent class, `org.restlet.Restlet`.

Some of the properties and methods that we discuss, like the `name` and `description` properties and the `start()` method, come from the `Restlet` class and apply to other subclasses than `Application`.

Table 2.1 summarizes the properties inherited from `Restlet`.

The first four properties are optional, but it can be useful to set their values for debugging purposes and feedback in administration consoles. The other properties provide information on the execution environment of `Restlet` and its lifecycle. The following listing defines the constructor of the application by setting some common properties.

Note also that `org.restlet.Restlet` implements the `org.restlet.Uniform` interface, which defines the central method that must be implemented by all `Restlet` subclasses to handle inbound or outbound calls: `handle(Request, Response)`.

We can now look at the `Application` class and its additional properties. The `inboundRoot` and `outboundRoot` serve as communication hooks between the framework and the user routing layer (discussed previously and illustrated in figures 2.3 and 2.4). By default the `inboundRoot` is null; you have to supply something useful in order to handle incoming application calls. In the next section we explore some routing features available in `Restlet`. The `outboundRoot` property has a default value corresponding to the `clientDispatcher` property of the application's context, but you can easily change it—for example, to preauthenticate calls before sending them to the target secured remote resources. Applications have an important responsibility regarding security because they define the list of available roles using a `roles` property. Those roles can be “administrator,” “supervisor,” “salesRep,” “director,” or anything that makes sense in your application domain. The `roles` should be

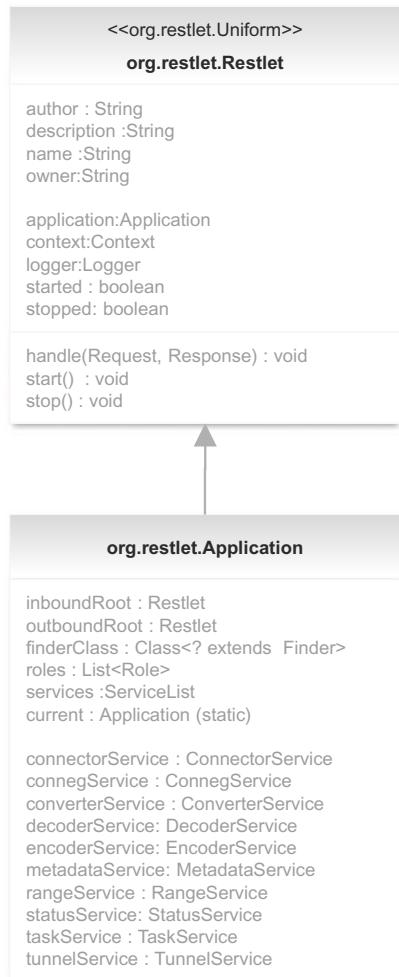


Figure 2.5 Class diagram of Application and its parent class, Restlet

Table 2.1 Properties of Restlet inherited by Application

Name	Description
author	Developer who is the author of this Restlet subclass
description	Description of the purpose of this Restlet subclass
name	Displayable name for this Restlet subclass
owner	Name of the organization owning the source code
application	Parent application containing this Restlet instance (read-only)
context	Execution context of this Restlet instance
logger	Logger to which warning, info, or debug messages can be sent
started	Indicates whether the Restlet instance was started (read-only)
stopped	Indicates whether the Restlet instance was stopped (read-only)

separate from the organization using the application, such as particular users and user groups. We'll come back to this important topic in chapter 5.

Listing 2.2 Setting basic application properties

```
public MailServerApplication(){
    setName("RESTful Mail Server");
    setDescription("Example for 'Restlet in Action' book");
    setOwner("Restlet SAS");
    setAuthor("The Restlet Team");
}
```



Then you find a static `current` property that lets you retrieve the current Application instance based on a variable stored in the current thread. Notice the `finderClass` property, which will be of interest if you want to integrate the Restlet Framework with your preferred Inversion of Control (IoC) container such as Spring or Guice, a topic we cover in section 4.4.

Let's continue our presentation of the properties in the next two subsections. First we describe the central `context` property that provides access to the application's environment. Then we'll cover the group of *Service properties listed in figure 2.5, which correspond to the service filtering layer.

2.3.3 Exploring the application context

Each Restlet application operates inside a given execution environment. This environment may vary significantly from one deployment of the application to another. For example, a given installation might use a standalone Java SE virtual machine, and another might use a complete Java EE Servlet container or even a cloud computing platform such as GAE. One deployment might use an LDAP directory to store user profiles, and another might prefer static files or a relational database.

To ensure the portability of your application and its isolation (from a security perspective) from other applications potentially hosted in the same JVM, you rely on the `org.restlet.Context` class. You'll normally share the same instance of this class between all the members of your application, which means between all objects of the user routing layer and between all client and server resources of the resource handling layer. For developers familiar with the Servlet API, the `org.restlet.Context` class is similar to the `ServletContext` class.

What does this Context class provide exactly? As summarized in figure 2.6, it first gives you two generic collections to store attributes and parameters. The `attributes` property is a map where you can add your own objects using unique names as keys (note that the `org.restlet` name prefix is reserved by the framework for its internal use). Note that by *property* we mean a Java member variable that is exposed through getter and setter methods, in this case `getAttributes()` and `setAttributes()`.

The `parameters` property is a `Series` of `Parameter` objects typically used for static configuration purposes. Each `Parameter` has a name `String` and a value `String`. Note that `Series` is a generic class from the `org.restlet.util` package implementing the `java.util.List<Parameter>` interface.

In addition, there are two dispatcher objects. The `clientDispatcher` provides a lower-level way to communicate with client connectors, such as sending HTTP requests. You generally don't use this property directly but instead rely on the higher-level `org.restlet.resource.ClientResource` class (introduced in chapter 1). More information will come in subsection 2.5.3, and you'll also learn how to add and configure client connectors in chapter 3, where we discuss deployment options for your Restlet applications.

Next the `serverDispatcher` property gives you access to a pseudoclient connector that lets you issue calls to the container of your application as if they were coming from a regular server connector like an HTTP server. The advantage of this approach over using a true HTTP client connector is that you bypass the entire network layer, resulting in performance improvements. We discuss this optimization topic in chapter 7.

What else does the `Context` class have to offer? It has a `logger` property, based on the standard `java.util.logging` API. This API can be used to hook many logging mechanisms such as the popular log4j and Logback libraries, if needed. If at any point in your application code you feel the need to log warnings, information messages, or debug traces, this is the property you should use. You'll read in chapter 3 about how the logging system can be configured to write rotating log files and more.



Figure 2.6 Class diagram of the `Context` class listing properties and special methods

Two properties relate to the security context of your application: the `defaultVerifier` property, to verify the credentials of a user wanting to authenticate, and the `defaultEnroler` property, to obtain the roles that apply to an authenticated user. We cover these features extensively in chapter 5 when you learn how to secure a Restlet application.

Two static class properties are available: `current`, returning the current context associated with the executing thread, and the related `currentLogger` property. These static methods can be useful in situations where you don't have direct access to the context, such as inside some third-party library callbacks.

To be exhaustive, there is also a `createChildContext()` method that lets you create an isolated child context—for example, to make sure two applications hosted in the same JVM remain isolated unless communication is explicitly allowed. By default, it clears all the properties and wraps the client and server dispatchers. You won't usually need to invoke this method yourself.

2.3.4 Configuring common services

Let's complete our review of Application properties with the group of `*Service` properties corresponding, as their name implies, to the application services part of the service filtering layer. The complete list is presented in table 2.2.

Table 2.2 Restlet applications offer several built-in services.

Name	Description
ConnectorService	Lets an application declare which connectors are expected to be available. It also provides callback methods for response lifecycle events (when a representation is about to be written, or when a representation is fully written), allowing the release of pending resources such as database transactions.
ConnegService	Provides a way to control the content-negotiation behavior at the application level. It offers two modes: strict and flexible (default). Other algorithms can be plugged in by overriding this class and reusing the Conneg subclasses in the Restlet engine.
ConverterService	Supports the automatic conversion between Restlet representations and regular Java objects. The conversion can work in both directions. We say more about this in section 2.4.
DecoderService	Provides automatic decoding or uncompressing of received entities (server-side requests or client-side responses). By default it supports the GZip and Deflate compression formats.
EncoderService	Provides automatic encoding or compressing of entities that are about to be sent (server-side responses or client-side requests). By default it supports the GZip and Deflate compression formats.
MetadataService	Gives access to metadata such as media types, character sets, languages or encodings, and their associated extension names. A list of default mappings covers most common cases. (An extension name can be used as a file extension or in URI parameters or in Restlet annotation values, as seen in section 2.3.4.)

Table 2.2 Restlet applications offer several built-in services. (continued)

Name	Description
RangeService	Automatically exposes partial resource representations. Server resources don't need to manually handle requested HTTP ranges; they can always return full representations that are then transparently cropped by this service, allowing the client to benefit from partial downloads.
StatusService	Handles error statuses. If an exception is thrown within your application or Restlet code, it will be intercepted by this service to customize the response status code and even the response entity to ensure a consistent look and feel.
TaskService	Launches tasks asynchronously with full Restlet context. The service instance returned won't invoke the runnable task in the current thread. In addition to allowing pooling, executing threads will have the thread local variables copied from the calling thread, ensuring that calls to methods like <code>Application.getCurrent()</code> work.
TunnelService	Rewrites request methods or client preferences. The tunneling can use URI query parameters, file-like extensions, or specific headers. This is particularly useful for browser-based applications that don't have full control over how HTTP requests are sent. See the example in the next section.

Application services are extensions of the `org.restlet.service.Service` class and provide inbound or outbound filtering of calls (logging or decoding of compressed request entities, for example). They can also support features common to all Restlets and resources of your application, like a metadata registry or the management of a thread pool.

The lifecycle of a service is the same as that of its parent application; it's started and stopped at the same time. In addition, a service can be enabled or disabled and is given a chance by the framework to provide both an inbound and an outbound filter to intercept calls made to or from your application.

You can register your own services if you need to, thanks to the `services` property of `Application`. It's also easy to retrieve them using the `getService(Class)` method. If your service is defined by the `MyService` class, you could register it using `myApplication.getServices().add(new MyService())` and retrieve it with `myApplication.getService(MyService.class)`.

Let's launch the application again and check that it does benefit from the default services we described. The default `TunnelService` lets you work around the limitations of some browsers that don't support all HTTP methods. It lets you specify a URI query parameter (named *method*) with the method name of your request. When issuing a GET request, it allows you to specify an OPTIONS method as an alternative; and when issuing a POST request, it allows you to specify any method, like PUT or DELETE.

To see the `TunnelService` in action, start the Restlet application from listing 2.1 and enter `http://localhost:8111/test/abcd?method=options` in your browser. You'll see the following result page:

```

Method      : OPTIONS
Resource URI : http://localhost:8111/test/abcd
IP address   : 127.0.0.1
Agent name    : Firefox
Agent version: 3.5.3

```

It's a nice result because this workaround is fully transparent to your application code. You receive an OPTIONS request as if it were issued by a fully capable HTTP client, and the target URI doesn't retain a trace of this extra URI query parameter. This illustrates one important goal of Restlet applications as the main unit of reusability and their ability to execute consistently in heterogeneous environments.

At this point we encourage you to look at the Javadocs of the Restlet API at www.restlet.org/documentation/ or in your Restlet distribution. They'll give you important details about each of the services in the `org.restlet.service` package, the features they provide, and the properties that can be customized. We'll continue to use these services during the rest of the book.

You've learned that Restlet applications are structured into three layers and discovered their various properties, including the group of `*Service` properties corresponding to the *service filtering* layer. The next section introduces you to the Restlet routing system, which is the basis of the *user routing* layer when filtering and dispatching calls.

2.4 The Restlet routing system

So far, the user routing layer you've used has been limited: a simple Restlet subclass tracing incoming calls. How can you build a filter that blocks some IP addresses or routes calls to target server resources based on a URI?

In this section we introduce the Restlet routing system and answer these questions. In particular, we present Restlet filters and routers, the two key elements of the routing system. Filters facilitate the preprocessing and postprocessing of calls, and routers dispatch calls to one of the several target routes available, typically based on a target URI.

2.4.1 Preprocessing and postprocessing calls with a Filter

On the server side, after receiving calls, or on the client side before sending calls, it's usual to systematically apply some behavior that doesn't depend on the target resource URI of the call. The common name for such processing elements is *filter*. Naturally the Restlet Framework offers an abstract `org.restlet.routing.Filter` class, which is a subclass of `org.restlet.Restlet`.

Restlet subclasses are multithreaded

Restlet subclasses must be thread-safe and support concurrent access. Writing code that behaves properly when executed by several threads isn't easy, and we highly recommend you read the book *Java Concurrency in Practice* [1] on this topic.

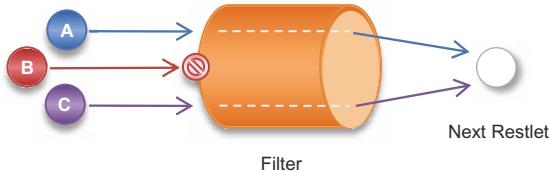


Figure 2.7 This filter is handling three concurrent calls, passing two of them to the next Restlet and blocking the third one.

Figure 2.7 illustrates three concurrent calls (A, B, and C) processed by the same Filter instance. Each thread supports a call (a request and response couple) and attempts to traverse the filter to reach the next Restlet. Sometimes the call will get blocked by the filter, and the thread will return immediately, without going to the next Restlet. (This is what `org.restlet.security.Authorizer`, a filter discussed later in the book, is designed to do.) In figure 2.7, call B is stopped by the filter as illustrated by the “stop” sign.

Like all Restlet subclasses, Filter’s entry point is the final `handle(Request, Response)` method declared in the Uniform interface. This method works in three steps. First it invokes the `beforeHandle(Request, Response)` method, which lets the filter preprocess the call. This method returns a result flag that indicates whether the processing should *continue* to the next Restlet, *skip* the next Restlet, or *stop* immediately and return up the thread stack.

In the first case, the filter invokes the next Restlet that was attached to it by invoking the `doHandle(Request, Response)` method (step 2). If no Restlet is attached, a “server internal error” (HTTP status 500) is set on the response.

When the next Restlet returns, or if the preprocessing asks to skip the next Restlet, the `afterHandle(Request, Response)` method (step 3) is invoked, giving the filter a chance to postprocess the call. Postprocessing is typically based on the response received—for example, to compress the representation returned for an HTTP GET method.

The following listing shows a concrete example that provides a blocking filter based on IP addresses. Note that you use a special thread-safe implementation of the `Set<String>` interface to store the list of blocked IP addresses.

Listing 2.3 An IP address-blocking filter

```
public class Blocker extends org.restlet.routing.Filter {
    private final Set<String> blockedAddresses;                                | List of blocked
                                                                           | IP addresses
    public Blocker (Context context) {
        super(context);
        this.blockedAddresses = new CopyOnWriteArrayList<String>();           |
    }                                                                           | Verify
    @Override                                                               | client IP
    protected int beforeHandle (Request request, Response response) {          | address
        int result = STOP;
        if (getBlockedAddresses()
            .contains(request.getClientInfo().getAddress())) {
            response.setStatus(Status.CLIENT_ERROR_FORBIDDEN,
        }
    }
}
```

```

        "Your IP address was blocked");
    } else {
        result = CONTINUE;
    }
    return result;
}

public Set<String> getBlockedAddresses() {
    return blockedAddresses;
}
}

```

To test this filter, you need to update your application, in particular the `createInboundRoot()` method, to return an instance of the `Blocker` filter instead of the `Tracer` class that contains the same logic as in listing 2.1 but packaged as a separate class extending `Restlet`.

Supporting IP v6 network addresses

If your OS is configured by default to use an IPv6 network stack, such as recent Linux distributions, you'll have to either adjust values such as `127.0.0.1` to the IPv6 format or switch back to an IPv4 stack, with this JVM setting:

```
System.setProperty("java.net.preferIPv4Stack", "true");
```

You can see in the following listing how the filter is returned as the new inbound root and attached to a `Tracer` instance using the `setNext(Restlet)` method.

Listing 2.4 An IP address-blocking filter

```

@Override
public Restlet createInboundRoot() {
    Blocker blocker = new Blocker (getContext());
    // blocker.getBlockedAddresses().add("127.0.0.1");
    blocker.setNext(new Tracer(getContext()));
    return blocker;
}

```

The next listing contains the simple `Tracer` class that responds to incoming requests with information on the resource URI, the client IP address, and user agent.

Listing 2.5 Restlet returning common request properties to the client

```

public class Tracer extends Restlet {

    public Tracer (Context context) {
        super(context);
    }

    @Override
    public void handle(Request request, Response response) {
        String entity = "Method      : " + request.getMethod()

```

```

+ "\nResource URI : "
+ request.getResourceRef()
+ "\nIP address    : "
+ request.getClientInfo().getAddress()
+ "\nAgent name    : "
+ request.getClientInfo().getAgentName()
+ "\nAgent version: "
+ request.getClientInfo().getAgentVersion();
response.setEntity(entity, MediaType.TEXT_PLAIN);
}
}

```

To test the filter, launch the updated application and point your browser to `http://127.0.0.1:8111/test/abcd`. It should return the same result as the previous listings. Now uncomment the second line in the method in listing 2.5, and restart your application. If you refresh your browser page, this time you should see the message “Your IP address was blocked,” indicating that the `Blocker` filter worked.

In addition to the base `Filter` class, the Restlet API comes with several filters ready to use, as shown in figure 2.8. The `Extractor` filter provides a way to extract values from request cookies, query parameters or posted forms, and store them as request attributes for easier processing down the road. Also, the `Validator` filter can verify the presence of specific attributes and then validate the format of their value.

There are two security-related filters: `Authenticator`, to ask the user to supply credentials and verify them when provided; and `Authorizer`, to enforce access policies. In chapter 5 we explain those last two filters in detail.

Note that these filters should be used when their behavior is applicable to a set of target resources; otherwise, it’s better to add this logic inside your resources directly. There is a fifth filter called `Route` and its common `TemplateRoute` subclass, which aren’t displayed in figure 2.8 because they’re rarely used directly. Instead you use them indirectly via the `Router` class, which is the topic of the next subsection.

2.4.2 Using a router to dispatch calls based on URIs

So far we’ve shown how inbound or outbound calls can be filtered to apply preprocessing or postprocessing. What’s missing from this puzzle is a way to direct calls to target resources, routing them based on properties like the target URI. This is the purpose of the `org.restlet.routing.Router` class, a direct subclass of `Restlet` that

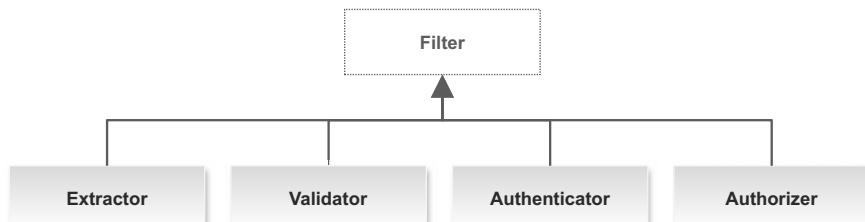


Figure 2.8 Class diagram showing common `org.restlet.routing.Filter` subclasses

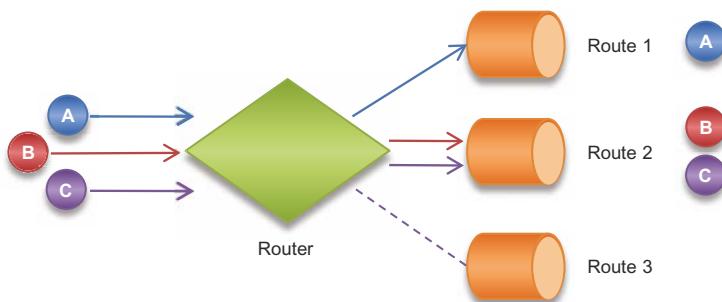


Figure 2.9 The router is handling three concurrent calls and dispatching them to attached routes.

consists of a list of routes and a set of properties that define how the routing of an incoming call to one of the routes happens.

Figure 2.9 shows how three calls (A, B, and C) supported by three concurrent threads enter a Router instance to end into one of the three attached routes (1, 2, and 3), but not necessarily at the same time. In this illustration, call A reaches route 1, and calls B and C reach route 2. If you pay attention to the form of the figure used for routes, you'll recognize the cylinder that we used for filters.

The class used is `TemplateRoute` in the same package, which defines a next Restlet like all filters, but also a `score(Request, Response)` method. As its name implies, this method computes an affinity score based on the matching of the URI of the target resource (contained in the `Request.resourceRef` property) with a given URI template. URI templates look like regular URIs, with the addition of variable parts, as in `http://localhost:8111/accounts/{accountId}`. We introduce them in detail in appendix D when explaining how to define the URI space of a web API.

To illustrate what you've learned, in the following listing you run a simple example based on the `Tracer` and `Blocker` classes previously developed and attach them to a Router instance.

Listing 2.6 Illustrating URI-based routing

```

@Override
public Restlet createInboundRoot(){
    Tracer tracer = new Tracer (getContext());
    Blocker blocker = new Blocker (getContext());
    blocker.getBlockedAddresses().add("127.0.0.1");
    blocker.setNext(tracer);

    Router router = new Router(getContext());
    router.attach("http://localhost:8111/", tracer);
    router.attach("http://localhost:8111/accounts/", tracer);
    router.attach("http://localhost:8111/accounts/{accountId}", blocker);

    return router;
}

```

In this scenario you create two Restlet instances: a Tracer instance and a Blocker filter with that Tracer instance as its next Restlet. Then, based on URI templates, you define which Restlet to dispatch to when the request's target URI matches a given route. Note that `TemplateRoute` instances aren't created explicitly; you use the `attach(String, Restlet)` method instead. Let's start the updated application and test a few URIs in a browser, as detailed in table 2.3.

Table 2.3 Results of a routing example

URI retrieved	Result observed
<code>http://localhost:8111/</code>	Trace results are displayed.
<code>http://localhost:8111/?param=value</code>	Trace results are displayed, including the query string.
<code>http://localhost:8111/123</code>	Error: "The server has not found anything matching the request URI."
<code>http://localhost:8111/accounts/</code>	Trace results are displayed.
<code>http://localhost:8111/accounts/123</code>	Error: "Your IP address was blocked."

One of the nice features of `Router` and `TemplateRoute` is that the URI variables are automatically extracted and stored in the attributes map of the request. In the last example URI, the string "123", corresponding to the `{accountId}` variable in the URI template `http://localhost:8111/accounts/{accountId}`, will be stored in an attribute named `accountId`. You'll use this when you handle the call in a `ServerResource` subclass to retrieve the underlying domain object from the database.

In some cases it's necessary to customize the way the URI matching happens. The `Router` class has a full range of options to take care of this need. The `defaultMatchingMode` property indicates whether only the beginning of the URI must match (see the `org.restlet.routing.Template.STARTS_WITH` constant) or whether the entire URI must match (see the `EQUALS` constant). The latter case is the default mode since Restlet 2.0, so be careful if you upgrade from a previous version.

Another `Router` property is `defaultMatchQuery`, which indicates whether the query string at the end of target URIs (the characters after the optional ? character) should be considered during matching. By default its value is *false* because the order of query parameters is rarely enforceable. Restlet developers tend to prefer handling them inside the resource classes.

When the router evaluates each route, it's possible to define when it considers a given route as the one to select and pass the call to. This is the purpose of the `routingMode` property, which can have one of the following values (those constants are defined in the `Router` class):

- Best match
- First match (default since Restlet 2.0)
- Last match

- Random match
- Round robin
- Custom

This arrangement opens the way for many interesting uses, such as writing a load-balancer of calls to several instances of a back-end web service.

In addition to Restlet instances, you can attach server resource classes directly to a Router using an `attach (String, Class<? Extends ServerResource>)` method, as illustrated in figure 2.10. But what exactly happens inside this method?

Filter and Router are the basic classes of the Restlet routing system. A third class, Redirector, will be presented in section 8.3. Now that you know how Restlet applications are structured and how *service filtering* and *user routing* layers work, let's transition to the *resource handling* layer. This layer, composed of Restlet client and server resources, is the heart of any Restlet application.

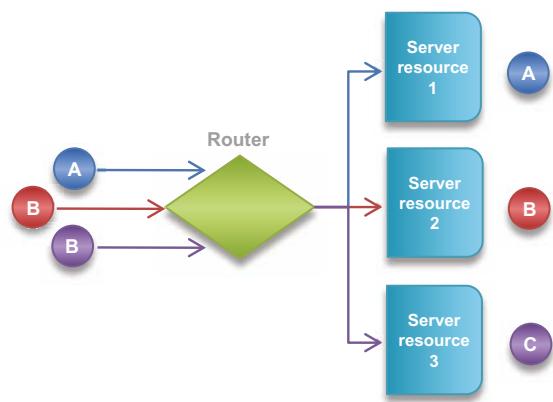


Figure 2.10 Router dispatching concurrent calls to three target server resources

2.5 Using Restlet resources in an application

In this section you continue the development of an initial Restlet application by using Restlet resources to handle calls. The `org.restlet.Resource` base class, renamed from `UniformResource` in version 2.1, encapsulates the state of all resources and exposes it through the uniform REST interface, typically composed of standard HTTP methods like GET, PUT, DELETE, and POST. We cover the server-side handling of incoming calls with the `ServerResource` class and then switch to the client side with the `ClientResource` class, which is used to make outgoing calls.

We also will explain how the five Java annotations supported by the Restlet API offer a high-level way to express the functionality of client and server Restlet resources. We then briefly compare the Restlet API with the JAX-RS API standard and update our example application fully using annotations.

2.5.1 Resource, the base of all resources

The `org.restlet.Resource` abstract class is the base class of all Restlet resources. It was designed to be as close as possible to the idea of a web resource in REST, the intended conceptual target of a URI reference and the main element of RESTful web APIs. In contrast to the `org.restlet.Restlet` class introduced in chapter 1, which is designed to support concurrent calls, a separate `Resource` instance is needed each time a particular resource is handled.

In practice, the state of a resource in Restlet Framework is composed of the call's request, the call's response, the parent application's context that we introduced earlier, and some user-provided state such as a persistent bean or a database result set.

In addition to its internal state, the `Resource` class provides many convenient methods to manipulate the request and response objects. Table 2.4 shows a few examples. Refer to the Javadocs of the class for the complete list of methods available.

Table 2.4 Shortcut methods in the Resource class

Method name	Equivalent call
<code>getCookies()</code>	<code>getRequest().getCookies()</code>
<code>getCookieSettings()</code>	<code>getResponse().getCookieSettings()</code>
<code>getReference()</code>	<code>getRequest().getResourceRef()</code>
<code>getQuery()</code>	<code>getReference().getQueryAsForm()</code>
<code>getQueryValue(String)</code>	<code>getQuery().getFirstValue(String)</code>
<code>setQueryValue(...)</code>	<code>getReference().setQuery(getQuery().set(...))</code>

The `Resource` class defines three lifecycle methods: `init(Context, Request, Response)` to provide the initial state of the resource; the `handle()` method to trigger the processing of the associated call, either on the client or the server side; and the `release()` method to prepare the resource for collection by the garbage collector. Three additional helper methods are provided to facilitate the customization of the resource initialization (the `doInit()` method), the release (the `doRelease()` method), and each time an exception or an error is caught (the `doCatch(Throwable)` method).

2.5.2 Using ServerResource as target of calls

A subclass of `Resource` that helps develop server-side RESTful web resources is `org.restlet.resource.ServerResource`. As explained in the previous subsection, all Restlet resources have a precise lifecycle composed of initialization, handling, and releasing steps.

In the case of `ServerResource`, it's the responsibility of the `org.restlet.resource.Finder` class to enforce this lifecycle when it instantiates a new resource instance to ask it to handle a call. You'll rarely manipulate a `Finder` directly, because it's implicitly created by the `Filter.setNext(Class)` and `Router.attach(String, Class)` methods, but it's nonetheless essential to clearly understand its role. Figure 2.11 illustrates this sequence of method invocations.

The `Finder` class stores the class name of the target resource and creates a new instance of it for each incoming call, following the factory design pattern. This class provides the transition from the user routing layer to the resource handling layer.

To make you feel fully comfortable with these aspects, let's change the `MailServerApplication` to attach a resource class to a router. Note in the following listing

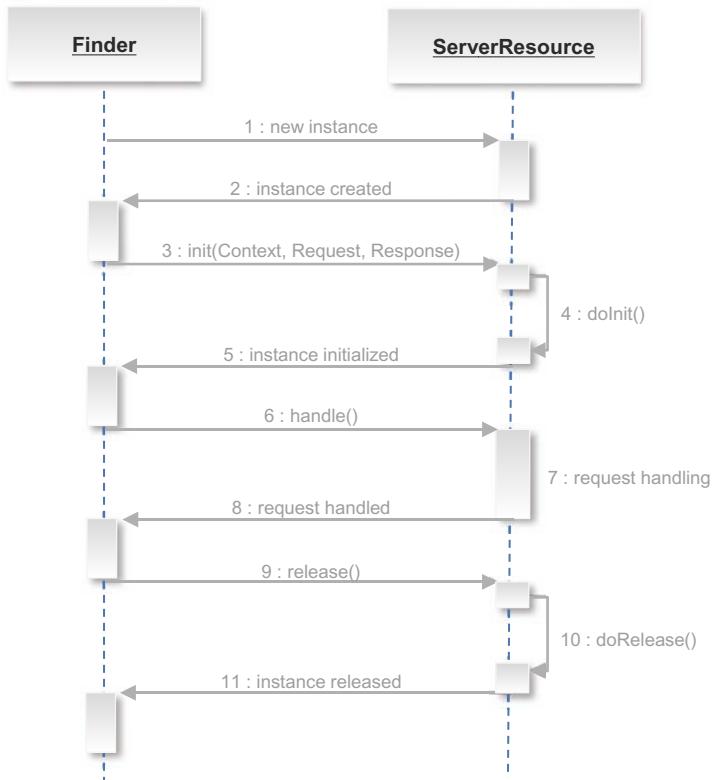


Figure 2.11 Lifecycle of a server resource for a given call

how you pass the `.class` value of the `ServerResource` subclass and not an instance of that class.

Listing 2.7 Routing to server resources

```

@Override
public Restlet createInboundRoot() {
    Router router = new Router(getContext());
    router.attach("http://localhost:8111/",
        RootServerResource.class);

    return router;
}
  
```

In addition to the methods found in its superclass, `ServerResource` provides shortcut methods that are useful when updating the response, as illustrated in table 2.5. Two common application services are also easily accessible via the `getConverterService()` and `getMetadataService()` methods.

Regarding security, a convenience `isInRole(String)` method will help you test whether the authenticated user has been granted a given application role (we cover securing Restlet applications in detail in chapter 5). The idea is to make your development life easier by putting the most important information in your hands when you

Table 2.5 Shortcut methods in the ServerResource class

Method name	Equivalent call
getAttribute(String)	getRequestAttributes().get(String)
redirectPermanent(String)	getResponse().redirectPermanent(String)
redirectTemporary(String)	getResponse().redirectTemporary(String)
setStatus(Status)	getResponse().setStatus(Status)
setStatus(Status, String)	getResponse().setStatus(Status, String)

develop those server resources. Additional properties are available to configure the default behavior of your server resources, as summarized in table 2.6.

Now you define the RootServerResource class to make it trace the steps of the resource lifecycle in the following listing. To simplify matters, you disable content negotiation in the constructor—otherwise you'd have to override the `get(Variant)`

Table 2.6 Special properties of ServerResource

Name	Description
annotated	Indicates whether annotations are supported. If your resource doesn't use them, you might see a small performance gain by setting this property to false. By default it's true. Resource annotations are discussed in section 2.5.4.
autoCommitting	Indicates whether the response should be automatically committed. When processing a request on the server side, setting this property to <code>false</code> lets you ask the server connector to wait for <code>commit()</code> to be explicitly called before sending the response back to the client. Only used for asynchronous call processing.
committed	Indicates whether the response has already been committed. Only used for asynchronous call processing.
conditional	Indicates whether conditional handling is enabled. Conditional handling is a feature of HTTP that lets you specify conditions on methods such as GET and PUT in order to prevent the retrieval of an unchanged representation or the update of an identical one. By default it's <code>true</code> .
existing	Indicates whether the resource identified by this instance of <code>ServerResource</code> exists. By default it's <code>true</code> . If the underlying domain object doesn't yet exist, you can set this property to <code>false</code> , and the client will automatically receive a "Not found client error" (HTTP status code 404) in response.
negotiated	Indicates whether content negotiation is enabled. Content negotiation is a powerful feature of HTTP that lets you define several representation variants for the same resource and automatically pick the best based on client preferences. By default it's <code>true</code> . We discuss this in chapter 4.
variants	Modifiable list useful for content negotiation to declare the supported representation variants. See chapter 4 for more details on its use.

and options(Variant) methods instead of get() and options(). We cover content negotiation in chapter 4.

Listing 2.8 Illustrating server resource lifecycle

```
import org.restlet.representation.Representation;
import org.restlet.representation.StringRepresentation;
import org.restlet.resource.ResourceException;
import org.restlet.resource.ServerResource;

public class RootServerResource extends ServerResource {

    public RootServerResource () {
        setNegotiated(false);                                ← Disable content
        // setExisting(false);                                negotiation
    }

    @Override
    protected void doInit()throws ResourceException {
        System.out.println("The root resource was initialized.");
    }

    @Override
    protected void doCatch(Throwable throwable) {
        System.out.println("An exception was thrown in the root resource.");
    }

    @Override
    protected void doRelease()throws ResourceException {
        System.out.println("The root resource was released.\n");
    }

    @Override
    protected Representation get()throws ResourceException {
        System.out.println("The GET method of root resource was invoked.");
        return new StringRepresentation("This is the root resource.");
    }                                              ← Handle OPTIONS showing impact of throwing exception
    @Override
    protected Representation options()throws ResourceException {
        System.out.println("The OPTIONS method of root resource was
invoked.");
        throw new RuntimeException("Not yet implemented");
    }
}
```

In addition to the constructor—the doInit(), doRelease(), and doCatch(Throwable) methods—the resource overrides two methods to handle HTTP methods: get() and options() for HTTP GET and OPTIONS handling. Note that additional delete(), post(Representation), put(Representation), and head() methods are available.

When receiving a GET call, you print a trace in the console and return a string representation using org.restlet.representation.StringRepresentation. In chapter 4 we present many more types of representations available in Restlet, but for now don't worry about this line and think of it as a wrapper around a Java string that will be sent back to the client.

When receiving an OPTIONS call, you also print a trace. But instead of returning a representation, you throw an exception indicating that this method isn't yet implemented in order to test the doCatch(Throwable) method.

Let's launch the new application and try to retrieve the following URIs in a browser: `http://localhost:8111/` and then `http://localhost:8111/?method=options`. Feel free to comment out the lines attaching resource classes like Account, because they will be introduced late in this chapter. The result displayed in the console confirms our explanations on the lifecycle and handling methods:

```
The root resource was initialized.  
The GET method of root resource was invoked.  
The root resource was released.
```

```
The root resource was initialized.  
The OPTIONS method of root resource was invoked.  
An exception was thrown in the root resource.  
The root resource was released.
```

Uncomment the `setExisting(false)` line in the constructor, restart the application, and retrieve `http://localhost:8111/`. This time, instead of displaying "This is the root resource," the browser displays "The server has not found anything matching the request URI" and returns an HTTP 404 status code, as expected!

There is more to say about ServerResource regarding content negotiation and the use of Java annotations to mark handler methods. We cover this topic in section 2.5.4, but for now let's switch to the other side and discuss the ClientResource class.

2.5.3 Using ClientResource as source of calls

Client-side resources are supported by the ClientResource class, the other subclass of Resource, presented in section 2.5.1. It acts as a proxy of a target resource. It can work with any remote server-side resource implemented in any technology for which there is a protocol defined and a Restlet connector available, such as HTTP, POP3, or FTP.

We present this class as a higher-level client because, in addition to the connectivity offered by lower-level client connectors (Restlet elements concretely implementing a specific protocol, something we haven't described yet), it offers features such as a retry strategy for failed requests, an ability to automatically follow redirection or the transparent serialization between Restlet representations sent or retrieved, and higher-level Java objects as representations.

Like ServerResource, ClientResource has its own shortcut methods to conveniently manipulate the wrapped request, as illustrated in table 2.7.

Table 2.7 Shortcut methods in the ClientResource class

Method name	Equivalent call
<code>setAttribute(String, Object)</code>	<code>getRequestAttributes().put(...)</code>
<code>setChallengeResponse(...)</code>	<code>getRequest().setChallengeResponse(...)</code>

Table 2.7 Shortcut methods in the ClientResource class (continued)

Method name	Equivalent call
setConditions(Conditions)	getRequest().setConditions(Conditions)
setCookies(Series<Cookie>)	getRequest().setCookies(Series<Cookie>)
setReference(String)	getRequest().setReference(String)

ClientResource also provides special properties to control its behavior, as detailed in table 2.8.

Table 2.8 Special properties of ClientResource

Name	Description
followRedirects	Indicates whether redirections should be automatically followed. Note that potential infinite loops are detected for you. By default it's true.
maxRedirects	Indicates the maximum number of HTTP redirects that can be automatically followed. By default it's 10.
next	The next Restlet processing the calls. By default it's the clientDispatcher property of the current context. If no context is available, a client connector is automatically instantiated based on the scheme protocol of the target URI.
requestEntityBuffering	Indicates whether transient or unknown-size request entities should be buffered before being sent. This is useful to increase the chance of being able to resubmit a failed request due to network error or to prevent chunked encoding from being used.
responseEntityBuffering	Indicates whether transient or unknown-size response entities should be buffered after being received. This is useful to be able to systematically reuse and process a response entity several times after retrieval.
retryOnError	Indicates whether idempotent requests (such as GET, HEAD, and OPTIONS) should be retried when a recoverable error is detected. By default it's true.
retryAttempts	Number of retry attempts before reporting a recoverable error. By default it's 2.
retryDelay	Delay in milliseconds between two retry attempts. By default it has a value of 2 seconds.
parent	The parent resource based on the parent URI of the current client resource.

It's now time to put those features in action with an example. The following listing creates an instance of ClientResource pointing to the resource at the `http://localhost:8111/` URI. You then attempt to retrieve a representation of this resource by invoking the

`get()` method, which returns an instance of the `Representation` class that you can display on the console thanks to a shortcut method, `write(OutputStream)`.

Listing 2.9 Illustrating features of client resources

```
import org.restlet.resource.ClientResource;

public class MailClient {

    public static void main(String[] args) throws Exception {
        ClientResource mailRoot =
            new ClientResource("http://localhost:8111/");
        mailRoot.get().write(System.out);
    }
}
```

If you have your mail server application from previous examples still running, stop it first because we want to illustrate what happens when a remote resource is unreachable. Then launch the `MailClient` program for the first time. Here's the console output, based on Restlet default logging messages, with timestamps removed for clarity:

```
Starting the internal [HTTP/1.1] client
A recoverable error was detected (1000), attempting again in 2000 ms.
A recoverable error was detected (1000), attempting again in 2000 ms.
Exception in thread "main" Connection Error (1000) - Unable to establish a
connection to localhost/127.0.0.1:8111
at org.restlet.resource.ClientResource.doError(ClientResource.java:639)
at org.restlet.resource.ClientResource.handleInbound(...)
at org.restlet.resource.ClientResource.handle(ClientResource.java:1101)
at org.restlet.resource.ClientResource.handle(ClientResource.java:1076)
at org.restlet.resource.ClientResource.handle(ClientResource.java:980)
at org.restlet.resource.ClientResource.get(ClientResource.java:685)
at org.restlet.example.book.restlet.ch02.sec5.sub3.MailClient.main(...)
```

As you can see, an HTTP client connector is automatically started, which is normal because no context is available; this is the case because you don't run inside a parent Restlet application, and the `next` property isn't set. Then you see two traces indicating that a recoverable error was detected with a 1000 status code and two retries were attempted. This status code isn't a standard HTTP status but a special status returned by Restlet when a connector error occurs, in this case meaning the communication with the HTTP server couldn't even be established. The `org.restlet.data.Status` class has more details about the statuses, their code, and their description.

Start the mail server application from section 2.3.2 with the `setExisting(false);` line commented out in the `RootServerResource` class. Launch the mail client again. This time the program runs successfully and displays "This is the root resource" in the console. Stop the application, uncomment the `setExisting(false);` line, and run the app again. This time the program throws a runtime `ResourceException` corresponding to the HTTP status code 404 (Not Found).

Other HTTP methods are supported in the same spirit: `delete()`, `head()`, `options()`, `post(Representation)`, and `put(Representation)`. It's also possible to directly invoke

the underlying `handle()` method, but you need to first make sure the method is set manually by using `setMethod(Method)`. In this case exceptions won't be automatically thrown when the response status is an error.

If you look at the Javadocs, you'll notice that all those methods we cite have additional signatures including an extra `Class` parameter and returning an instance of this class. This might puzzle you initially, so here is a usage example: `String result = mailRoot.get(String.class)`. If you run it, the `result` variable will contain the same "This is the root resource" string. The extra parameter indicates the desired result when converting from the resource representation; `ClientResource` attempts to do the conversion automatically, with the help of the `converterService` mentioned earlier (and presented in detail in chapter 5). Used in combination with resource annotations, the topic of the next subsection, this automatic conversion service can yield powerful results.

2.5.4 Higher-level resources with Java annotations

Even though the `Resource`, `ServerResource`, and `ClientResource` classes provide a convenient API for implementing RESTful resources, it's tempting to go even farther down the abstraction road by taking advantage of Java annotations.

As a member of the expert group at the Java Community Process (JCP) that helped define the JAX-RS API, Jérôme Louvel realized that although annotations could be a powerful abstraction, at the same time there were drawbacks to an API mainly based on annotations: the code seemed less readable, extending the default behavior was harder, supporting dynamic applications wasn't easy, and how the framework worked was less obvious.

With the help of the Restlet community feedback, we found a middle ground that builds on the classic Restlet API and adds the best of the annotation-based JAX-RS API. This resulted in the addition of 5 annotations in version 2.0 of the Restlet API (compared to the 21 annotations in JAX-RS 1.1), one for each HTTP method, as detailed in table 2.9.

Table 2.9 Restlet provides annotations for defining resources

Name	Description
<code>@Get</code>	Annotation for methods that retrieve a resource representation. Its semantics are equivalent to an HTTP GET method. No input entity allowed.
<code>@Put</code>	Annotation for methods that store submitted representations. Its semantics are equivalent to an HTTP PUT method. Input entity allowed.
<code>@Delete</code>	Annotation for methods that remove representations. Its semantics are equivalent to an HTTP DELETE method. No input entity allowed.
<code>@Post</code>	Annotation for methods that accept submitted representations. Its semantics are equivalent to an HTTP POST method. Input entity allowed.
<code>@Options</code>	Annotation for methods that describe a resource. Its semantics are equivalent to an HTTP OPTIONS method. No input entity allowed.

Each of these annotations has an optional value that lets you restrict the metadata of the response entity, typically the media type but also (since version 2.1) other metadata such as the language, character set, or encoding. Those metadata are specified using their extension names as specified in the `MetadataService` and not their full name. For the JPEG media type, you would use a `jpg` annotation value instead of the `image/jpeg` MIME type name.

Request entities can also be specified to have different metadata than response entities using the `:` separator, and alternative variants can be specified using the `|` separator, such as `@Post(json|form:html)` to accept entities posted in either JSON or web form media type, returning an HTML response entity.

Also, a single variant can specify multiple metadata using the `+` separator, such as `@Get(xml+fr)` to retrieve an XML representation in French. Since version 2.1, it's possible to use the same annotation for multiple query parameters by adding the query string in the annotation value, such as `?light` or `?logLevel=fine` right after the optional value specifying the supported metadata.

You can put these annotations in action by rewriting the `RootServerResource` to take advantage of the `@Get` and `@Options` annotations, as illustrated in the following listing.

Listing 2.10 Illustrating resource annotations

```
import org.restlet.resource.Get;
import org.restlet.resource.Options;
import org.restlet.resource.ServerResource;

public class RootServerResource extends ServerResource {

    @Get ("txt")
    public String represent(){
        return "This is the root resource";
    }

    @Options ("txt")
    public String describe(){
        throw new RuntimeException("Not yet implemented");
    }
}
```

We encourage you to run the application and test the GET and OPTIONS methods again to observe that the result is the same or similar depending on your Restlet Framework version. In contrast to the previous approach of overriding the `get()` and `options()` methods of `ServerResource`, you're free to name the methods as you want: `represent()`, `toString()`, `toText()`, or anything you like. Another benefit is that you can return representations as regular Java objects and not as Restlet Representation subclasses as before.

This conversion is possible thanks to the extensible `ConverterService`. This service knows, for example, how to convert a `String` into a `StringRepresentation` but

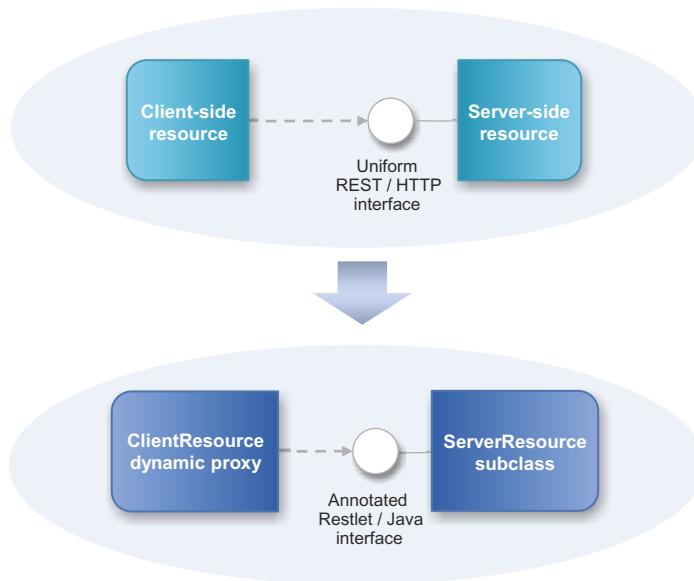


Figure 2.12 Restlet-annotated Java interfaces can be used by client resources as a client proxy or when implementing a server resource subclass.

can go much further and convert plain Java objects (POJOs) into XML or JSON representations. More on this topic in chapter 4, where we discuss representations.

Another difference compared to JAX-RS 1.1 is that those annotations are usable on the client side as well! In fact, they let you express your resource uniform interfaces as annotated Java interfaces. The *design implementation* step can then become straightforward because your RESTful web API will result in a set of annotated Java interfaces, one for each resource class. Once they're defined, you can use those interfaces either with the ClientResource to create a dynamic client proxy or with a subclass of ServerResource, implementing it as illustrated in figure 2.12.

Let's take advantage of this new approach by refactoring RootServerResource. First, extract an annotated interface that you name RootResource, as in the following listing.

Listing 2.11 Annotated Java interface for the root resource

```
import org.restlet.resource.Get;
import org.restlet.resource.Options;

public interface RootResource {
    @Get ("txt")
    public String represent();

    @Options ("txt")
    public String describe();
}
```

Now that you've defined the contract as a Java interface, you can refactor the RootServerResource class to implement it as shown in the following listing.

Listing 2.12 Implementing the Java annotated resource interface

```

import org.restlet.resource.ServerResource;

public class RootServerResource
    extends ServerResource implements RootResource {

    public String represent() {
        return "This is the root resource";
    }

    public String describe() {
        throw new RuntimeException("Not yet implemented");
    }
}

```

You can launch your application again and verify that it still behaves the same. Now you have one more artifact to maintain—the annotated Java interface—but what are the benefits compared to a single annotated ServerResource subclass? First, it's easier to create those interfaces based on a designed web API than to develop shell server resources. Also, those interfaces can be put in a common package and immediately used on the client side to build client toolkits or unit tests without having to wait for a testable server-side implementation, even as a mock-up.

Let's look at how to use this interface on the client side. The idea is to take advantage of a little-known feature of Java called *dynamic proxies* (see the `java.lang.reflect.Proxy` class introduced in Java 1.3), which let you dynamically provide an implementation of a given interface. Based on this mechanism, the `ClientResource` provides methods that return a proxy instance of an annotated interface as illustrated in the following listing.

Listing 2.13 Creating dynamic proxies based on annotated Java interfaces

```

import org.restlet.resource.ClientResource;

public class MailClient {

    public static void main(String[] args) throws Exception {
        RootResource mailRoot = ClientResource.create(
            "http://localhost:8111/", RootResource.class);
        String result = mailRoot.represent();
        System.out.println(result);
    }
}

```

What's remarkable is that you can directly invoke the `represent()` method, which transparently issues an HTTP GET call to the remote `RootServerResource` implementing this exact same annotated Java method! The URI of the target resource was provided to the `create()` method: `"http://localhost:8111/"` in this case. Like the static `create(String, Class)` method, other ones are provided by `ClientResource` including the `wrap(Class)`, `getChild(String, Class)`, and `getParent(Class)` methods.

Restlet API vs. JAX-RS API

In addition to its much broader features scope—like its ability to work on the client side and the server side, its availability in several editions (Java SE, Java EE, OSGi, GWT, GAE, and Android), its powerful routing and security APIs, and its ability to use several protocols besides HTTP—the Restlet API comes with the best of Java annotations usage for RESTful Java development based on the experience of the JAX-RS API design by the JCP expert group. Plus its classic inheritance-based design makes it easier to learn, debug, and extend.

Despite our criticisms of the JAX-RS API, we recognize its important role as a JCP-approved standard in broadening the adoption of REST in Java land. The Restlet Framework even provides a solid implementation of JAX-RS, contributed by Stefan Koops. This implementation ships as the `org.restlet.ext.jaxrs` extension and makes it possible for JAX-RS applications to use the Restlet API at the same time.

2.5.5 Updating the example mail application

To finish this chapter, let's update the `MailServerApplication` to achieve a more comprehensive working application. As a first step, let's implement the three top resources (modeled in figure D.14, found in appendix D)—the Root, Accounts, and Account resources—with the exact HTTP methods that they should support.

To store the accounts, use an in-memory list of strings, with the account name as sole value for now. You'll also decouple the application with a specific deployment environment, to make it more reusable, by attaching the resources relative to a parent URI. The following listing contains the updated application.

Listing 2.14 Updating the MailServerApplication

```
public MailServerApplication(){
    setName("RESTful Mail Server application");                                ← Set basic properties
    setDescription("Example application for 'Restlet in Action' book");
    setOwner("Restlet SAS");
    setAuthor("The Restlet Team");
}

@Override
public Restlet createInboundRoot() {                                              ← Dispatch calls to
    Router router = new Router(getContext());
    router.attach("/", RootServerResource.class);
    router.attach("/accounts/", AccountsServerResource.class);
    router.attach("/accounts/{accountId}", AccountServerResource.class);
    return router;
}
```

To better organize this growing example, we have decided to split the classes into three packages. We recommend that you follow a similar structure for your own Restlet projects. Note that instead of packages, you may prefer separate projects:

- *Common* for the annotated Java interfaces and related classes shared by the client and server sides

- *Server* for your application and the server resources implementing the interfaces in “common”
- *Client* for containing the mail clients developed in the next chapter

Let's now update the interfaces in the “common” package, still based on the exposed method defined in the resource model of appendix D. The following listing shows the result.

Listing 2.15 Annotated interfaces in the “common” package

```
public interface RootResource {
    @Get ("txt")
    public String represent();
}

public interface AccountsResource {
    @Get ("txt")
    public String represent();

    @Post ("txt")
    public String add(String account);
}

public interface AccountResource {
    @Get ("txt")
    public String represent();

    @Put ("txt")
    public void store(String account);

    @Delete
    public void remove();
}
```

Now that the REST API is defined in terms of a Java API annotated with Restlet-specific markers, let's provide the server-side implementations (see the following listing). The code should be self-explanatory. Let's highlight how you obtain the `accountId` value in the `AccountServerResource` directly from the request attributes. This value is automatically extracted from the resource URI by the Restlet Framework because you attached the resource to your application router with the URI template `/accounts/{accountId}`.

Listing 2.16 Implementation of annotated interfaces in the “server” package

```
public class RootServerResource
    extends ServerResource
    implements RootResource {

    public String represent() {
        return "Welcome to the " + getApplication().getName() + " !";
    }
}

public class AccountsServerResource
    extends ServerResource
    implements AccountsResource {
```

```

private static final List<String> accounts =
    new CopyOnWriteArrayList<String>();

public static List<String> getAccounts() {
    return accounts;
}

public String represent(){
    StringBuilder result = new StringBuilder();

    for (String account : getAccounts()) {
        result.append((account == null) ? "" : account).append('\n');
    }

    return result.toString();
}

public String add(String account) {
    getAccounts().add(account);
    return Integer.toString(getAccounts().indexOf(
        account));
}
}

public class AccountServerResource
    extends ServerResource
    implements AccountResource {

private int accountId;

@Override
protected void doInit() throws ResourceException {
    this.accountId = Integer.parseInt(getAttribute("accountId"));
}

public String represent() {
    return AccountsServerResource.getAccounts().get(
        this.accountId);
}

public void store(String account) {
    AccountsServerResource.getAccounts().set(
        this.accountId, account);
}

public void remove() {
    AccountsServerResource.getAccounts().remove(this.accountId);
}
}

```

Return static list of accounts stored in memory

Mail account resource implementation

Retrieve account identifier based on URI variable

Note that this implementation is simple but naïve because the index of accounts in the list is used in their URI. If you delete one account, the URI of the remaining accounts changes, which isn't useful because URIs aren't stable and can't be usefully bookmarked. A better implementation would use a unique and stable identifier, part of a more complex account structure (for example, based on relational database sequences or UUIDs).

Also, because you changed the attachment URIs to remove the `http://localhost:8111` prefixes, you cannot test the application directly, as in the previous sections.

This is intentional and only temporary; you’re now in good shape to begin the next chapter and see how you can concretely deploy this example application using Restlet components.

2.6 **Summary**

In this chapter you learned how to begin the development of Restlet applications. You saw their purpose as structuring elements of a RESTful web API project, illustrating why Restlet Framework is not merely a toolkit or a library where you can pick features as needed. You saw that Restlet applications are containers of RESTful web resources on both the server side and client side, and how they provide the main units of reuse across heterogeneous deployment environments.

You learned how Restlet applications are structured in three concentric layers and looked at the details of each layer, including the special role of `inboundRoot` and `outboundRoot` as anchors for the user routing layer:

- The outermost service filtering layer provides services common to all contained resources, such as automatic decoding of compressed representations and support for partial representations.
- The middle user routing layer is where inbound or outbound calls can be filtered (for example, for authentication purposes) or dispatched based on a target URI.
- The innermost resource handling layer is where a target server resource handles a request and replies with a response and where a source client resource can issue calls to remote resources or other local resources.

In addition you applied this background information in developing a concrete Restlet application, and you saw the properties available, including the special context property that helps isolate an application from specific deployment environments, and the `*Service` properties that allow configuration and extension of the service filtering layer.

We continued the exploration with the user routing layer, introducing the Restlet routing system that allows preprocessing and postprocessing of both inbound and outbound calls with the `Filter` class, and call dispatching to routes attached using the `Router` class.

Finally, you saw how resource handling works, using both server-side and client-side resources as instances of `ServerResource` and `ClientResource`, both extending `Resource`. You saw that those resources encapsulate generic state, such as the call request and response and the parent context, and also user-specific state, such as persistent domain objects. We explained how the Restlet Framework uses five Java annotations to provide an alternative way to develop Restlet resources, using an annotated Java interface as a realization of a RESTful web API.

You’re now well advanced in the “getting started” part of this book, which ends with a chapter explaining how Restlet applications can be deployed in a standalone Java SE virtual machine, in a Java EE container such as Apache Tomcat, or in OSGi environments such as Eclipse Equinox.



Deploying a Restlet application

This chapter covers

- Background on Restlet components
- Deploying in standalone Java SE virtual machines
- Configuring virtual hosts and log and status services
- Declarative XML configuration
- Deploying in Java EE application servers and OSGi environments

In chapter 2 you learned how to develop a Restlet application, the basic unit of reuse in the Restlet Framework. Your goal now is to deploy those applications to local machines in your organization and to make sure that they can be properly used and tested.

In this chapter you'll learn the purpose of a Restlet component and how it's structured in four layers. You'll then see how to set up a Restlet component in a Java SE environment, adding server and client connectors, exploring virtual hosting and internal routing, and configuring logging and default status messages.

As an alternative to using the Restlet API with Java code, we introduce two ways to use XML for more declarative configuration of Restlet components—one built in

the Restlet Framework and another relying on the Spring Framework. Then we explain how to deploy the same Restlet application to a pre-existing Java EE server. Finally we mention a third deployment approach based on OSGi.

This chapter isn't the last one covering deployment; chapter 8 shows how Restlet applications can be easily deployed in cloud computing infrastructures such as Google App Engine, Amazon EC2, or Microsoft Azure.

3.1 The purpose of Restlet components

When we introduced REST in chapter 1, we mentioned the role of REST components as coarse-grained distributed architecture elements that communicate between themselves using connectors and network protocols such as HTTP. We also indicated how the Restlet Framework directly embodies those elements in its API. Like chapter 2, where we illustrate how REST resources map to instances of Restlet Resource class (specifically its subclasses, ClientResource and ServerResource), this chapter introduces the Restlet Component, Connector, Client, and Server classes.

Figure 3.1 expands figure C.5 from appendix C (introducing the REST architecture style in detail) to illustrate how a Restlet component interacts with other REST components using client and server connectors. Note that the other components need not be developed using Restlet; the interoperability is solely based on the communication protocols implemented by the connectors and on the data exchanged (the resource representations).

Figure 3.1 tells us another important thing about the purpose of a Restlet component: it's a container of Restlet applications. The containment plus the connectivity offered to applications makes the Restlet components the perfect vehicles to deploy and run applications on local or remote infrastructures.

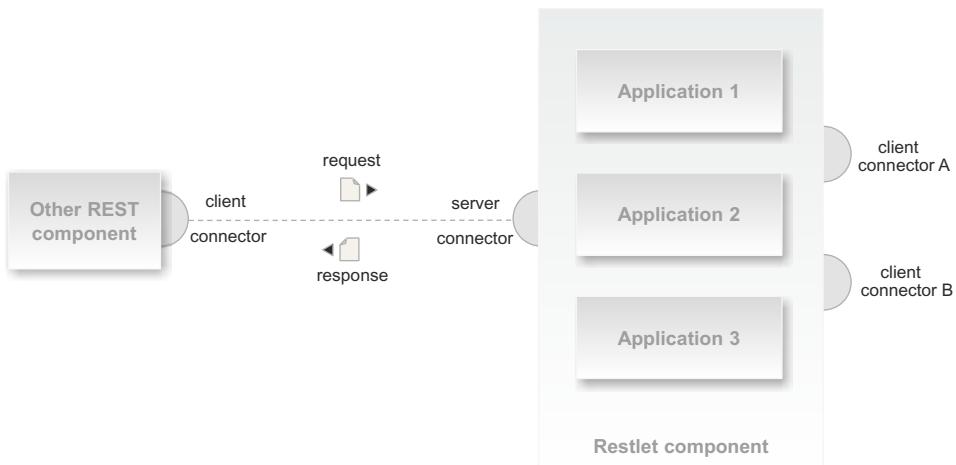


Figure 3.1 Restlet components are containers of Restlet applications and connect them to other distributed REST components.

As you'll see in the next sections—where the purpose of applications is to ensure portability of user routing logic and contained resources—the purpose of components is to deploy those applications and adapt them to specific target environments, such as Java SE and Java EE. This ensures, for example, that your applications aren't tied to a specific IP address or domain name, thanks to the virtual hosting capabilities of components.

Also Restlet components provide, through the `Realm` class, the ability to abstract contained applications from security aspects specific to a given deployment environment, such as how and where user credentials (logins, passwords, certificates, and so on) are stored (LDAP directory, relational database, local file, and so forth) and which application roles are granted to authenticated users. More details on the security aspects are explained in chapter 5.

In the next section we get more concrete and explain how Restlet components are organized in layers.

3.2 *The structure of Restlet components*

As in section 2.2, where we discuss the structure of Restlet applications, the structure of Restlet components is organized in concentric layers, which are progressively more specific in their function as you move toward the center. Restlet components are like applications in being able to handle both inbound and outbound calls, as illustrated in figure 3.2 with large arrows.

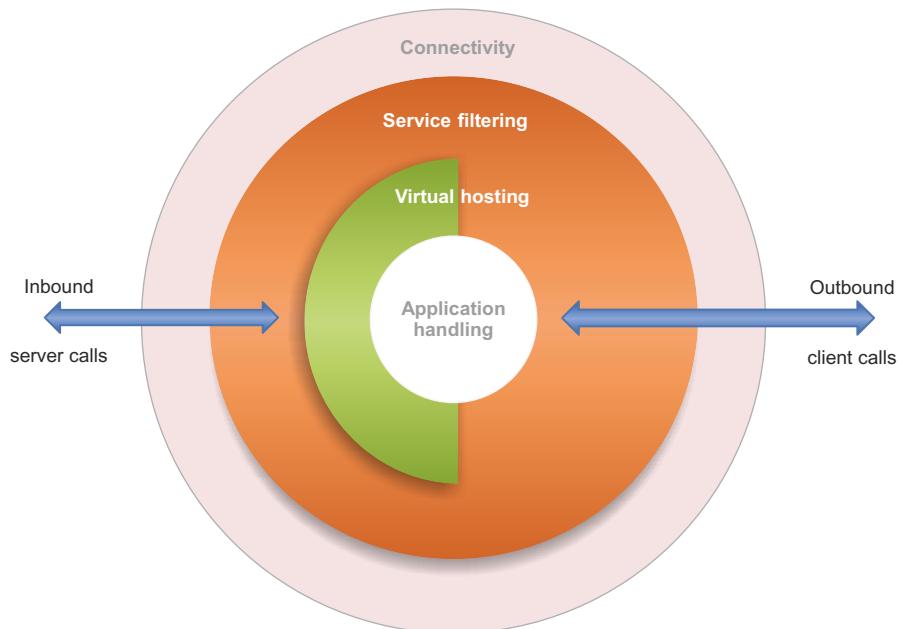


Figure 3.2 Restlet components are structured into four concentric layers, processing inbound and outbound calls in logical steps.

An inbound call is first received by the connectivity layer, which handles transport aspects of network protocols, such as HTTP persistent connections or FTP connection establishment. Then the call goes through the service filtering layer, similar to the one found inside each Restlet application. By default, this layer provides global access logging, similar to the HTTP access logs produced by traditional servers like Apache HTTP or Microsoft IIS. Next the call enters into the virtual hosting layer, which is only relevant on the server side.

Virtual hosting

This is a common mechanism found in HTTP servers helping to serve several domain names and websites from a single IP address and port number. It's useful when the number of available IP addresses is limited on a single machine and you still want to use the default ports of protocols. For example, HTTP uses 80 as its default port, so you can omit :80 in HTTP URLs.

Once the call has been dispatched to one of the declared virtual hosts, it can enter into the application handling layer. This layer is the Restlet application handling described in chapter 2.

On the other side, a client call is typically issued by a contained application in the application handling layer. It goes directly to the service filtering layer because virtual hosting doesn't apply to outbound calls. It then enters the connectivity layer, where it's picked up by the matching client connector, based on either an explicit call protocol or a protocol implied by its context (for example, the target URI of the call). Client connectors are reused across all outbound calls of all applications and must be scalable and stateless to ensure isolation between applications.

To illustrate the call flow, figure 3.3 expands on the Restlet component portion of figure 3.1 and provides more detail. Two virtual hosts are declared, and the three application instances are attached to them using relative URI paths (for example, /app3). Note that the same application instance can be attached to several virtual hosts. (It's also possible to instantiate a Restlet application class several times, attaching each instance to its own virtual host, typically for data-isolation purposes.)

Also note that an internal router is shown with a forbidden link from the server connector. This is a private router to your component that can only be accessed for internal calls; such calls don't go through the local network loopback. The internal router is closely related to the Restlet Internal Access Protocol (RIAP) pseudoprotocol that we take advantage of in section 7.5.2 when talking about modularizing large Restlet systems.

At this point you should have enough background information to understand the purpose and structure of Restlet components. Let's write a first Restlet component using the Restlet edition for Java SE and the example mail server application from the end of chapter 2.

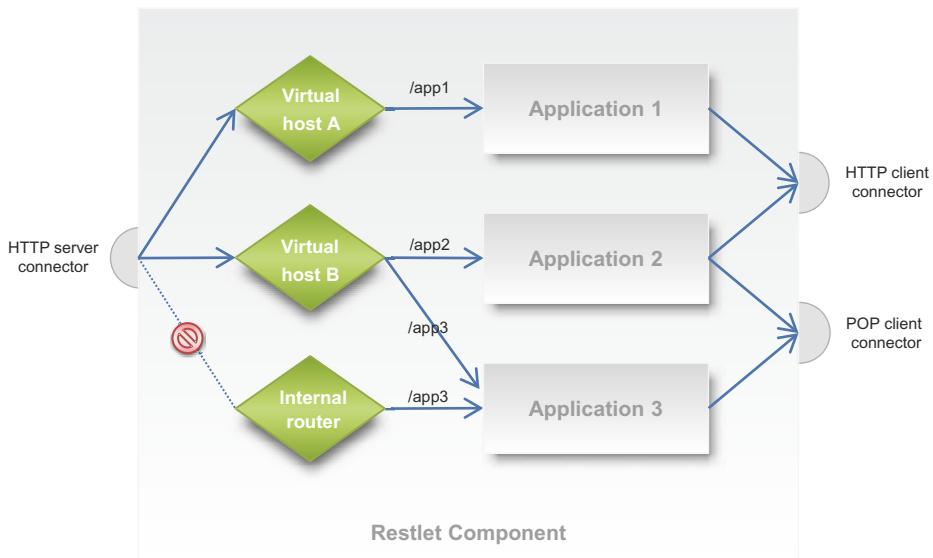


Figure 3.3 A closer look at the content of a Restlet component

3.3 Standalone deployment with Java SE

In this section we explain how to use Restlet components to deploy standalone applications on a regular Java SE virtual machine, which is the simplest way to get started with Restlet deployment, but still a powerful one. For this purpose we use the Restlet edition for Java SE, which is best suited for development phases, when you need instant turnaround between coding and testing; for embedded scenarios, when the lightest footprint is required; and in situations where you have complete access to the server machine and want the maximum flexibility regarding which connectors to use.

Minimum Java SE version

The main requirement for the version of the Restlet Framework covered by this book (2.1) is Java SE 5.0 and above.

You'll see how to set up an `org.restlet.Component` object via Java code, how to add client and server connectors, how to set up virtual hosts, and how to configure the common services, such as the default log and status services.

3.3.1 Creating a Component subclass

Starting with the Restlet mail server application and resources defined in section 2.5.5, we'll create a Restlet component. The first step will be to extend `org.restlet.Component`. In listing 3.1 you add an HTTP server connector to the component's list

of servers and attach your application to the default virtual host. Note that Restlet components are typically configured in their constructor, whereas applications are configured in an overridden `createInboundRoot()` method, and resources in an overridden `doInit()` method.

Listing 3.1 Creating the MailServerComponent to deploy the application

```
import org.restlet.Component;
import org.restlet.data.Protocol;

public class MailServerComponent extends Component {           ← RESTful component containing mail application

    public static void main(String[] args) throws Exception {   ← Launch mail server component
        new MailServerComponent().start();
    }

    public MailServerComponent() {
        setName("RESTful Mail Server component");
        setDescription("Example for 'Restlet in Action' book");
        setOwner("Restlet SAS");
        setAuthor("The Restlet Team");

        getServers().add(Protocol.HTTP, 8111);                  ← Add HTTP server connector
    }

    getDefaultHost().attachDefault(new MailServerApplication()); ← Attach application to default host
}

}
```

As you can see, starting the component is as easy as invoking its `start()` method, which in turn arranges for the HTTP server connector added in the constructor to listen on port 8111, ready to answer to inbound calls. In the next subsection, we study in detail how to configure a Restlet component, its connectors, virtual hosts, services, and other features—but before that you need to ensure that the component is actually working.

Listing 3.2 shows a simple mail client that sets up a base service client resource and obtains child client proxies for each remote resource from that service that it needs to interact with. You create client proxies for child resources using the `getChild(String relativeUri, Class<T>)` method, as demonstrated in listing 2.13. For space reasons, we've omitted the class declaration.

Listing 3.2 Simple mail client interacting with component resources

```
System.out.println("\n1) Set up the service client resource\n");
Client client = new Client(new Context(), Protocol.HTTP);
ClientResource service = new ClientResource("http://localhost:8111");
service.setNext(client);

System.out.println("\n2) Display the root resource\n");
RootResource mailRoot = service.getChild("/", RootResource.class);
System.out.println(mailRoot.represent());

System.out.println("\n3) Display the initial list of accounts\n");
AccountsResource mailAccounts = service.getChild("/accounts/",
    AccountsResource.class);
```

```

String list = mailAccounts.represent();
System.out.println(list == null ? "<empty>\n" : list);

System.out.println("4) Adds new accounts\n");
mailAccounts.add("Homer Simpson");
mailAccounts.add("Marjorie Simpson");
mailAccounts.add("Bart Simpson");
System.out.println("Three accounts added !");

System.out.println("\n5) Display the updated list of accounts\n");
System.out.println(mailAccounts.represent());

System.out.println("6) Display the second account\n");
AccountResource mailAccount = service.getChild(
    "/accounts/1", AccountResource.class);
System.out.println(mailAccount.represent());

System.out.println(
    "\n7) Update the individual account and display it again\n");
mailAccount.store("Homer Jay Simpson");
System.out.println(mailAccount.represent());

System.out.println(
    "\n8) Delete the first account and display the list again\n");
mailAccount = service.getChild("/accounts/0", AccountResource.class);
mailAccount.remove();
System.out.println(mailAccounts.represent());

```

Note how each call to a remote resource looks as though that resource were locally present. You might also wonder what happens when an error occurs or when you need access to underlying HTTP details (for example, for preemptive authentication). To give you that access, dynamic client proxies created by ClientResource also implement the org.restlet.resource.ClientProxy interface. This interface declares a single getClientResource() method that points to the wrapped ClientResource instance and provides the richness of all the Restlet API to inspect and modify the request and response objects.

After making sure that the server-side component is started, you can launch the MailClient class and observe the console output, which looks like this:

- 1) Set up the service client resource

```
org.restlet.engine.http.connector.HttpClientHelper start
INFO: Starting the default HTTP client
```

- 2) Display the root resource

```
Welcome to the RESTful Mail Server application !
```

- 3) Display the initial list of accounts

```
<empty>
```

- 4) Adds new accounts

```
Three accounts added !
```

- 5) Display the updated list of accounts

```

Homer Simpson
Marjorie Simpson
Bart Simpson
6) Display the second account
Marjorie Simpson
7) Update the individual account and display it again
Marge Simpson
8) Delete the first account and display the list again
Marge Simpson
Bart Simpson

```

After this basic test run of the Component class, embedding the example mail system started in chapter 2, let's continue with the description of connectors and how they can be managed by a Restlet component.

3.3.2 Adding server and client connectors

When your Restlet application needs to communicate with other applications—either locally or remotely, acting as a server or as a client—it needs to use a specific medium, like a TCP/IP socket, and exchange data following a specific protocol, such as HTTP, FTP, or SMTP.

In REST, connectors are in charge of those communication needs. The Restlet API, as usual, exactly follows this terminology by having an abstract `org.restlet.Connector` class and two concrete subclasses—`org.restlet.Client` and `org.restlet.Server`—each of which can be used with a variety of protocols thanks to the uniform interface supported by all Restlet subclasses.

The class diagram in figure 3.4 illustrates the relationships between those three classes along with their main properties. (To be concise, we don't mention all the

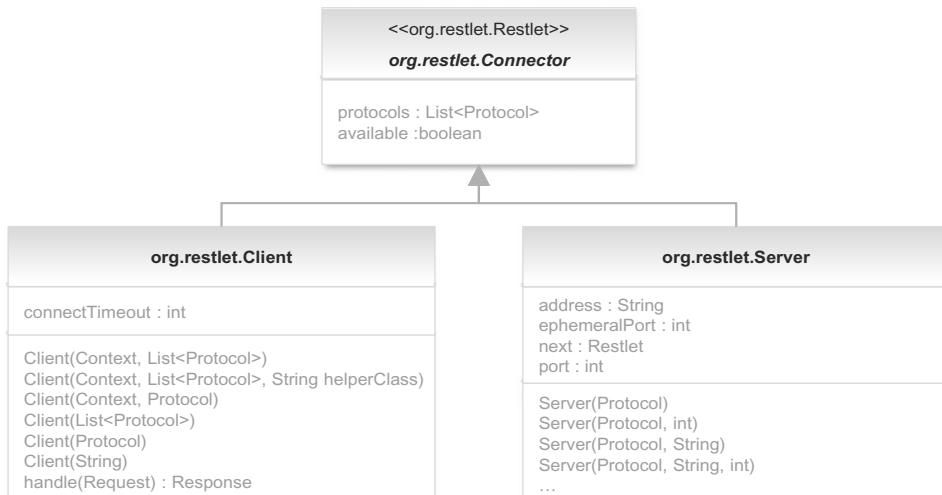


Figure 3.4 Class diagram of the Connector superclass and the Client and Server child classes

constructors of the Server class.) Refer to the Javadocs for authoritative information about the Restlet API. Table 3.1 presents each property of the classes.

Table 3.1 Properties of the Connector, Client, and Server classes

Name	Description
protocols	List of the protocols expected to be supported at the same time by the same connector instance. On the server side, it's generally one protocol like HTTP or HTTPS but not both. On the client side, it can be both HTTP and HTTPS at the same time.
available	Indicates whether the connector was able to look up an implementation helper matching the specified protocol(s).
connectTimeout	The delay in milliseconds that a client connector will wait while attempting to establish a network connection with a remote host.
address	The optional listening IP address in textual format. It's generally specified at construction time, and changes aren't taken into account until the connector is restarted.
ephemeralPort	Actual ephemeral port used by a server connector when the listening port is set to 0. This allows you to listen on any available port without knowing it in advance, typically for unit-testing purposes when you don't know the available fixed ports.
next	Reference to the Restlet that will handle inbound calls received by a server connector. When a <code>ServerResource</code> subclass is specified, an implicit <code>Finder</code> is used as a resource factory.
port	The listening socket port number. It's generally specified at construction time, and changes aren't taken into account until the server connector is restarted.

The Restlet Engine comes with built-in connectors, but additional ones are also available via the set of Restlet Extensions. You can also develop your own connectors and plug them dynamically in the Restlet Engine. If you want to do so, contact the Restlet community for guidance and additional documentation.

Let's review the available connectors and their main characteristics, starting with the default ones provided in the Restlet core (distributed as a single `org.restlet.jar` file in version 2.1). All these built-in connectors listed in table 3.2 are usable in both development and light deployment scenarios.

Those connectors are key elements of the framework, but in the end you'll rarely directly invoke them. Instead, you'll use higher-level classes such as `ClientResource` and `ServerResource` as discussed previously, or the `Directory` class to serve static files as you'll see in section 8.1. We make concrete use of the RIAP pseudoprotocol in section 8.5 when discussing modularization of large applications.

Let's now introduce the extension connectors that are provided in extension JAR files, such as `org.restlet.ext.jetty.jar`. Additional library dependencies are provided in the Restlet distribution for Java SE and described in a `readme.txt` file of the `lib` top directory. Table 3.3 lists the connectors available.

Table 3.2 Characteristics of built-in connectors

Protocols	Description
HTTP/HTTPS client and server	<p>Complete client and server connectors supporting all HTTP features, including</p> <ul style="list-style-type: none"> ▪ Persistent connections (with limit settable per host and in total) ▪ Pipelining connections (for reduced network latency) ▪ Chunked encoding (for entities with unknown length) ▪ Provisional responses (for informational status) ▪ Configurable worker thread pool ▪ Raw tracing of requests and responses to system console ▪ Asynchronous callbacks (added to Restlet API 2.0)
FILE client	<p>Local client connector to manipulate (list, read, write, and delete) files using the <i>file</i> URI scheme. For example, you can send a PUT request to the file:///C/www/index.html resource with a request entity that holds the new content of the (existing or new) file identified by the URI. Note that for best portability across OS, you should try using the CLAP client or the WAR client under the Java EE environment.</p>
ZIP/JAR client	<p>Local client connectors to read resources inside local ZIP or JAR files using the <i>zip</i> or <i>jar</i> URI scheme. For example, you can send a GET request to the jar:file:///C/foo/foo.jar!/org/foo/Bar.class URI.</p>
CLAP client	<p>JVM connector to get the representations of resources accessible via class loaders and the <i>clap</i> URI scheme. Class Loader Access Protocol (CLAP) supports these URI authorities:</p> <ul style="list-style-type: none"> ▪ <i>class</i>—For resources accessible via the classloader of the connector class. If your request contains an “<i>org.restlet.clap.classLoader</i>” attribute, you can provide a custom classloader. ▪ <i>system</i>—For resources accessible via the system’s classloader. See the <code>System.getSystemClassLoader()</code> method for details. ▪ <i>thread</i>—For resources accessible via the current thread’s classloader. See the <code>Thread.getContextClassLoader()</code> for details on this classloader. <p>For example, you can send a GET request to the <i>clap://class/org/restlet/Restlet.class</i> resource to retrieve the content of the <i>org.restlet.Restlet.class</i> file. It can also retrieve properties files or be used to serve static websites bundled as JAR files.</p>
RIAP client and server	<p>JVM connector to manipulate (with any method) the representation of resources accessible via the local Restlet Framework and the <i>riap</i> URI scheme. RIAP supports these URI authorities:</p> <ul style="list-style-type: none"> ▪ <i>application</i>—For resources accessible relative to the current Restlet application if available ▪ <i>component</i>—For resources accessible relative to the current Restlet component if available ▪ <i>host</i>—For resources accessible relative to the current Restlet virtual host if available <p>For example, you can send a PUT request to update the <i>riap://application/foo/bar</i> resource to update the resource under the relative URI <i>foo/bar</i> in the current application. This pseudoprotocol is implicitly available through the <code>Context.clientDispatcher</code> mechanism.</p>

In addition to the main features described in table 3.3, you can obtain more information in the Javadocs, including a list of parameters that can be adjusted for each connector. The following listing shows how to configure such a parameter by adjusting the `MailServerComponent` class from listing 3.1.

Table 3.3 Characteristics of extension connectors

Extension	Protocols	Description
HTTP Client	HTTP/ HTTPS client	<p>Complete and production-ready connector based on the latest version of the popular Apache HTTP Client library. All HTTP features are supported, including:</p> <ul style="list-style-type: none"> ■ Persistent connections (with limit settable per host and in total) ■ Chunked encoding (for entities with unknown length) ■ HTTP proxy configuration ■ Timeout configuration
JavaMail	POP/POPS, SMTP/ SMTPS/ SMTP StartTLS client	<p>Complete and production-ready client connector to post and retrieve emails from a remote mail server based on the JavaMail API and reference implementation. It also supports common authentication mechanisms.</p> <p>The originality of this connector is that it maps the HTTP semantics of the GET and POST methods to the JavaMail API, using XML documents to represent emails and mail boxes. There's no need to learn JavaMail to use it.</p> <p>Dynamic XML emails can easily be created using a template engine such as FreeMarker or Velocity.</p>
JDBC	JDBC client	<p>Complete and production-ready client connector to post SQL statements to a JDBC data source, typically a relational database based on the JDBC API. It supports regular authentication and connection pooling via the Apache Commons Pool library.</p> <p>The originality of this connector is that it maps the HTTP semantics of the POST method to the JDBC API, using XML documents to represent SQL statements and result sets. There's no need to learn JDBC to use it.</p> <p>Dynamic XML statements can easily be created using a template engine such as FreeMarker or Velocity.</p>
Jetty	AJP/HTTP/ HTTPS server	<p>Complete and production-ready connector based on the popular Jetty HTTP server. Most HTTP features are supported:</p> <ul style="list-style-type: none"> ■ Persistent connections (with limit settable per host and in total) ■ Chunked encoding (for entities with unknown length) ■ Configurable worker thread pool ■ Graceful shutdown ■ Timeouts configuration ■ Buffer size configuration ■ Choice of BIO or NIO implementation <p>Besides the usual HTTP and HTTPS protocols, this connector supports the AJP protocol, which is sometimes used to tunnel HTTP calls between a front-end Apache HTTP daemon and a back end HTTP server as an alternative to reverse HTTP proxies.</p>
Lucene	SOLR client	Client connector to an embedded Apache Solr indexing and retrieval engine.
Net	HTTP/ HTTPS and FTP clients	<p>Complete and production-ready connector based on the <code>java.net.HttpURLConnection</code> class from the JDK. Main HTTP features are supported, including:</p> <ul style="list-style-type: none"> ■ Persistent connections ■ Chunked encoding (for entities with unknown length) ■ HTTP proxy configuration (JVM settings) ■ Timeout configuration

Table 3.3 Characteristics of extension connectors (continued)

Extension	Protocols	Description
Simple	HTTP/ HTTPS server	The main interest of this connector is that it requires no additional dependency and is the only HTTP client supported in the Google App Engine edition.
SIP	SIP client and server	Alternative server connector ready for production based on the lightweight Simple HTTP framework, with features including: <ul style="list-style-type: none"> ■ Chunked encoding (for entities with unknown length) ■ Configurable worker thread pool ■ Graceful shutdown Standalone client and server connectors built on top of the internal HTTP/NIO connector, enabling the construction of convergent web and VoIP applications. SIP is the base protocol of VoIP, inspired by HTTP. Note that the extension doesn't provide a SIP state machine but only a Java API that maps most SIP methods, statuses, and headers as well as SipClientResource and SipServerResource classes to process SIP calls.

Removed connectors

Note that some experimental connectors based on Grizzly and Netty NIO frameworks were introduced in version 2.0, then removed in 2.1 in favor of new internal HTTP connectors that take full advantage of asynchronous NIO processing.

Listing 3.3 Adding HTTP tracing to the internal server connector

```
...
// Adds an HTTP server connector
Server server = getServers().add(Protocol.HTTP, 8111);
server.getContext().getParameters().set("tracing", "true");
...
```

To be taken into account, the "tracing" parameter must be set before the connector is started. When enabled it asks the default HTTP server to trace in the Java console the exact content of all the requests and responses exchanged with clients. This can be handy when troubleshooting.

The next subsection explores virtual hosting.

3.3.3 Setting up virtual hosting

In chapter 2, we discuss the Restlet routing system and in particular the Router class. The role of a router is to dispatch inbound calls to the best of the attached routes based on some criteria, typically the target URI of the request, or more precisely the relative part of the URI that hasn't been routed.

For example, inside the MailServerApplication class updated in listing 3.13, we attached resources to the root router using relative URI paths, assuming that the base

of the URI had already been consumed by the default host in `MailServerComponent` in listing 3.1. In this section we examine how the base of the URI—for example, `http://localhost:8111`—is effectively matched by a Restlet component and the exact behavior of a virtual host.

Virtual hosting in Restlet allows you to bind the same IP address to several internet domain names and still retain the ability to serve each domain name with a separate Restlet application. You can also listen at the same time to several IP addresses, handled by the same application if necessary—for example, for network load-balancing or fail-over purposes.

To get started, look at table 3.4, which lists the properties of `org.restlet.routing.VirtualHost`, which extends the `Router` class.

Table 3.4 Properties of the `VirtualHost` classes

Property	Description
<code>hostDomain</code>	Pattern string that is matched against the domain name of the host URI reference of the incoming call, extracted from the <code>Request.hostRef.hostDomain</code> property.
<code>hostPort</code>	Pattern string that is matched against the port number of the host URI reference of the incoming call, extracted from the <code>Request.hostRef.hostPort</code> property. If not present in the URI, the pattern is matched against the default port number of the scheme, such as 80 for HTTP or 443 for HTTPS.
<code>hostScheme</code>	Pattern string that is matched against the scheme name of the host URI reference of the incoming call, extracted from the <code>Request.hostRef.scheme</code> property.
<code>resourceDomain</code>	Pattern string that is matched against the domain name of the target resource URI reference of the incoming call, extracted from the <code>Request.resourceRef.hostDomain</code> property.
<code>resourcePort</code>	Pattern string that is matched against the port number of the target resource URI reference of the incoming call, extracted from the <code>Request.resourceRef.hostPort</code> property. If not present in the URI, the pattern is matched against the default port number of the scheme, such as 80 for HTTP or 443 for HTTPS.
<code>resourceScheme</code>	Pattern string that is matched against the scheme name of the target resource URI reference of the incoming call, extracted from the <code>Request.resourceRef.scheme</code> property.
<code>serverAddress</code>	Pattern string that is matched against the server IP address that received the incoming call, extracted from the <code>Response.serverInfo.address</code> property.
<code>serverPort</code>	Pattern string that is matched against the server IP port number that received the incoming call, extracted from the <code>Response.serverInfo.port</code> property. If not specified, the pattern is matched against the default port number of the protocol used to receive the call, defined by <code>Request.protocol.defaultPort</code> .

Each property is a string containing a pattern to be matched. The expression language used for the pattern is defined by the `java.util.regex.Pattern` class. For example, all the properties have the same default value `.*`, which matches any sequence of characters. The value `a*` would only match a sequence of zero or more `a` characters. Regular expressions can be powerful but complex, so we recommend reading the Javadocs of the `Pattern` class and related documentation for help.

The last piece that's missing is the selection of the best virtual host for an inbound call by attempting to match the previously mentioned properties. This is the role of the engine logic inside each Restlet component, which uses a special router adding a route for each virtual host declared via the `Component.hosts` property as well as the `Component.defaultHost` property. In case no virtual host matches, a 404 (Not Found) status is returned.

Note that the Component's default host is initialized to match any request, so be careful when going into production and precisely check which Restlet applications are attached to it. You can set it to null or make sure that its patterns are more restrictive, like a regular virtual host bound to a specific domain name or IP address. Finally, remember that a `VirtualHost` instance is still a kind of `Router` instance, so you can attach applications to it, or any kind of Restlet instance if necessary, using several attachment subpaths for different applications attached to the same host.

Applying what you've learned, let's plan an (imaginary) production deployment of the RESTful mail system that makes `MailServerComponent` and the contained `MailServerApplication` available on the Web at the following URIs: `www.rmep.com`, `www.rmep.net`, and `www.rmep.org`. Let's also imagine that the dedicated server machine has two network cards for network fail-over reasons, each linked to a different internet provider. (Another reason for having multiple IP addresses could be for network load-balancing or to map different domain names to separate IP addresses.) Figure 3.5 summarizes the virtual host configuration you want to obtain.

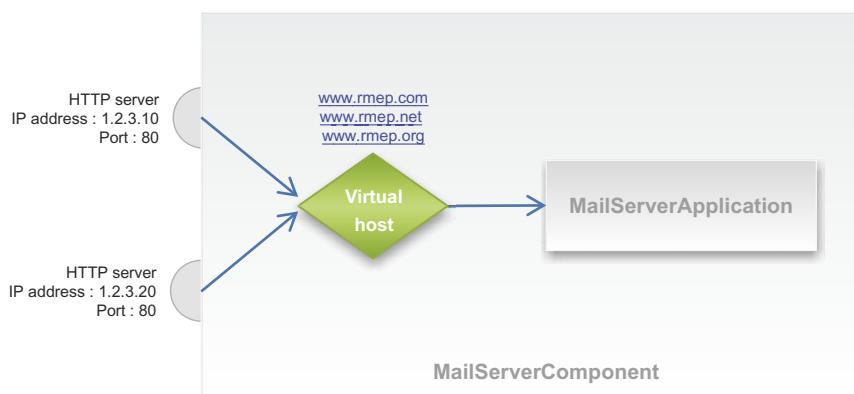


Figure 3.5 Virtual host listening on two IP addresses and serving several domain names

You want to update your component to match the desired configuration. Because there's only one virtual host, you'll reconfigure the default virtual host, which by default matches all requests. The following listing sets more precise patterns to match the host domain name of the request and the HTTP server IP address and port number.

Listing 3.4 Configuring the virtual hosting

**Configure
default
virtual host**

```
VirtualHost host = getDefaultHost();
host.setHostDomain("www\\.rmep\\.com|www\\.rmep\\.net|www\\.rmep\\.org");
host.setServerAddress("1\\.2\\.3\\.10|1\\.2\\.3\\.20");
host.setServerPort("80");
host.attachDefault(new MailServerApplication());
```

**Attach application
to default host**

If you restart the component and attempt to GET the `http://localhost:8111/accounts/` URI in your browser, the following error message is displayed: "No virtual host could handle the request." It might appear that you can't test this configuration unless all the domain names are bought and the network is configured with the matching IP addresses. Now it's time to demonstrate how to use the Restlet Framework for testing purposes by creating test requests and responses and locally invoking any Restlets (components, applications) without going through the connector and network layers.

Listing 3.5 presents a simple JUnit test case that builds a request/response as it would be received by the server connector. It directly invokes a local instance of the `MailServerComponent` class. To make sure this test fails if the domain name doesn't match, try running it again after changing the request domain name to `rmep1.org`.

Listing 3.5 Unit test for virtual host

```
import junit.framework.TestCase;
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.data.Method;
import org.restlet.example.book.restlet.ch03.sec3.server.
MailServerComponent;
public class MailClientTestCase extends TestCase {
    public void testVirtualHost() {
        MailServerComponent component = new MailServerComponent();
```

**Instantiate
Restlet
component**

Request request = new Request();

Prepare test HTTP call

request.setMethod(Method.GET);

request.setResourceRef("http://www.rmep.org/accounts/");

request.setHostRef("http://www.rmep.org");

Response response = new Response(request);

response.getServerInfo().setAddress("1.2.3.10");

response.getServerInfo().setPort(80);

component.handle(request, response);

assertTrue(response.getStatus().isSuccess());

**Test if response
was successful**

}

We're now nearly done with our exploration of standalone deployment of Restlet applications; we need to cover the available services in Component and declarative XML configuration.

3.3.4 Configuring common services

In section 2.3.4, we explain how to configure common services for Application. For the Component class, the same service mechanism applies; services extend the Service class and can be added to the Component.services property for registration. Two default services are available, listed in table 3.5, enabled and preregistered for you.

Table 3.5 Restlet applications can use several built-in component services.

Name	Description
LogService	Global access log for all server connectors and applications contained in the component. By default, it produces log entries in a format similar to IIS and Apache HTTP servers.
StatusService	Handles error statuses. If an exception is thrown within your application or Restlet code, it's intercepted by this service in order to customize the response status code and entity to ensure a consistent look-and-feel.

The status service is complementary to the one defined in each application but can be configured to overwrite the status pages to make sure that they have a consistent look-and-feel across all applications contained.

Let's look more closely at the log service and configure it to maintain a rotating access log file. The following listing updates the MailServerComponent to load logging properties via the classpath.

Listing 3.6 Configuring access logging

```
getClients().add(Protocol.CLAP);
[...]
getLogService().setLoggerName("MailServer.AccessLog");
getLogService().setLogPropertiesRef(
    "clap://system/org/restlet/example/book/restlet/
    ch03/sec3/server/log.properties");
```

In this example it's interesting to note how you use the Restlet client connector for the CLAP pseudoprotocol to easily retrieve the log.properties file from the classpath so it can be stored in the same directory as the MailServerComponent.java source file. Have a look at the self-documented configuration file to understand how you set up log handlers for the console, the rotating access log file, and the rotating debug file. You can easily adjust the logging level in this configuration to capture more or fewer log events in the console or in the log files.

Alternatively, you can indicate the location of the login properties with a JVM parameter:

```
-Djava.util.logging.config.file="/foo/bar/log.properties"
```

You can use advanced logging frameworks via the `org.restlet.ext.slf4j` extension. Switch Restlet logging from built-in Java logging to SLF4J by setting the `org.restlet.engine.loggerFacadeClass` system property to the value `org.restlet.ext.slf4j.Slf4jLoggerFacade`—or achieve the same effect programmatically with this code:

```
Engine.getInstance().setLoggerFacade(new Slf4jLoggerFacade());
```

SLF4J documentation describes how to configure the façade to use the desired framework (LogBack, log4j, and others). Note that currently the line numbers in log messages aren't for the calling code but are from the logger façade class itself. Apart from this drawback, the Restlet bridge from `java.util.logging` to SLF4J is more efficient than the one provided by SLF4J because no additional log record objects are created; the façade invokes the SLF4J APIs directly.

In this section you saw that the Restlet API encompasses all aspects of REST and lets you build coarse-grained components and connectors using simple Java code. We described the available connectors and virtual hosts and saw how to configure them within Restlet components. You'll now see a completely different way to configure Restlet components—using XML.

3.4 Declarative configuration in XML

Imagine that your system administrator has no Java skills but still needs to adjust the virtual hosts, port numbers, and IP addresses for your deployed applications. In such cases, you can use XML to configure your standalone Restlet components.

In this section we explain how to configure a component using a declarative XML approach with results equivalent to those obtained using plain Java code. You'll see an example using the simple XML format supported by the `Component` class and then learn about a more complete and powerful approach based on the Spring Framework.

3.4.1 XML configuration with Component

The first way to use XML is built into the `Component` class itself, allowing you to use a simple XML document to configure your connectors, virtual hosts, and other properties that we used in the `MailServerComponent` class. The following listing replaces the `MailServerComponent` class with a single XML document.

Listing 3.7 Declarative XML configuration with the Component class

```
<?xml version="1.0"?>
<component xmlns="http://www.restlet.org/schemas/2.0/Component" <-- Set basic properties
            name="RESTful Mail Server component"
            description="Example for 'Restlet in Action' book"
            owner="Restlet SAS"
            author="The Restlet Team">

            <client protocol="CLAP" /> <-- Declare connectors
            <server protocol="HTTP">
```

```

<parameter name="tracing" value="true" />
</server>

<!--
<defaultHost
    hostDomain="www\\.rmepl\\.com|www\\.rmepl\\.net|www\\.rmepl\\.org"
    serverAddress="1\\.2\\.3\\.10|1\\.2\\.3\\.20" serverPort ="80">
    <attachDefault
        targetClass="org.restlet.example.book.restlet
                    .ch03.sec3.server.MailServerApplication" />
</defaultHost>
-->

<defaultHost
    <attachDefault
        targetClass="org.restlet.example.book.restlet
                    .ch03.sec3.server.MailServerApplication" />
</defaultHost>
<logService loggerName="MailServer.AccessLog"
            logPropertiesRef="clap://system/org/restlet/example/
                            book/restlet/ch03/sec3/server/log.properties" />
</component>

```

The annotations are represented by callout boxes with arrows pointing to specific parts of the XML code:

- A callout box labeled "Declare virtual hosts" points to the first `<defaultHost>` block, which defines multiple host domains and their corresponding server addresses.
- A callout box labeled "Configure log service" points to the `<logService>` element, which specifies the logger name and the reference to the log properties file.

This configuration is similar to the equivalent Java code but can be more easily changed, without recompilation or Java knowledge. To run this component, you invoke the `Component.main()` static method, passing as a parameter a URI that refers to the XML configuration. Any available protocol or pseudoprotocol can be used, such as CLAP for the classpath, FILE for a local file, and even HTTP. In this case, we stored the configuration in a `component-simple.xml` file stored in the source package, resulting in this URI: `clap://system/org/restlet/example/book/restlet/ch03/sec3/server/component-simple.xml`.

3.4.2 XML configuration with Spring Framework

Another popular way to declaratively configure a Restlet component is with the Spring Framework, using its special XML syntax for dependency injection even though it's

Spring Framework

Spring is a layered application framework and lightweight container with foundations in the book *Expert One-on-One J2EE Design and Development* by Rod Johnson (Peer Information, 2002). (The Spring project itself started in 2003.) The lightweight container and an aspect-oriented programming (AOP) system are the main building blocks of Spring. Spring also comes with a common abstraction layer for transaction management, integration with various persistence solutions (plain JDBC, Hibernate, JPA), and Java enterprise technologies (JMS, JMX). Spring even provides its own Model View Controller (MVC) Web framework called Spring MVC. The Spring framework isn't an "all-or-nothing" solution: you can choose the modules according to need, the lightweight container being the glue for the application and the Spring classes.

also possible to use annotations. This approach encourages loose coupling between entities and improves application maintainability.

To simplify using Restlet with the Spring Framework, a special extension is provided in the `org.restlet.ext.spring` module. It contains classes to facilitate configuration; for example, you can use `SpringComponent` instead of its parent `Component` class.

The following listing demonstrates configuring a component equivalent to the `MailServerComponent` using Spring XML.

Listing 3.8 Declarative XML configuration with the Spring Framework

```
<?xml version="1.0"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-3.0.xsd">
    <!-- Root element with namespaces declarations -->
    <!-- Component properties set by value/reference -->
    <bean id="component"
          class="org.restlet.ext.spring.SpringComponent">
        <property name="name" value="RESTful Mail Server component" />
        <property name="description"
                  value="Example for 'Restlet in Action' book" />
        <property name="owner" value="Restlet SAS" />
        <property name="author" value="The Restlet Team" />
        <property name="client" value="clap" />
        <property name="server" ref="server" />
        <property name="defaultHost" ref="defaultHost" />
    </bean>
    <bean id="component.context"
          class="org.springframework.beans.factory
                  .config.PropertyPathFactoryBean" />
    <!-- Extraction of component's context for reuse -->
    <bean id="server" class="org.restlet.ext.spring.SpringServer">
        <constructor-arg value="http" />
        <constructor-arg value="8111" />
        <property name="parameters">
            <props>
                <prop key="tracing">true</prop>
            </props>
        </property>
    </bean>
    <!-- HTTP server with tracing enabled -->
    <bean id="defaultHost" class="org.restlet.ext.spring.SpringHost">
        <constructor-arg ref="component" />
        <!-- Virtual host with attached mail server application -->
        <!--
            <property name="hostDomain"
            value="www\\.rmep\\.com|www\\.rmep\\.net|www\\.rmep\\.org" />
            <property
                name="serverAddress" value="1\\.2\\.3\\.10|1\\.2\\.3\\.20"
        />
        <property
    
```

```

        name="serverPort" value="80" />
-->

<property name="defaultAttachment" ref="mailServerApplication" />
</bean>

```

Each instance managed by Spring is defined using a bean XML element that can specify both the class of the instance (the class attribute) and its properties (the properties XML element). The property XML element can contain either a value or a reference to another bean managed by Spring. The given configuration declares the equivalent of the existing MailServerComponent.

With the Spring Framework, you can go further and configure other parts of the Restlet API, such as an application and its contained resources. The following listing contains the continuation of listing 3.8.

Listing 3.9 Declarative XML configuration with the Spring Framework (continued)

```

<bean id="componentChildContext" class="org.restlet.Context" > ↗ Create child
    <lookup-method name="createChildContext" context
        bean="component.context" />
</bean>

<bean id="mailServerApplication" class="org.restlet.Application" > ↗ Application
    <constructor-arg ref="componentChildContext" />

    <property name="name" value="RESTful Mail Server application"/>
    <property name="description"
        value="Example application for 'Restlet in Action' book" />
    <property name="owner" value="Restlet SAS" />
    <property name="author" value="The Restlet Team" />
    <property name="inboundRoot" > ↗ Routes declaration
        <bean class="org.restlet.ext.spring.SpringRouter" with SpringRouter
            <constructor-arg ref="mailServerApplication" />
            <property name="attachments">
                <map>
                    <entry key="/" > ↗
                        value="org.restlet.example.book.restlet.
                            ↗ ch03.sec3.server.RootServerResource" />
                    <entry key="/accounts/" > ↗
                        value="org.restlet.example.book.restlet.
                            ↗ ch03.sec3.server.AccountsServerResource" />
                    <entry key="/accounts/{accountId}" > ↗
                        value="org.restlet.example.book.restlet.
                            ↗ ch03.sec3.server.AccountServerResource" />
                </map>
            </property>
        </bean>
    </property>
</bean>
</beans>

```

This part of the configuration is a bit verbose because there's a lot of information to provide on the application. First you set the child context and general application information. Then you define a SpringRouter instance and its attachments property

to indicate how to access your resources. This router is injected within the application using its `inboundRoot` property. This corresponds to what you specify in the `createInboundRoot` method without Spring.

Note that you have to manually create a child context because the component's context can't be directly passed to isolate all contained applications, for security reasons. With the Java API, this child context is automatically created by the attachment methods, but with Spring it has to be done explicitly at instantiation time.

As a result, you can configure both your component and your application in XML with Spring, leaving only the server resources to code. This can be useful in situations where you want to provide different sets of resources for an application based on some deployment aspects, or different sets of applications for a component.

This is an overview of the features offered by the Spring extension for Restlet. Refer to the Javadocs and the online user guide for details. Let's now make sure the Spring XML configuration works by running the small bootstrap code in the following listing.

Listing 3.10 Running the declarative XML configuration with the Spring Framework

```
public class MailServerSpring {
    public static void main(String[] args) throws Exception {
        // Load the Spring container
        ClassPathResource resource = new ClassPathResource(
            "org/restlet/example/book/restlet/ch03"
            ➔ /sec3/server/component-spring.xml");
        BeanFactory factory = new XmlBeanFactory(resource);

        // Start the Restlet component
        Component component = factory.getBean("component", Component.class);
        component.start();
    }
}
```

The `BeanFactory` entity representing the Spring container can be loaded using several strategies. Listing 3.10 loads from the classpath based on the `ClassPathResource` class. The latter looks for the file in the classpath interpreting the path relative to the classpath root. Because you use an XML file for metadata, the `XmlBeanFactory` class is used. Now that the container is loaded, you can access configured beans by identifier using the `getBean` method. The example gets the Restlet component instance and starts it with its `start` method.

In this section you saw how to configure Restlet components with XML instead of Java code, first using the built-in XML configuration capabilities of the `Component` class and then using the more advanced XML configuration based on the Spring Framework. More approaches have been successfully explored by Restlet community members, such as the use of other dependency injection frameworks like Google Guice and dynamic languages such as Groovy and Scala.

Spring XML configurations can be verbose due to XML itself. To address this issue, Spring lets you use dedicated XML schemas for particular use cases. The most well-known

are the ones for AOP and transactions. With them, there's no need to know about the underlying beans; the schema hides this complexity. The Restlet extension for Spring doesn't provide such a schema yet, but work is in progress (see issue #508 on GitHub for more details). Once available, the resulting XML configuration will look like the following listing.

Listing 3.11 Configuration using Spring namespace for Restlet

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:restlet="http://www.restlet.org/schema/spring-restlet"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.restlet.org/schema/spring-restlet
                           http://www.restlet.org/schema/restlet/spring-restlet.xsd">

    <restlet:application id="application">
        <restlet:attachments>
            <restlet:attachment route="/companies/{id}"
                                resource="companyResource"/>
        </restlet:attachments>
    </restlet:application>
</beans>
```

Next we introduce another common way to deploy Restlet applications: reusing existing investments in Java EE application servers.

3.5 Deployment in an existing Java EE server

As you saw in section 3.3, the Restlet Framework can be used as a standalone framework on top of Java SE providing a lightweight web development platform. But it's often necessary to deploy to Java EE application servers in production environments or to integrate with existing Servlet applications to add RESTful support without starting again from scratch.

In this section we present the Java EE edition of the Restlet Framework, especially its Servlet extension, and we discuss the integration modes supported. We illustrate those integration options by deploying our example RESTful email system into Apache Tomcat, the most widely used Servlet engine.

3.5.1 The Servlet extension

The Java EE edition of the Restlet Framework contains the exact same API and engine as the Java SE edition but comes with a different set of extensions. For example, the standalone HTTP/HTTPS server connectors such as Jetty and Simple aren't provided but the new Servlet extension is.

This extension, available as an org.restlet.ext.servlet.jar file, provides a bridge between the Servlet API and the Restlet API. This bridge was possible because the Restlet API is designed at a higher abstraction level and has a much broader feature scope.

NOTE Moving forward in this section, you'll need to download the distribution for Java EE, including the addition Servlet extension, in order to run the examples.

Now let's look at the main bridging scenarios supported by the extension.

3.5.2 **Servlet engine as a connector for a Restlet component**

The first scenario makes the most use of the Restlet Framework. You keep using a Restlet component as a container of Restlet applications. The underlying Servlet engine is only used as a server connector to support HTTP and HTTPS protocols, as illustrated in figure 3.6.

This scenario can ensure broad portability of a Restlet project from a standalone Java SE deployment to a Java EE deployment but requires extra care in the way the Servlet engine is configured. Indeed, although the Restlet API gives you full control over its routing system, the Servlet API has adopted a more declarative way, typically based on a /WEB-INF/web.xml file.

As a result, the initial URI-based routing of incoming calls will be routed first through the Servlet engine before reaching the Restlet component. Because the Restlet component is designed to fully handle routing, including virtual hosting, it starts the routing again from the beginning of the URI. Therefore you must ensure that its routing configuration is consistent with the Servlet's.

Let's put that advice in practice by deploying the `MailServerComponent` inside Apache Tomcat 6.0, a popular Servlet engine. First, you trim the class down a little in the following listing because you don't need to declare the HTTP server connector anymore and you can rely on Tomcat to write the access log.

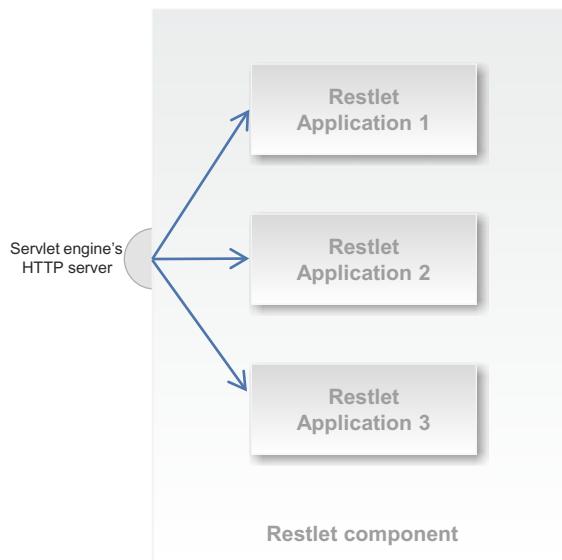


Figure 3.6 Using the Servlet engine as an HTTP server connector for a Restlet component

Listing 3.12 Trimmed down MailServerComponent for Servlet deployment

```

public class MailServerComponent extends Component {
    public MailServerComponent() throws Exception {
        // Set basic properties
        setName("RESTful Mail Server component");
        setDescription("Example for 'Restlet in Action' book");
        setOwner("Restlet SAS");
        setAuthor("The Restlet Team");

        // Attach the application to the default virtual host
        getDefaultHost().attachDefault(new MailServerApplication());
    }
}

```

As you can see, you don't need to declare the CLAP client connector because you no longer load the log properties file. But it's common in the Servlet world to store configuration files inside the WAR structure, typically in the /WEB-INF/ directory. Those files are easily accessible from the Restlet world thanks to a WAR client connector that's automatically added to your component. As an exception, you don't have to manually declare it. To use it, use a ClientResource or the Context.dispatcher property and issue GET requests on URI references like war:///WEB-INF/myConfig.xml.

Let's now configure the Servlet engine to route HTTP calls to your component. For this, you need to edit the usual /WEB-INF/web.xml file, using the ServerServlet adapter class from the Servlet extension, as illustrated in the following listing.

Listing 3.13 Configuring the Servlet's web.xml file

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns="http://java.sun.com/xml/ns/javaee"
          xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
          id="WebApp_ID" version="2.5">

    <display-name>
        Servlet engine as a connector for a Restlet component
    </display-name>
    <servlet>
        <servlet-name>MailServerComponent</servlet-name>
        <servlet-class>
            org.restlet.ext.servlet.ServerServlet
        </servlet-class>
        <init-param>
            <param-name>org.restlet.component</param-name>
            <param-value>
                org.restlet.example.book.restlet.ch03.sec3.server.
                MailServerComponent
            </param-value>
        </init-param>
    </servlet>

```

```

        </param-value>
    </init-param>
</servlet>

<servlet-mapping>
    <servlet-name>MailServerComponent</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
</web-app>

```

Map all URIs to
bridge Servlet

The next step is to copy two JARs from the Restlet distribution for Java EE into the /WEB-INF/lib/ directory:

- org.restlet.jar
- org.restlet.ext.servlet.jar

Finally you need to package the Servlet project as a WAR and deploy it with no base URI path. (Ordinarily the WAR name is used as the base URI path, but this wouldn't match the URI routing declared inside the MailServerComponent where / maps to the RootServerResource.) You can then launch Tomcat and test your Restlet component by entering the following URI in your browser: <http://localhost:8080/>. The following message should be displayed: "Welcome to the RESTful Mail Server application!" Note that the default port number of Tomcat is 8080—a change from the 8111 value you've used until now.

3.5.3 Servlet engine as a container of Restlet applications

This second scenario is useful when you have existing Servlet applications deployed and want to deploy Restlet applications as well under separate URI paths. In this case the idea is to consider the Servlet engine as the component containing the applications and providing the server HTTP connector.

Figure 3.7 illustrates a Java EE deployment with two Servlet applications and one Restlet application coexisting. Each application is supposed to be deployed as a separate WAR file, contrary to the previous scenario.

Let's now configure the Servlet engine to route HTTP calls to the example MailServerApplication. For this you need to edit the usual /WEB-INF/web.xml file, using the ServerServlet adapter class from the Servlet extension as illustrated in the following listing.

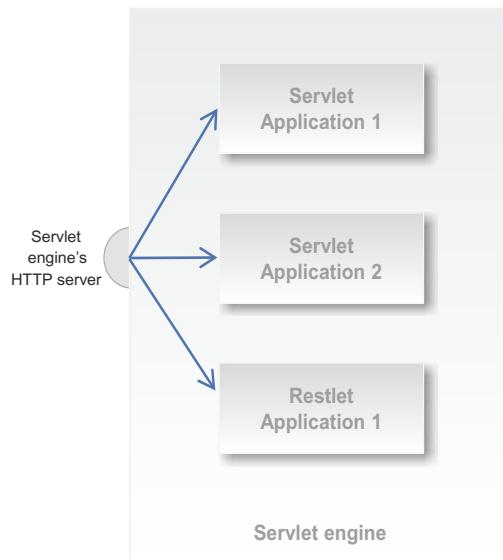


Figure 3.7 Using the Servlet engine as a container of Restlet applications

Listing 3.14 Configuring the Servlet's web.xml file

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns="http://java.sun.com/xml/ns/javaee"
           xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                               http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
           id="WebApp_ID" version="2.5">

    <display-name>
        Servlet engine as a container of Restlet applications
    </display-name>

    <servlet>
        <servlet-name>MailServerApplication</servlet-name>
        <servlet-class>
            org.restlet.ext.servlet.ServerServlet
        </servlet-class>
        <init-param>
            <param-name>org.restlet.application</param-name>
            <param-value>
                org.restlet.example.book.restlet.ch03.sec3.server.
                    MailServerApplication
            </param-value>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>MailServerApplication</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>

```

Create bridge Servlet

Point bridge to application

Map all URIs to bridge Servlet

Using the same JAR files as in the previous scenario, you can now deploy your WAR into Tomcat. This time, you can attach our Servlet WAR to any base URI—for example, to /rest/ in order to keep other root paths for other web applications, potentially including other bridged Restlet applications. After restarting Tomcat, you can access the `http://localhost:8080/rest/` URI in your browser to see the welcome message.

This example illustrates how portable Restlet applications are. The exact same code was used in both standalone Java SE and Java EE deployments. Note that in this scenario, it's possible to declare several Restlet applications at the same time. You need to give them different `servlet-name` values and adjust the `servlet-mapping` element accordingly.

In the next section, we look at another way to deploy Restlet applications within a Servlet engine—more precisely, inside an Oracle RDBMS!

3.5.4 The Oracle XML DB extension

The `org.restlet.ext.xdb` extension provides a Restlet server connector usable within an Oracle 11g database. It allows you to expose RESTful web APIs using the Oracle JVM, an embedded JVM compatible with Java 1.5 running inside the database. This

approach has the benefits of saving memory space and being fast due to local access to the SQL resources. In addition, Oracle XML DB provides a Servlet 2.2 engine that can work together with the Restlet Framework to process HTTP calls.

In order to deploy your Restlet application inside the Oracle JVM, you need to enable the HTTP service in Oracle and then use the `org.restlet.ext.xdb.XdbServer-Servlet` class instead of the usual `ServerServlet`. This special subclass is necessary due to the limited capabilities of Oracle's embedded Servlet engine (no session, no cookies, only one Servlet context supported, and no request dispatcher).

Those constraints are fine from a Restlet point of view, which makes it a good fit if you're already using an Oracle database to store the state of your Restlet resources. You can even benefit from native optimization when writing XML documents.

3.5.5 Restlet Framework as a library inside Servlet applications

The last scenario is sometimes preferred by Restlet users because it allows a fine-grained integration between existing Servlet applications and the Restlet Framework. It considers the Restlet Framework as a regular library or toolkit from which selected features, such as the `ClientResource` class or the `Representation` subclasses, can be used without bringing in the entire Restlet framework. Figure 3.8 illustrates this scenario with three Servlet applications deployed, one of them using Restlet as a library rather than as part of a comprehensive framework.

Another intermediary case is when you still want to use Restlet for server-side resources, but after some Servlet-based handling. For this purpose, the Servlet extension provides an `org.restlet.ext.servlet.ServletAdapter` class that has a `service(HttpServletRequest, HttpServletResponse)` method that's able to adapt a Servlet call into a Restlet call and to pass it to an attached Restlet defined by the `next` property. For example,

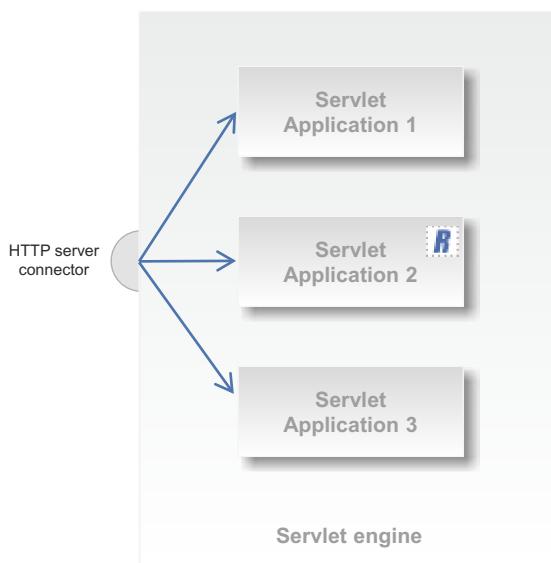


Figure 3.8 Using the Restlet Framework as a library inside Servlet applications

RestletFrameworkServlet class in the org.restlet.ext.spring extension makes use of the adapter class to provide another integration mode between Spring and Restlet.

We next introduce an alternative way to deploy Restlet applications, based on the highly modular and dynamic OSGi environments.

3.5.6 Dynamic deployment in OSGi environments

OSGi is a popular execution environment on top of the JVM that allows dynamic assembly of applications. Its main features are the ability to isolate applications running in the same JVM, even allowing deployment of several versions of the same application or library at the same time. It also provides hot deployment, hot update, and a loosely coupled service framework. The most famous OSGi example is the Eclipse IDE itself, but in recent years use has expanded to the server side as well.

The Restlet Framework provides out-of-the-box support for OSGi with a dedicated edition that ensures that each of its modules, including the Restlet core and extension JARs, are properly defined OSGi bundles. As a result, Restlet applications can be part of larger OSGi systems. Restlet can be embedded within OSGi bundles based on *activators*, entities that are triggered when bundles start and stop. In a Restlet context this enables you to start a Restlet server on bundle startup and stop it on bundle shutdown. The following listing describes how to implement this approach.

Listing 3.15 An OSGi activator to manage an embedded Restlet server

```
public class RestletComponentActivator implements BundleActivator {
    private Component component;

    public void start(BundleContext bundleContext) throws Exception {
        component = new Component();
        component.getServers().add(Protocol.HTTP, 8182);
        component.start();
    }

    public void stop(BundleContext bundleContext) throws Exception {
        if (component != null) {
            component.stop();
            component = null;
        }
    }
}
```

Starts Restlet component

Stops Restlet component

The activator then needs to be specified in the MANIFEST.MF file of the bundle that embeds it, as described in the following snippet:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Embedded Restlet Server
Bundle-SymbolicName: org.restlet.osgi.server
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: org.restlet.osgi.server.ServerActivator
Import-Package: org.osgi.framework;version="1.3.0"
Require-Bundle: org.restlet;bundle-version="2.1.0"
```

Because OSGi implements strict class loading, you must import the necessary packages and bundles corresponding to the classes you used in your activator (the OSGi package as well as the Restlet core bundle, in this case).

At this point, you have an embedded Restlet server in an OSGi container but no virtual hosts or applications are associated with it. You could configure everything within a single bundle, but you would lose modularity by not taking advantage of OSGi's dynamic capabilities. The idea here is to detect virtual hosts and automatically configure against the previous server. This is the classic OSGi approach, following the whiteboard pattern.

OSGi and the whiteboard pattern

This pattern is based on the OSGi service registry. Some providers register service implementations against a particular interface, and consumers listen for matching services. When such services are added or removed, consumers are notified and can adapt their processing.

This pattern allows simple decoupling of producers from consumers and uses the dynamic capabilities of OSGi.

For information see <http://www.osgi.org/wiki/uploads/Links/whiteboard.pdf>. (Peter Kriens and BJ Hargrave, 2001–2004, OSGi Alliance.)

Some bundles provide virtual hosts using the OSGi service registry. Each virtual host that needs to be added within the component is registered as an OSGi service. Activators of provided bundles are responsible for this registration and unregistration, as shown in the following listing.

Listing 3.16 An OSGi activator to manage registration of a virtual host

```
public class RestletComponentActivator implements BundleActivator {
    private ServiceRegistration registration;

    public void start(BundleContext bundleContext) throws Exception {
        VirtualHost virtualHost = createVirtualHost();
        this.registration
            = bundleContext.registerService(
                "org.restlet.routing.VirtualHost",
                virtualHost, null);
    }

    private VirtualHost createVirtualHost(){ ... }

    public void stop(BundleContext bundleContext) throws Exception {
        if (registration!=null) {
            registration.unregister();
        }
    }
}
```

**Registers
virtual host**

**Unregisters
virtual host**

Now you need to update the server bundle to listen to service updates—in particular, services of type `VirtualHost`. If a service is registered, the bundle detects it and adds it to the component. On unregistration, the bundle removes it from the component. This feature is based on listening mechanisms provided by OSGi.

The code contained in the following listing must be added at the end of the `start` method of the server activator.

Listing 3.17 Handling registrations and unregistrations of virtual host services

```
// Finding out already registered virtual hosts
ServiceReference[] virtualHostReferences
    = bundleContext.getServiceReferences(
        "org.restlet.routing.VirtualHost", null);
if (virtualHostReferences != null) {
    for (ServiceReference serviceReference : virtualHostReferences) {
        VirtualHost virtualHost = (VirtualHost)
            bundleContext.getService(serviceReference);
        // Register the virtual host
        (...)}
}

// Listening to virtual hosts services
ServiceListener virtualHostListener
    = new ServiceListener() {
    public void serviceChanged(ServiceEvent event) {
        if (!isVirtualHostService(event)) {
            return;
        }
        int type = event.getType();
        if (type == ServiceEvent.REGISTERED) {
            // Register the virtual host
            (...)}
        } else if (type == ServiceEvent.UNREGISTERING) {
            // Unregister the virtual host
            (...)}
    }
};
bundleContext.addServiceListener(virtualHostListener);
```

You need to find out whether some virtual host services are already present in the OSGi container ①. By default, bundles can randomly start, and the server bundle can start up after some bundles providing virtual host services. For those that will be added or removed after, a listener is implemented and registered ②. Its `serviceChanged` method is called on such events. After detecting the service type, the method handles the registration or the unregistration of the virtual host against the Restlet component.

Don't forget to remove the OSGi service listener on bundle shutdown in the `stop` method of the server activator, as described in the following snippet:

```
bundleContext.removeServiceListener(virtualHostListener);
```

Another promising way to use OSGi with Restlet is the Eclipse Virgo project (the contribution of SpringSource dm Server to the Eclipse Foundation, available at www.eclipse.org/virgo/), which provides a completely OSGified web application server alternative to the usual Java EE application servers.

You can also directly use Eclipse Gemini project (the contribution of SpringSource “Spring Dynamic Modules,” available at www.eclipse.org/gemini/) to let Spring manage your Restlet applications in an OSGi environment.

To learn more about deploying Restlet applications in OSGi, we recommend reading *Spring Dynamic Modules in Action* (Manning, 2011)—especially chapter 8, “Developing OSGi web components with Spring DM and web frameworks.” It even comes with a comprehensive deployment example for the Restlet Framework 2.0!

3.6 **Summary**

In this chapter you learned how to deploy Restlet applications *on premises*, a term used to describe machines that you physically control. You saw that the purpose of a Restlet component is to provide a RESTful web container for developers to host their Restlet applications and connect them with other REST components with client and server connectors. You saw that Restlet components are structured with four layers:

- 1 The outermost connectivity layer provides shared client and server connectors between all contained applications.
- 2 The service filtering layer provides services common to all contained applications such as access logging or global status handling.
- 3 The virtual hosting layer, available only on the server side, provides component-level pre-routing on host, port, and scheme; this allows, for example, several applications, each with its own domain name, to share the same IP address.
- 4 The central application handling layer handles inbound calls with Restlet applications and can issue outbound calls.

We also introduced the connectors provided in the Restlet Core and in Restlet Extensions, supporting a broad set of protocols and pseudoprotocols, including CLAP, FILE, FTP, HTTP/HTTPS, JAR/ZIP, JDBC, POP, RIAP, SOLR, and SMTP. We described the main characteristics of those connectors, helping you to make the best decision regarding which one to select in specific scenarios. You also learned how to use the virtual hosting and service filtering layers to, for example, map several IP addresses to a single domain name and to configure a web access log.

We described two alternative deployment mechanisms useful for administrators who can’t recompile Java code to configure things specific to deployment environments, such as port number and domain name. In those cases, declarative XML configuration files can be used, using the Spring Framework if necessary.

We also showed you some benefits of Restlet application portability by covering how Restlet edition for Java EE can let you easily deploy to Servlet engines such as Apache Tomcat, Oracle XML DB, and even OSGi environments. We described the main deployment scenarios with various levels of usage of Restlet.

You've now completed the "Getting started" part of this book. So far, we briefly introduced REST (additional details are available in appendix C) and then dove into Restlet coding, beginning the development of Restlet applications and resources and deploying them on premises with Restlet components.

You'll now start the second part of your journey, exploring in more depth important aspects of Restlet development, such as resource representations, security, documentation and more, giving you enough knowledge to roll out into production.

Part 2

Getting ready to roll out

P

art 1 showed that the Restlet Framework provides a comprehensive solution to developing your RESTful web projects. So far, we've put the framework in action with the RESTful mail system example by introducing only the concepts needed to get you started. This second part shares with you the necessary knowledge to change a prototype into a production-ready Restlet application.

Chapter 4 extensively covers the development of Restlet representations. Rather than simple strings, as in part 1, you'll expose and consume complex representations using languages such as XML, JSON, and HTML. You'll see how to use a minimal amount of code with techniques such as content negotiation and automatic binding with the converter service.

Chapter 5 explains how to secure a Restlet application, covering aspects such as confidentiality with TLS/SSL and user authentication. It covers authorization of specific actions using coarse- or fine-grained approaches as well as ways to ensure end-to-end data integrity—for example, to ensure nonrepudiation.

Chapter 6 goes over common requirements for RESTful web APIs: producing and maintaining user documentation and listing available resources, their URI templates, their exposed representations, and allowed methods. WADL is the de facto standard for this in RESTful applications, and chapter 6 discusses how it's supported in Restlet. We also discuss when and how to version your web API.

Chapter 7 talks about how to fulfill common needs such as processing web forms, handling file uploads, and setting cookies on web clients. You'll also learn best practices regarding testing and optimization of Restlet applications.

By the end of part 2 you should be equipped to develop a complete Restlet application and make it ready for deployment in production!

Producing and consuming Restlet representations



This chapter covers

- Producing and consuming XML and JSON representations
- Producing HTML using template representations
- HTTP content negotiation
- Simplifying representation handling with the converter service

In part 1 of the book, we had several opportunities to discuss representations. In chapter 1 we briefly explained that representations in REST are used to transfer resource state. We pointed to appendix A for an overview of the `org.restlet.representation` package and to appendix D for details on RESTful representation design, covering the definition of representation classes and the importance of hypermedia as the preferred mechanism to interact with an application in REST. Finally we manipulated basic representations in chapters 2 and 3 using the `StringRepresentation` class and using the converter service to return `String` instances directly, without needing to wrap them in a complete representation.

In this chapter we explore in detail how to create and consume real-world representations. As XML is a primary language for RESTful representations, a large part

of this chapter covers its deep support in Restlet. What you learn will also be applicable to other textual formats or media types. As an illustration we use a mail representation from appendix D and expose it in various formats using Restlet extensions.

4.1 Overview of representations

In the previous chapters we mainly explained how to work with resources and animate them with RESTful method calls. When needing to expose a representation on the server side or display one on the client side, we used the `StringRepresentation` class or the `String` class. Even though this is sufficient for basic cases, in real life you will find many reasons to go beyond that—for example, working with large representations and using automatic serialization and deserialization of regular Java objects to media types such as XML and JSON.

All those possibilities are based on the `org.restlet.representation` package introduced in chapter 1 and a set of Restlet extensions. In this section we explore this package, starting with root `Variant` and `RepresentationInfo` classes used to describe representation variants. We then continue with the abstract `Representation` class and the various subclasses provided to deal with byte and character content.

4.1.1 The Variant and `RepresentationInfo` base classes

This section provides a closer look at those two classes, starting with a class diagram in figure 4.1.

The purpose of the `Variant` class is to describe a representation with enough metadata for content negotiation to work. Content negotiation is a powerful feature of HTTP that we cover in section 4.5, but for now you can see `Variant` as the ancestor of all representations, containing the most important properties as detailed in table 4.1.



Figure 4.1 The `Variant` and `RepresentationInfo` classes are ancestors of all Restlet representations.

Table 4.1 Variant properties

Name	Description	Equivalent HTTP header
<code>characterSet</code>	Character set used to encode characters in textual representations.	Content-Type, charset parameter
<code>encodings</code>	Modifiable list of encodings applied. Encodings are modifiers of a representation's media type, useful to apply compression without losing the identity of the underlying media type or to mark a template representation with the type of template engine.	Content-Encoding

Table 4.1 Variant properties (continued)

Name	Description	Equivalent HTTP header
identifier	Optional identifier of the representation as a URI reference. If specified, it can be treated as an independent resource.	Content-Location
languages	Modifiable list of languages. Normally only one language is used for a single representation, but it can happen that several languages are interleaved.	Content-Language
mediaType	Media type that defines the exact format of the representation (that is, text/HTML, text/plain).	Content-Type

Note that each of the properties in table 4.1 relies on classes located in the `org.restlet.data` package—that is, `CharacterSet`, `Encoding`, `Language`, and `MediaType`. For the `identifier` property we rely on the `Reference` class, which allows flexible manipulation of any kind of URI reference. These classes provide constants for common metadata such as `CharacterSet.UTF_8`, `Encoding.ZIP`, or `MediaType.APPLICATION_XML`.

The `RepresentationInfo` class extends `Variant` to provide the two additional properties, shown in table 4.2. Its purpose is to support conditional method processing in an optimized way, without having to build full representations when a condition does not hold. First it adds the `tag` property based on the `org.restlet.data.Tag` class and a `modificationDate` property based on the usual `java.util.Date` class.

Table 4.2 RepresentationInfo properties

Name	Description	Equivalent HTTP header
<code>modificationDate</code>	Timestamp of this representation's last modification	<code>Last-Modified</code>
<code>tag</code>	Validation tag uniquely identifying the content of the representation	<code>ETag</code>

Because this sort of optimization is an advanced topic, we address it in chapter 7, section 7.4.5. For now, view it as a holder of additional representation properties you should try to set.

4.1.2 The `Representation` class and its common subclasses

Let's continue our overview of Restlet representations with the abstract `Representation` class itself, the superclass of all concrete representations. Figure 4.2 gives an overview of this class, its parent `RepresentationInfo` class, and its main properties and methods.

In REST, representations correspond to the current or intended state of a resource. They're composed of content as a sequence of bytes and metadata as a collection of properties.

The content of a representation can be retrieved several times if there is a stable and accessible source, like a local file or a string. When the representation is obtained via a temporary source like a network socket, its content can be retrieved only once. The `transient` and `available` properties are available to help you figure out those aspects at runtime.

Properties in addition to those provided by `Variant` and `RepresentationInfo` are added by the `Representation` class, as summarized in table 4.3. They use the `Digest`, `Disposition`, and `Range` classes from the `org.restlet.data` package.

In addition to all those properties, the `Representation` class provides many ways to consume its content, such as the `getStream()`, `getReader()`, `getChannel()`, and `getText()` methods that return, respectively, an instance of the `java.io.InputStream`, `java.io.Reader`, `java.nio.ReadableByteChannel`, and `java.lang.String` classes.

Representation
<pre> <<RepresentationInfo>> available : boolean availableSize : long digest : Digest disposition : Disposition empty : boolean expirationDate : Date range : Range size : long transient : boolean ... append(Appendable) : void exhaust() : long getChannel() : ReadableByteChannel getReader() : Reader getStream() : InputStream getText() : String hasKnownSize() : boolean release() : void setListener(ReadingListener) : void write(OutputStream) : void write(WritableByteChannel) : void write(Writer) : void ... </pre>

Figure 4.2 The abstract `Representation` class is the superclass of all Restlet representations.

Table 4.3 Additional `Representation` properties

Name	Description	Equivalent HTTP header
<code>available</code>	Indicates whether some fresh content is available, without having to consume it.	N/A
<code>availableSize</code>	Size effectively available. This value is the same as the <code>size</code> property except for partial representation where this is the size of the selected range.	Content-Length and Content-Range
<code>digest</code>	Value and algorithm name of the digest or checksum associated to the representation.	Content-MD5
<code>disposition</code>	Suggested download filename for this representation.	Content-Disposition
<code>expirationDate</code>	Future timestamp when this representation expires.	Expires
<code>range</code>	Range within the full content where the available partial content applies.	Content-Range
<code>size</code>	Size in bytes if known, <code>UNKNOWN_SIZE (-1)</code> otherwise.	Content-Length
<code>transient</code>	Indicates whether the representation's content is <i>transient</i> , meaning that it can be obtained only once.	N/A

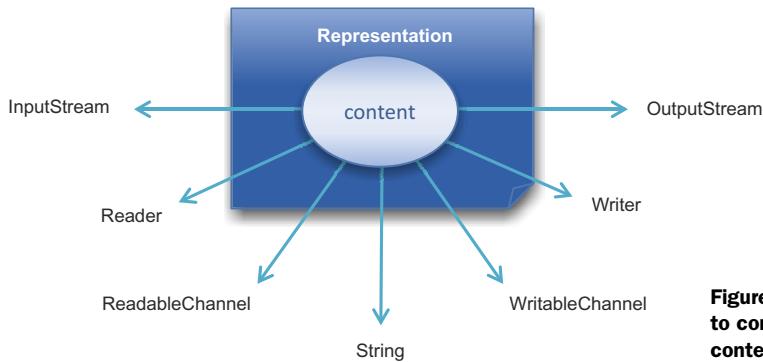


Figure 4.3 The various ways to consume representations' content

Another common way to consume the content of a representation is to ask it to write itself onto a target medium like a `java.io.OutputStream`, `java.io.Writer`, or `java.nio.WritableByteChannel` via one of the available `write(...)` methods. (Note that this approach requires the calling thread to block, which isn't optimal for asynchronous handling.) Figure 4.3 summarizes how the content of any Restlet representation can be consumed.

When you need to provide a concrete representation, you rarely start from scratch with this abstract `Representation` class but instead use one of its subclasses. The four class diagrams in figures 4.4, 4.5, 4.6, and 4.7 present the hierarchy of base representations provided in the core Restlet API. (Classes with italicized names are abstract.)

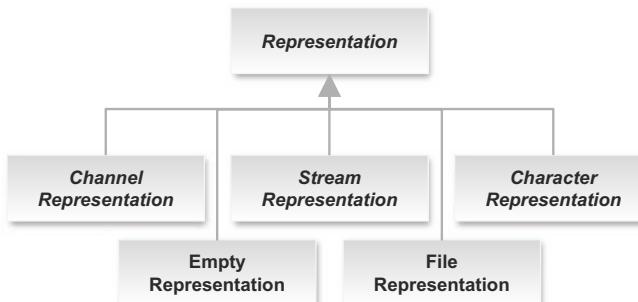


Figure 4.4 Subclasses of representation

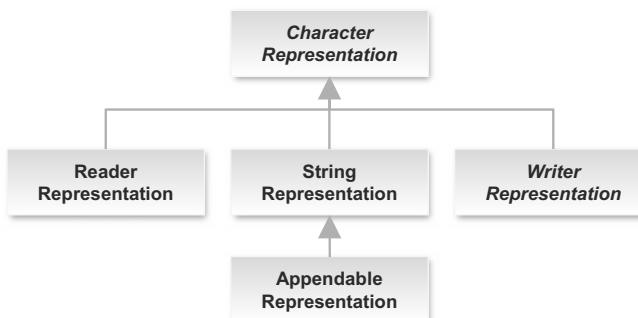


Figure 4.5 Character-based representation classes

You've already seen `StringRepresentation` in earlier chapters, but notice in figure 4.5 that a convenient subclass called `AppendableRepresentation` lets you build a string in several steps without having to supply the full text at construction time.

Keep in mind that additional representations are provided by Restlet extensions. We cover some of these extensions later in the book, but we encourage you to check the Javadocs and additional online documentation for comprehensive coverage.

As explained later in this chapter, the converter service reduces the need to manipulate those representation classes directly, but it's nonetheless important to be familiar with them and to be able to manipulate them occasionally. The following listing shows a little sample of how an `AppendableRepresentation` wraps textual content that grows dynamically and that can be manipulated in various ways.

Listing 4.1 Manipulating the `AppendableRepresentation`

```
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.Writer;

import junit.framework.TestCase;

import org.restlet.engine.io.BioUtils;
import org.restlet.representation.AppendableRepresentation;

public class AppendableTestCase extends TestCase {

    public void testAppendable() throws IOException {
        AppendableRepresentation ar = new AppendableRepresentation();
        ar.append("abcd");
        ar.append("1234");
        assertEquals("abcd1234", ar.getText());
        ar.append('\n');           ← Append new line character
        ar.write(System.out);
        BioUtils.copy(ar.getStream(), System.out);

        Writer writer = new OutputStreamWriter(System.out);
        ar.write(writer);
    }
}
```

Write content to console's output stream

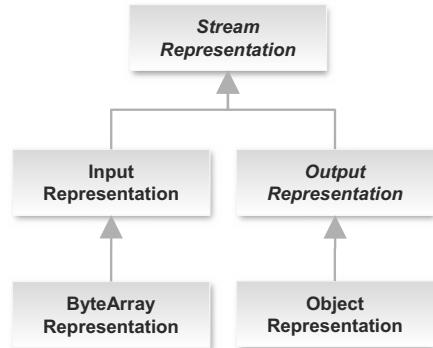


Figure 4.6 BIO stream-based representation classes

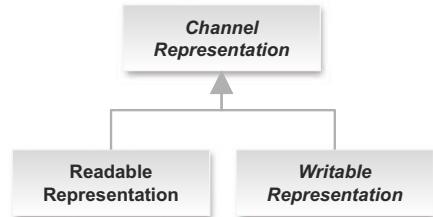
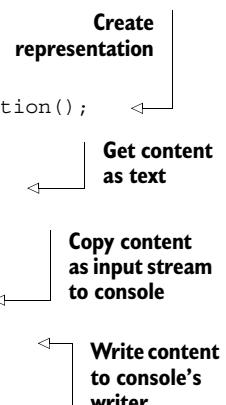


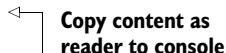
Figure 4.7 NIO channel-based representation classes



```

        BioUtils.copy(ar.getReader(), writer);
    }
}

```



Copy content as reader to console

If you run this test case, it should succeed as no exception is thrown and the assertion is valid. The console should also print four lines with the abcd1234 content. We will have other opportunities to discuss these representation classes, such as in section 7.4 when explaining how to improve the performance of your Restlet applications.

Although the classes discussed define the foundation of all Restlet representations, you will often prefer to use more specific classes, such as `SaxRepresentation`, `JaxbRepresentation`, `JsonRepresentation`, and `JacksonRepresentation`, to handle complex structures like XML or JSON documents. The goal is to use higher levels of abstraction instead of working with representations at the byte or character level. The next section extensively covers different ways of dealing with XML representations.

4.2 Producing and consuming XML representations

As XML is the most popular format used by REST resources to expose their state as representations, it's essential to cover how the Restlet Framework supports them. XML is as important for programmatic REST clients as HTML is for human REST clients. It's a generic markup language with numerous concrete applications such as Atom for web feeds, DocBook for technical publications, RDF for semantic description of resources, and SVG for vector graphics, among others.

As you'll see in this section, the Restlet Framework provides numerous powerful and extensible ways to produce XML representations, consume and validate them, transform them, extract specific content, and bind them to regular Java beans.

The first ways to produce and consume representations as XML documents are based on the standard DOM and SAX APIs, which offer very fine-grained levels of manipulation—for example, allowing evaluation of XPath expressions, handling of XML namespaces, and validation against XML schemas. We introduce three convenient Restlet extensions (`org.restlet.ext.jaxb`, `org.restlet.ext.jibx`, and `org.restlet.ext.xstream`) supporting the automatic binding between XML representations and regular Java objects. JAXB is a standard API built into Java SE since version 6, and JiBX and XStream are powerful open source libraries.

To illustrate those features, we represent the Homer mail resource number 123 introduced in section 2.4.7 as an XML document. This resource corresponds to a specific email resource that would have been received by the Homer user of our RESTful mail system. Having an XML representation of this email allows its retrieval in other applications—for example, in indexing and search purposes. Figure 4.8 illustrates this use case, with the resource identified by the `/accounts/chunkylover53-mails/123` absolute URI.

For reference, the following listing shows the XML mail document that we want to expose and consume as a representation of the Homer mail resource.



Figure 4.8 Example resource exposing an XML document as representation

Listing 4.2 The target XML mail representation

```

<mail>
    <status>received</status>
    <subject>Message to self</subject>
    <content>Doh!</content>
    <accountRef>
        http://www.rmep.org/accounts/chunkylover53/
    </accountRef>
    ...
</mail>
  
```

Before using DOM and SAX support available in Restlet via the `DomRepresentation` and `SaxRepresentation`, you should know about their parent `XmlRepresentation` class.

4.2.1 The `org.restlet.ext.xml.XmlRepresentation` class

The `XmlRepresentation` abstract class has been provided since version 2.0 of the framework in the `org.restlet.ext.xml` extension. As illustrated in figure 4.9, it first contains a set of mostly Boolean properties that allows the configuration of SAX and DOM parsers in a way similar to the properties defined in the standard JAXP API provided in Java SE in the `javax.xml.parser` package.

The default values of those properties are typically sufficient to get started, so we don't present them in detail in this book, but you can refer to Restlet Javadocs if you want to know more about each of them. Additional properties are also available and will be introduced in other subsections.

In the methods compartment, we find two constructors and four methods that can be used to manipulate an `XmlRepresentation` instance as a JAXP source object for integration purposes. The last static method is a convenience method that can aggregate all

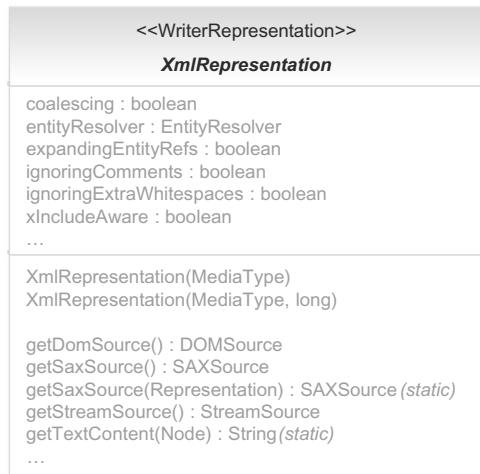


Figure 4.9 Partial details of the base `XmlRepresentation` class

the text content of a given DOM node. More methods are available; they're introduced in the subsections covering XML namespace handling, XPath expression evaluation, and XML schema validation. Speaking of DOM, the next section shows how to concretely build and parse XML representation as DOM documents.

4.2.2 Using the DOM API

The Document Object Model (DOM) API was defined by the W3C and provides a standard way to manipulate an XML document in Java as a hierarchy of node objects. As the document is fully stored in memory, this API allows node navigation, insertion and removal. The main drawback is the memory consumed, especially for large documents.

In Java SE, the `org.w3c.dom` package and subpackages provide the DOM API itself, mostly as a set of Java interfaces, whereas the `javax.xml.parser` package provides the classes to parse XML documents as DOM documents. In Restlet, we support DOM-driven manipulation of XML documents with `DomRepresentation`, also part of the `org.restlet.ext.xml` extension.

As illustrated in figure 4.10, the `DomRepresentation` class contains two properties, `document` and `indenting`. The `document` property is the DOM document object that has been parsed or that has been created empty (by default) and that can be written as XML content using the standard representation methods such as `getStream()` or `write(OutputStream)`. The `indenting` Boolean property indicates if XML content should be pretty printed with nice indentation or written in a more compact way.

Regarding constructors, the first two assume that you want to start with an empty DOM document that you will later edit via the `document` property, and the third provides an existing `Document` instance as a starting point. The last constructor can parse XML representations, independent of the `Representation` subclass used, into a DOM document object. Note that in this case the parsing is done lazily, only when the `getDocument()` method is invoked. Providing several representation constructors for different purposes is a common and important Restlet API design pattern.

The methods compartment contains three methods that can be useful for advanced scenarios. For example, `createTransformer()` can be overridden to customize the way the DOM document is serialized as XML.

For the mail server example, imagine that you want to use DOM to produce the XML representations of emails exchanged with your clients. As illustrated in the following listing, we first create a simple test application where you can attach the `MailServerResource` class on the proper URI template.

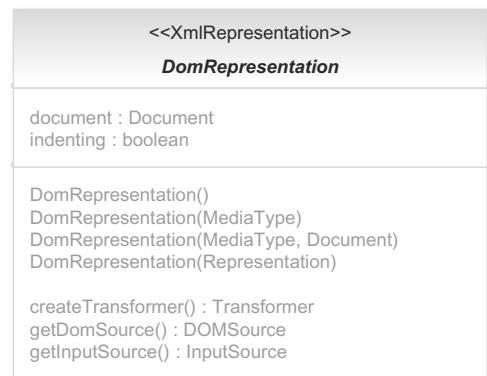


Figure 4.10 Properties and methods of the `DomRepresentation` class

Listing 4.3 Simple test application serving mail resources

```

public class MailServerApplication extends Application {
    public static void main(String[] args) throws Exception {
        Component mailServer = new Component();
        mailServer.getServers().add(Protocol.HTTP, 8111);
        mailServer.getDefaultHost().attach(new MailServerApplication());
        mailServer.start();
    }

    @Override
    public Restlet createInboundRoot() {
        Router router = new Router(getContext());
        router.attach("/accounts/{accountId}/mails/{mailId}",
                      MailServerResource.class);
        return router;
    }
}

```

The code is annotated with several arrows pointing to specific parts of the code:

- An arrow from the right points to the `mailServer.start()` call with the text "Launch application with HTTP server".
- An arrow from the right points to the `Router router = new Router(getContext());` line with the text "Create root Router to dispatch calls".
- An arrow from the left points to the `createInboundRoot()` method with the text "Append root node".

Nothing much is new in listing 4.3, so check out listing 4.4 for the `MailServerResource` class, which uses a `DomRepresentation`. This class supports the GET and PUT methods to illustrate the generation of a DOM representation as well as its parsing.

Listing 4.4 Mail server resource using the DOM API

```

public class MailServerResource extends ServerResource {
    @Get
    public Representation toXml() throws IOException {
        DomRepresentation result = new DomRepresentation();
        result.setIndenting(true);
        Document doc = result.getDocument();
        Node mailElt = doc.createElement("mail");
        doc.appendChild(mailElt);
        Node statusElt = doc.createElement("status");
        statusElt.setTextContent("received");
        mailElt.appendChild(statusElt);
        Node subjectElt = doc.createElement("subject");
        subjectElt.setTextContent("Message to self");
        mailElt.appendChild(subjectElt);
        Node contentElt = doc.createElement("content");
        contentElt.setTextContent("Doh!");
        mailElt.appendChild(contentElt);
        Node accountRefElt = doc.createElement("accountRef");
        accountRefElt.setTextContent(new Reference(getReference(), "...")
                                     .getTargetRef().toString());
        mailElt.appendChild(accountRefElt);
        return result;
    }

    @Put
    public void store(DomRepresentation mailRep) throws IOException {
}

```

The code is annotated with several arrows pointing to specific parts of the code:

- An arrow from the left points to the `result.setIndenting(true);` line with the text "Append root node".
- An arrow from the right points to the `DomRepresentation result = new DomRepresentation();` line with the text "Create empty DOM representation".
- An arrow from the right points to the `Document doc = result.getDocument();` line with the text "Ensure pretty printing".
- An arrow from the right points to the `Node mailElt = doc.createElement("mail");` line with the text "Retrieve DOM document to populate".
- An arrow from the right points to the `statusElt.setTextContent("received");` line with the text "Append child nodes and set text content".
- An arrow from the right points to the `accountRefElt.setTextContent(...)` line with the text "Compute the parent URI including slash".

```

Document doc = mailRep.getDocument();
Element mailEl = doc.getDocumentElement();
Element statusEl = (Element)
mailEl.getElementsByTagName("status").item(0);
Element subjectEl = (Element)
mailEl.getElementsByTagName("subject").item(0);
Element contentEl = (Element)
mailEl.getElementsByTagName("content").item(0);
Element accountRefEl = (Element) mailEl.getElementsByTagName(
    "accountRef").item(0);

System.out.println("Status: " + statusEl.getTextContent());
System.out.println("Subject: " + subjectEl.getTextContent());
System.out.println("Content: " + contentEl.getTextContent());
System.out.println("Account URI: " + accountRefEl.getTextContent());
}

```

Output XML element values for debugging

Parse and normalize DOM document

If you're familiar with the DOM API, listing 4.4 should look simple and natural. If not, we recommend reading the *Java & XML* book [2] which also covers other ways to manipulate XML in Java, which we explore in the rest of this chapter.

The following listing shows a simple mail client that will first GET the XML mail and then PUT it back in the same state.

Listing 4.5 Mail client retrieving a mail, then storing it again on the same resource

```

public class MailClient {

    public static void main(String[] args) throws Exception {
        ClientResource mailClient = new ClientResource(
            "http://localhost:8111/accounts/chunkylover53-mails/123");
        Representation mailRepresentation = mailClient.get();
        mailClient.put(mailRepresentation);
    }
}

```

To try it out, we first launch the `MailServerApplication` class, thanks to its main method, and then the `MailClient`. The output you should see in the console, ensuring that DOM document creation, serialization, deserialization, and parsing are working as expected, looks like this:

```

Status: received
Subject: Message to self
Content: Doh!
Account URI: http://localhost:8111/accounts/chunkylover53/

```

Let's now imagine that the mail XML representation is getting very large and that your server has to handle many of them at the same time. You can't hold all the DOM documents in memory, so you need to look at alternative approaches. The SAX API introduced in the next subsection is an excellent solution to this problem.

4.2.3 Using the SAX API

The Simple API for XML (SAX), defined by David Megginson, provides a standard way to manipulate an XML document in Java as a stream of events. Because the document isn't fully stored in memory, this API allows processing of infinitely large XML documents. The main drawback is that it's harder to process and modify XML because it must happen on the fly, whereas the DOM API is easier to process with but can use a lot of memory.

In Java SE, the `org.xml.sax` package and subpackages provide the SAX API itself, as a set of Java interfaces and classes, whereas the `javax.xml.parser` package provides the classes to parse XML documents as SAX event streams. In Restlet we support SAX-driven parsing and writing of XML documents with `SaxRepresentation`, also part of the `org.restlet.ext.xml` extension.

As illustrated in figure 4.11, the `SaxRepresentation` class contains a single source property that is the source of SAX events, typically a Restlet XML representation. Note that this property only *describes* the source—in the same way that `java.io.File` is only a file descriptor. In order to get the XML content, you need to explicitly call the `parse(ContentHandler)` method. On the other side, if you want to write a SAX representation, you need to produce SAX events on the fly by overriding the `write(XmlWriter)` method. Regarding constructors, the first two assume that you want to produce SAX events, and the other constructors will set up the `saxSource` property to parse XML content.

The following listing illustrates those features with the same example that we used for the `DomRepresentation`, but this time using a `SaxRepresentation`.

Listing 4.6 Mail server resource using the SAX API

```
public class MailServerResource extends ServerResource {
    @Get
    public Representation toXml() {
        SaxRepresentation result = new SaxRepresentation() { ←
            public void write(org.restlet.ext.xml.XmlWriter writer)
                throws IOException {
                try { ←
                    writer.startDocument();
                    writer.startElement("mail"); ←
                    ←
                    ←
                }
            }
        };
    }
}
```

```

writer.startElement("status");
writer.characters("received");
writer.endElement("status");

writer.startElement("subject");
writer.characters("Message to self");
writer.endElement("subject");

writer.startElement("content");
writer.characters("Doh!");
writer.endElement("content");

writer.startElement("accountRef");
writer.characters(new Reference(getReference(), "...")
    .getTargetRef().toString());
writer.endElement("accountRef");

writer.endElement("mail");
writer.endDocument();
} catch (SAXException e) {
    throw new IOException(e.getMessage());
}
};

return result;
}

@Override
public void store(SaxRepresentation mailRep) throws IOException {
    mailRep.parse(new DefaultHandler() {

        @Override
        public void startElement(String uri, String localName,
            String qName, Attributes attributes) throws SAXException
        {
            if ("status".equals(localName)) {
                System.out.print("Status: ");
            } else if ("subject".equals(localName)) {
                System.out.print("Subject: ");
            } else if ("content".equals(localName)) {
                System.out.print("Content: ");
            } else if ("accountRef".equals(localName)) {
                System.out.print("Account URI: ");
            }
        }

        @Override
        public void characters(char[] ch, int start, int length)
            throws SAXException {
            System.out.print(new String(ch, start, length));
        }

        @Override
        public void endElement(String uri, String localName, String
qName)
            throws SAXException {

```

Append child nodes,
set text content

End root
node

End
document

Output XML
element
names

Output XML element values

```
        System.out.println();
    }
}
});  
}
```

If you execute the `MailClient` again with this different code, you will obtain the exact same output on the console. Note how we used an anonymous subclass of SAX's `DefaultHandler` to get called back by the SAX parser each time a parsing event occurs, such as the start of an element. In this case the parsing logic is rather straightforward, but it can quickly get complex if your XML structure is deep or if the same element name is used in several parts of the document, as you will have to maintain some sort of state machine. The great advantage again is the ability to parse very large XML representations using limited system resources.

Now let's look at a convenient way to extract specific content from an XML representation using XPath.

4.2.4 Evaluating XPath expressions

When we introduced the `XmlRepresentation` class, we didn't list all the available methods. The additional methods in figure 4.12 evaluate XML Path Language (XPath) expressions. XPath is a W3C recommendation that can be used to address specific parts of an XML document. It can be used standalone (explained soon), inside XSLT documents (see subsection 4.2.7), or in other places such as XQuery or XPointer (not covered in this book—refer to other books or online documentation for more information). In Java SE, the `javax.xml.xpath` package contains a standard API for XPath engines.

<<WriterRepresentation>>

XmlRepresentation

...

getBoolean(String) : Boolean
getNode(String) : Node
getNodes(String) : NodeList
getNumber(String) : Double
getText(String) : String

...

The `XmlRepresentation` class and its two concrete `DomRepresentation` and `SaxRepresentation` subclasses can call those methods to evaluate an XPath expression passed as a string parameter and return a single Boolean, a DOM Node, a Double number, or character String.

When the expression is expected to return several items, you can use `getNodes(String)` which returns an instance of `org.restlet.ext.xml.NodeList`. Note that this Restlet's `NodeList` class implements both the `List<Node>` and DOM's `NodeList` interface by wrapping an instance of DOM's `NodeList`. The advantage is that you can use this structure in a loop thanks to its iterator, whereas DOM's interface has to be manually iterated.

The following listing shows the result of implementing the PUT method of the `MailServerResource` to use XPath expression to get the elements' content.

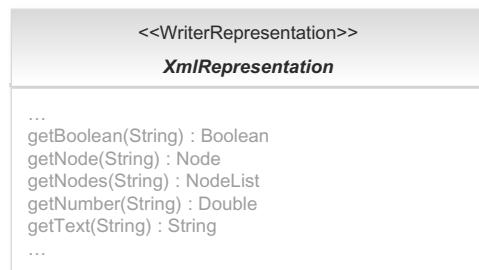


Figure 4.12 XPath-related methods of the `XmlRepresentation` class

Listing 4.7 Mail server resource using the XPath API

```

@Put
public void store(DomRepresentation mailRep) {
    String status = mailRep.getText("/mail/status");
    String subject = mailRep.getText("/mail/subject");
    String content = mailRep.getText("/mail/content");
    String accountRef = mailRep.getText("/mail/accountRef");

    System.out.println("Status: " + status);
    System.out.println("Subject: " + subject);
    System.out.println("Content: " + content);
    System.out.println("Account URI: " + accountRef);
}

```

Retrieve XML element using XPath expressions

Output XML element values

As before, if you run the `MailClient` you will obtain the same output in the console. Compared to the DOM and SAX equivalents listed earlier, this PUT method implementation is very simple. And we barely used XPath's capabilities; there is an extensive XPath functions library.

The advanced topic of XML namespaces and how the `XmlRepresentation` subclasses can handle them continues our exploration of XML support in Restlet.

4.2.5 Handling XML namespaces

XML namespaces allow the mixing of XML elements and attributes from several vocabularies in the same XML document. Namespaces are identified by URI references, and prefixes are used as shortcuts.

In Java SE, XML namespaces are supported by two packages: `javax.xml` with its `XMLConstants` class containing common namespace URI references and prefixes, as well as the `javax.xml.namespace` package and its `NamespaceContext` interface, which is implemented by our `XmlRepresentation` class. Figure 4.13 lists the three methods of `NamespaceContext` plus the `namespaceAware` Boolean property used to activate its support in the `DomRepresentation` and `SaxRepresentation` subclasses, and the `namespaces` property, containing a modifiable map of namespace prefix and URI references.

Now it's time to add XML namespace support to the example `MailServerResource`. In the following listing we update the example based on `DomRepresentation` to create an XML element inside a dedicated RMEP namespace with a URI of `www.rmep.org/namespaces/1.0`.

Listing 4.8 Mail server resource-handling XML namespaces

```

@Get
public Representation toXml() throws IOException {
    DomRepresentation result = new DomRepresentation();

```

<<WriterRepresentation>> XmlRepresentation
... namespaceAware : boolean namespaces : Map<String, String> ...
... getNamespaceURI(String) : String getPrefix(String) : String getPrefixes(String) : Iterator<String> ...

Figure 4.13 XML namespace-related properties and methods of the `XmlRepresentation` class

```

result.setIndenting(true);
result.setNamespaceAware(true);                                ← XML namespace configuration

Document doc = result.getDocument();                         ← Populate DOM document
Node mailElt = doc.createElementNS(
    "http://www.rmep.org/namespaces/1.0", "mail");
doc.appendChild(mailElt);

Node statusElt = doc.createElement("status");
statusElt.setTextContent("received");
mailElt.appendChild(statusElt);

Node subjectElt = doc.createElement("subject");
subjectElt.setTextContent("Message to self");
mailElt.appendChild(subjectElt);

Node contentElt = doc.createElement("content");
contentElt.setTextContent("Doh!");
mailElt.appendChild(contentElt);

Node accountRefElt = doc.createElement("accountRef");
accountRefElt.setTextContent(new Reference(getReference(), "...")
    .getTargetRef().toString());
mailElt.appendChild(accountRefElt);
return result;
}

@Put
public void store(DomRepresentation mailRep) {
    String rmepNs = "http://www.rmep.org/namespaces/1.0";
    mailRep.setNamespaceAware(true);
    mailRep.getNamespaces().put("", rmepNs);
    mailRep.getNamespaces().put("rmep", rmepNs);                ← XML namespace configuration

    String status = mailRep.getText("/:mail/:status");
    String subject = mailRep.getText("/rmep:mail/:subject");
    String content = mailRep.getText("/rmep:mail/rmep:content");
    String accountRef = mailRep.getText("/:mail/rmep:accountRef");                ← Retrieve XML element using XPath expressions

    System.out.println("Status: " + status);
    System.out.println("Subject: " + subject);
    System.out.println("Content: " + content);                    ← Output XML element values
    System.out.println("Account URI: " + accountRef);
}

```

In the GET handling method, we used the `createElementNS(String, String)` method from the DOM API to create XML elements qualified with the RMEP namespace. If you try to obtain the XML representation in a browser entering the URI reference `http://localhost:8111/accounts/chunkylover53-mails/123`, you will obtain the following document. Note that a default namespace (without prefix) is declared with the `xmlns` attribute on the root `mail` element:

```

<?xml version="1.0" encoding="UTF-8"?>
<mail xmlns="http://www.rmep.org/namespaces/1.0">
    <status>received</status>
    <subject>Message to self</subject>
    <content>Doh!</content>

```

```
<accountRef>http://localhost:8111/accounts/chunkylover53/</accountRef>
</mail>
```

Coming back to listing 4.8, let's present the method-handling PUT requests. We still use the XPath approach but this time have to declare namespaces—otherwise the code from subsection 4.2.4 would no longer work. To illustrate the flexibility provided in XPath expressions, we declare two prefixes for the same RMEP namespace URI: “” for the default namespace and “rmepl.” Then we can qualify the XML elements to select, such as :mail or rmepl:mail. We encourage you to run the MailClient again to verify that the console output is still the same!

Even if this mechanism adds some complexity, it allows you to mix several XML languages in the same document, which is necessary with languages such as Atom XML for web feeds. Let's continue our exploration of XML features provided by Restlet with XML schemas—powerful mechanisms to validate the correctness of the structure and content of XML documents.

4.2.6 Validating against XML schemas

Imagine that you need to validate the XML documents submitted by users of your applications and warn them precisely about the cause of the refusal. How would you do it?

XML schema languages such as W3C XML Schema (XSD) and Relax NG (RNG) can help. Let's see how to use them in Restlet with the `XmlRepresentation` by first looking at the remaining properties and methods listed in figure 4.14.

Restlet largely relies on the features of the `javax.xml.validation` package to offer an integrated, simpler way to validate XML representations. For example, the `schema` property is an instance of the `Schema` class from this package, but there's also a `setSchema(Representation)` method that allows you to provide the schema using any type of Restlet representation—but with the media type correctly set (`application/x-xsd+xml` for W3C XML Schemas, `application/relax-ng-compact-syntax`, and `application/x-relax-ng+xml` for Relax NG).

Note that DTDs are an older way to validate XML documents, built directly inside the XML language. DTD has been largely deprecated, so we recommend you use XML Schema or modern alternatives such as Relax NG or Schematron instead. In case you need DTD validation at parsing time, you can turn the `validatingDtd` Boolean property to true.

Let's now illustrate how this could be used with our example mail resource. The W3C XML Schema in the following listing corresponds to our current mail XML format.

<<WriterRepresentation>>	
<code>XmlRepresentation</code>	
...	
errorHandler : ErrorHandler	
schema : Schema	
validatingDtd : boolean	
...	
validate(Schema) : void	
validate(Schema, Result) : void	
validate(Representation) : void	
validate(Representation, Result) : void	

Figure 4.14 XML schema validation-related properties and methods of the `XmlRepresentation` class

Listing 4.9 W3C XML Schema for the mail XML representations

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:rmeP="http://www.rmep.org/namespaces/1.0"
         targetNamespace="http://www.rmep.org/namespaces/1.0"
         elementFormDefault="qualified">
    <element name="mail">
        <complexType>
            <sequence>
                <element name="status" type="string" />
                <element name="subject" type="string" />
                <element name="content" type="string" />
                <element name="accountRef" type="string" />
            </sequence>
        </complexType>
    </element>
</schema>
```

This schema defines a root `mail` element as a sequence of four elements: `status`, `subject`, `content`, and `accountRef`, themselves of the datatype XSD string. With the proper definition of the target namespace, this schema matches the XML representation produced in the previous subsection.

The next listing shows how to ensure that the XML representations PUT to your mail resource validate against this schema. There are two steps: first set the `schema` property and then set the `errorHandler` to catch parsing errors and warnings.

Listing 4.10 Mail server resource handling XML namespaces

```
@Put
public void store(DomRepresentation mailRep) throws IOException {
    Representation mailXsd = new ClientResource(
        LocalReference.createClapReference(getClass().getPackage()
            + "/Mail.xsd").get();
    mailRep.setSchema(mailXsd);
    mailRep.setErrorHandler(new ErrorHandler() {
        public void error(SAXParseException exception) throws SAXException {
            throw new ResourceException(exception);
        }

        public void fatalError(SAXParseException exception)
            throws SAXException {
            throw new ResourceException(exception);
        }

        public void warning(SAXParseException exception)
            throws SAXException {
            throw new ResourceException(exception);
        }
    });
    String rmepNs = "http://www.rmep.org/namespaces/1.0";
    mailRep.setNamespaceAware(true);
```

Configure XML Schema for validation: A callout box with an arrow pointing to the line `mailRep.setSchema(mailXsd);`. It contains the text "Configure XML Schema for validation".

XML namespace configuration: A callout box with an arrow pointing to the line `mailRep.setNamespaceAware(true);`. It contains the text "XML namespace configuration".

```

mailRep.getNamespaces().put("", rmepNs);
mailRep.getNamespaces().put("rmep", rmepNs);

String status = mailRep.getText("/:mail/:status");
String subject = mailRep.getText("/rmep:mail/:subject");
String content = mailRep.getText("/rmep:mail/rmep:content");
String accountRef = mailRep.getText("/:mail/rmep:accountRef");

System.out.println("Status: " + status);
System.out.println("Subject: " + subject);
System.out.println("Content: " + content);
System.out.println("Account URI: " + accountRef);
}

```

Now you can run the `MailClient` against this new resource to verify that the XML representation received by the PUT method is indeed valid. Then change the name of an element in the GET method—for example, `mail` to `email`—to observe that a failure to validate produces a log entry on both the server and client sides, with an explicit description of the error.

Note that it's not usually a good idea to systematically require inbound XML documents to conform to a given schema. Web browsers are flexible about the format of the HTML documents they can parse, so you should consider giving this level of flexibility to your users, using XPath for parsing, for example. But when you send output XML documents, you should try to respect any provided schema as closely as possible. This general internet rule was nicely summarized by Jon Postel: “Be conservative in what you do; be liberal in what you accept from others.” (“DoD standard Transmission Protocol.” <http://tools.ietf.org/html/rfc761>.)

Restlet’s XML support isn’t limited to static XML documents. With XML transformations, you can produce representations in other text formats.

4.2.7 Applying XSLT transformations

XSL Transformations (XSLT) is a standard XML transformation language that is natively supported by Java SE, thanks to the `javax.xml.transform` package and sub-packages. It’s ideal for producing other markup-oriented XML languages such as HTML or XHTML or XML vocabularies.

Here we want to illustrate the use of the `TransformRepresentation` class, part of the `org.restlet.ext.xml` package described in figure 4.15. Note that unlike `DomRepresentation` and `SaxRepresentation` classes, `TransformRepresentation` is a one-way converter. The two important properties are `sourceRepresentation`, which defines the wrapped XML representation to be transformed, and `transformSheet`, which contains the XSLT document to apply to the source to obtain the resulting `TransformRepresentation`. Other properties support more advanced configuration options. All the methods mentioned provide ways to customize the default behavior.

To illustrate this feature we transform the example XML mail into a format used by Restlet’s extension for JavaMail.

<<WriterRepresentation>>
TransformRepresentation
outputProperties : Map<String, String>
parameters : Map<String, Object>
sourceRepresentation : Representation
templates : Templates
transformSheet : Representation
uriResolver : UriResolver
TransformRepresentation(Context, Representation, Representation)
TransformRepresentation(Representation, Representation)
...
toSaxSource(Representation) : SAXSource (<i>static</i>)
getSaxSource() : SAXSource
getTransformer() : Transformer
getTransformerHandler() : TransformerHandler
getXmlFilter() : XMLFilter
transform(Source, Result) : void
write(Result) : void

Figure 4.15 Properties and methods of the TransformRepresentation class

INTRODUCING JAVAMAIL EXTENSION

To add support for the SMTP and POP 3 protocols to your Restlet application, you can use the JavaMail API directly in your resource classes. But the `org.restlet.ext.javamail` extension provides an easier-to-use and more flexible alternative that's also more RESTful. The idea is to use a pseudoprotocol based on the SMTP and POP URI schemes, as in

```
smtp://host[:port]
pop://host[:port]
```

Authentication is handled using the `ChallengeResponse` class (introduced in chapter 5) and the `SMTP_PLAIN`, `POP_BASIC`, and `POP_DIGEST` authentication schemes. The client connector provided is capable of sending and receiving emails using either the JavaMail Message class or Restlet's specific XML format, as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<email>
    <head>
        <subject>Account activation</subject>
        <from>support@restlet.org</from>
        <to>user@domain.com</to>
        <cc>log@restlet.org</cc>
    </head>
    <body>
        <! [CDATA[Your account was successfully created!] ]>
    </body>
</email>
```

When sending an email out, you need to POST an email representation to the relaying SMTP server defined by the target `smtp://` URI. When reading an email from your inbox, you GET it from the POP3 server defined by the target `pop://` URI. Generally you first need to retrieve the representation of the list of available emails as Restlet's specific XML document.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<emails>
    <email href="/1234"/>
    <email href="/5678"/>
    <email href="/9012"/>
    <email href="/3456"/>
</emails>
```

Now that you have a better understanding of the JavaMail extension and the XML representations required, you can use XSLT to achieve your original goal.

APPLYING THE XSLT TRANSFORMATION

The next listing contains the XSLT transform sheet that we'll apply to the example XML mail representation to obtain the desired JavaMail XML format. We could use this capability to provide a bridge between our RESTful mail system and traditional SMTP and POP servers.

Listing 4.11 XSLT transform sheet

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <email>
            <head>
                <subject>
                    <xsl:value-of select="/mail/subject" />
                </subject>
                <from>chunkylover53@rmep.org</from>
                <to>test@domain.com</to>
            </head>
            <body>
                <xsl:value-of select="/mail/content" />
            </body>
        </email>
    </xsl:template>
</xsl:stylesheet>
```

Email addresses
hard-coded for
simplicity

The transform sheet contains a skeleton for the target XML document we want to produce, including `xsl:value-of` elements using XPath expressions in `select` attributes to dynamically insert content from the source XML representation.

Look at the modified MailServerResource in the following listing.

Listing 4.12 Mail server resource applying an XSLT transformation

```
@Get
public Representation toXml() throws IOException {
    DomRepresentation rmepMail = new DomRepresentation();
    [...]
    Representation transformSheet = new ClientResource(
        LocalReference.createClapReference(getClass().getPackage()
            + "/Mail.xslt").get();
```

Transform to
another XML
format using
XSLT

Content similar to subsection 4.2.2 and 4.2.4

```

        TransformRepresentation result = new TransformRepresentation(rmepMail,
            transformSheet);
        result.getOutputProperties().put(OutputKeys.INDENT, "yes");
        return result;
    }

    @Put
    public void store(DomRepresentation mailRep) {
        String subject = mailRep.getText("/email/head/subject"); ←
        String content = mailRep.getText("/email/body");

        System.out.println("Subject: " + subject); ←
        System.out.println("Content: " + content); ←
    }
}

```

Retrieve XML element using XPath expressions

Output XML element values

As you can see, the code is straightforward; the hardest part is writing the XSLT transform sheet. If you call GET on the resource in your browser, you will obtain the following output:

```

<?xml version="1.0" encoding="UTF-8"?>
<email>
<head>
<subject>Message to self</subject>
<from>chunkylover53@rmep.org</from>
<to>test@domain.com</to>
</head>
<body>Doh!</body>
</email>

```

Now you can run the MailClient again and observe the expected values for the subject and body XML elements on the console log.

The Transformer class in `org.restlet.ext.xml` can provide a Restlet filter to systematically apply the same XSLT transformation to incoming request entities or to outgoing response entities.

This overview of the Restlet XML extension is now over, but the Restlet Framework provides additional ways to handle XML, thanks to the integration with XML binding libraries such as JAXB, JiBX, and XStream. These are the topics of the next three subsections.

4.2.8 Using the JAXB extension

Java Architecture for XML Binding (JAXB) is a standard Java API that's built into Java SE version 6.0 but is also usable with Java SE 5.0 as an additional library. Compared to DOM and SAX, it works at a higher level, providing marshaling (serialization) and unmarshaling (deserialization) between Java classes and XML documents.

You can use JAXB in three ways. The first and most common is to start from a W3C XML Schema and generate the corresponding Java classes using the provided `xjc` tool. The second way uses the provided `schemagen` tool to derive an XML Schema from Java classes. And you can manually annotate your own Java classes to generate the correct XML documents. All three approaches can be customized and

work fine, but starting from an XML Schema is definitely a convenient and productive way.

Let's generate the JAXB artifacts from the Mail.xsd file that we introduced in subsection 4.2.6; the following listing shows the command line to accomplish this.

Listing 4.13 Compiling the XML Schema into JAXB annotated Java classes

```
xjc -npa -p org.restlet.example.book.restlet.ch04.sec2.sub8
    org\restlet\example\book\restlet\ch04\sec2\sub8\Mail.xsd
```

The xjc tool produces two Java classes in the example package: Mail, containing your four simple properties, and ObjectFactory, acting as both a factory and a JAXB registry.

The Restlet extension for JAXB is available in the org.restlet.ext.jaxb package. The main class, JaxbRepresentation<T>, is illustrated in figure 4.16. It either parses an XML representation into a given Java class or generates an XML representation given a Java object. All properties are optional; in this example we set the formattedOutput property to get nice indentation and line breaks for easier debugging in a web browser.

It's time to rework the mail example using JAXB and compare with previous approaches. The following listing contains the new MailServerResource.

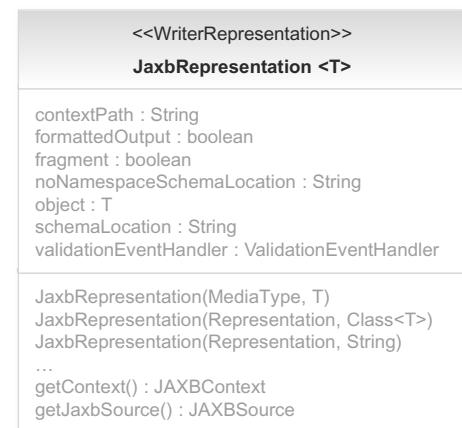


Figure 4.16 Properties and methods of the JaxbRepresentation class

Listing 4.14 Compiling the XML Schema into JAXB annotated Java classes

```

@Get
public Representation toXml() {
    Mail mail = new Mail();
    mail.setStatus("received");
    mail.setSubject("Message to self");
    mail.setContent("Doh!");
    mail.setAccountRef(new Reference(getReference(), "..."))
    .getTargetRef().toString());
}

    JaxbRepresentation<Mail> result = new
    JaxbRepresentation<Mail>(mail);
    result.setFormattedOutput(true);
    return result;
}

@Put
public void store(Representation rep) throws IOException {
    JaxbRepresentation<Mail> mailRep = new

```

Create mail bean

Wrap bean with JAXB representation

Parse XML representation to get mail bean

```
JaxbRepresentation<Mail>(rep, Mail.class);
    Mail mail = mailRep.getObject();

    System.out.println("Status: " + mail.getStatus());
    System.out.println("Subject: " + mail.getSubject());
    System.out.println("Content: " + mail.getContent());
    System.out.println("Account URI: " + mail.getAccountRef());
}

```

 **Output XML element values**

The resulting code is very simple, and the parsing logic produced is optimized and naturally provides schema validation. Although you gained productivity, you lost some flexibility, such as the ability to interact with the XML parsing logic. For example, it's harder to make evolutions to schema without requiring changes to existing clients of your application. This is the usual drawback of these kinds of abstractions, but they're so powerful that it is hard to resist using them!

In addition to JAXB, the Restlet Framework supports several alternatives, which we introduce in the next section.

4.2.9 Alternative XML binding extensions

Although JAXB is a standard binding solution for Java, it comes with the drawbacks discussed, leaving some place for alternative solutions. In the Restlet Framework, we decided to support three alternatives: EMF, JiBX, and XStream. It would be easy to support others if necessary.

The first extension is for Eclipse Modeling Framework (EMF), an open source library-facilitating, model-driven development with code generation and tooling within Eclipse IDE. You can use EMF to define representation structures and generate the matching representation beans in Java for use with Restlet-annotated interfaces. The generated classes extend base EMF classes by default, which doesn't fit very well with other XML serialization mechanisms supported by the Restlet Framework.

But EMF comes with its own serialization capabilities. By default, EMF lets you serialize any EMF object as XML Metadata Interchange (XMI), a generic XML language originating from UML to exchange diagrams. Plus, if your EMF model was created by importing an XML Schema, EMF is capable of producing and consuming regular XML representations, conforming to the original schema.

Logically the `org.restlet.ext.emf` extension uses this capability with an `EmfRepresentation` class and a related converter helper. `EmfRepresentation` can produce simple HTML representations, even distinguishing property values corresponding to hyperlinks by adding special EMF eAnnotations to your model. Those annotations need to be defined in the `www.restlet.org/schemas/2011/emf/html` namespace and then contain an entry with the linked name and a `true` value.

The second extension is for JiBX, an open source library providing XML binding. JiBX uses byte code enhancement instead of Java reflection and a special pull parser to obtain the best parsing and generation performance. It also offers a rich XML language to define the mapping between the XML structure and the Java structure. To define the binding, it provides tools to start from either code or an XML schema. For

more information read the Javadocs of the `org.restlet.ext.jibx` package and the JiBX project's documentation itself.

The third extension is for XStream, another open source library providing XML binding. This library has a focus on ease of use—for example, removing the need for mapping information in many cases. It is also a nonintrusive solution that can work with Java classes over which you have no control, which can prove very convenient. For more information read the Javadocs of the `org.restlet.ext.xstream` package and the XStream project's documentation itself.

It's time to wrap up our coverage of XML representations, which took you from the low-level SAX and DOM APIs to higher-level JAXB and XStream binding libraries. Although XML is an excellent format for most REST representations, it isn't the most compact and can affect performance of intensive AJAX applications. That's why an alternative JSON format has emerged, with a lighter syntax and fewer features. In the next section you'll discover that the Restlet Framework provides an equivalent support for JSON, with the exception of validation, transformation, and XPath-like selection, which aren't available yet for JSON.

4.3 *Producing and consuming JSON representations*

JavaScript Object Notation (JSON) is a lightweight data format that's typically used for exchanges between AJAX clients and web services. You can use JSON in other situations, but keep in mind that it's simpler than XML. For example, there's no standard like XPath to select content from JSON content, nor is there one like W3C XML Schema to validate a JSON document.

The Restlet Framework has extensive support for JSON, thanks to three dedicated extensions. In this section we continue to illustrate Restlet features using our example Homer mail resource—except this time exposing and consuming a JSON representation, as illustrated in figure 4.17.

For reference, the JSON mail document that we want to expose and consume as a representation of the Homer mail resource is in the following listing.

Listing 4.15 The target JSON mail representation

```
{  
    "content": "Doh!",  
    "status": "received",  
    "subject": "Message to self",  
    "accountRef": "http://localhost:8111/accounts/chunkylover53/"  
}
```

The first way to build and consume representations as JSON documents is an equivalent of the DOM API for JSON. We then introduce two convenient Restlet extensions (`org.restlet.ext.xstream` and `org.restlet.ext.jackson`) supporting the automatic binding between JSON representations and regular Java objects.



Figure 4.17 Example resource exposing a JSON document as representation

4.3.1 Using the JSON.org extension

The www.json.org website is the official home of the JSON language and provides a Java library to manipulate JSON documents that lives in `org.json.*` packages. It's similar to the DOM API for XML documents in the sense that it stores all the JSON data in memory when parsing, but this is rarely an issue because JSON documents are not generally intended to be very large due to AJAX constraints in browsers.

Let's rewrite the `MailServerResource` to exchange JSON representations. Listing 4.16 contains the new code, using the `org.json.JSONObject` class to store a map of properties. You can then use the `JsonRepresentation` from the `org.restlet.ext.json` package to wrap this JSON object and provide the proper metadata. By default the `application/json` media type is used.

Listing 4.16 Mail server resource using the JSON.org API

```

@Get
public Representation toJson() throws JSONException {
    JSONObject mailEl = new JSONObject();
    mailEl.put("status", "received");
    mailEl.put("subject", "Message to self");
    mailEl.put("content", "Doh!");
    mailEl.put("accountRef", new Reference(getReference(), "..."))
        .getTargetRef().toString());
    return new JsonRepresentation(mailEl);
}

@Put
public void store(JsonRepresentation mailRep) throws JSONException {
    JSONObject mailEl = mailRep.getJSONObject();
    System.out.println("Status: " + mailEl.getString("status"));
    System.out.println("Subject: " + mailEl.getString("subject"));
    System.out.println("Content: " + mailEl.getString("content"));
    System.out.println("Account URI: " + mailEl.getString("accountRef"));
}
  
```

Create JSON object structure similar to a map

Parse JSON representation to get mail properties

Output JSON element values

If you GET the mail resource in your browser, you should obtain the same document as in listing 4.15, except the order of the properties might vary. This is normal because JSON objects are unordered structures. The `JsonRepresentation` class illustrated in figure 4.18 is rather simple and works like the other extensions in this chapter by wrapping either a JSON value or a JSON representation, providing various methods to parse or consume this value.

As with XML representations, it's often more convenient to use a JSON binding library to marshal and unmarshal JSON representations. The Restlet Framework has two extensions to achieve this: XStream and Jackson. The XStream extension—a powerful XML binding solution introduced in the previous section—provides an interesting bridge to JSON representations by using the Jettison JSON library. But the JSON representations that it produces tend to be verbose, so we generally recommend using the more specialized Jackson extension for JSON instead.

4.3.2 Using the Jackson extension

Jackson is an open source library that was quickly embraced by various frameworks due to its features and outstanding performance. It's a comprehensive solution providing both a DOM-like API and a JSON binding mechanism.

In this section we illustrate the `org.restlet.ext.jackson` extension and its main `JacksonRepresentation` class, described in figure 4.19. It has three properties: the `object` property contains the wrapped Java object that was parsed or can be formatted; the `objectClass` indicates the target class of the object to parse; and `objectMapper` offers a way to customize the mapping. There are three constructors, two for serialization and one for deserialization, and a `createObjectMapper()` method that can be overridden to customize Jackson's behavior.

Even though the class looks simple, it can produce very nice results, as illustrated in listing 4.17. We manually wrote a `Mail` class, containing our four `String` properties with getters and setters, and then in the GET handling method let Jackson produce the correct JSON representation. Handling of the PUT methods and parsing of the JSON representation received into a `Mail` instance is equally simple and very similar to the way the JAXB extension is working.

Listing 4.17 Mail server resource using the Jackson extension

```
@Get
public Representation toJson() {
```

<<WriterRepresentation>>	
JsonRepresentation	
indenting : boolean	
indentingSize : int	
JsonRepresentation(JSONArray)	
JsonRepresentation(JSONObject)	
JsonRepresentation(JSONStringer)	
JsonRepresentation(JSONTokener)	
JsonRepresentation(Map<Object, Object>)	
JsonRepresentation(Object)	
JsonRepresentation(Representation)	
JsonRepresentation(String)	
getJSONArray() : JSONArray	
getJSONObject() : JSONObject	
getJsonTokener() : JSONTokener	

Figure 4.18 Properties and methods of the `JsonRepresentation`

<< WriterRepresentation >>	
JacksonRepresentation <T>	
object : T	
objectClass : Class<T>	
objectMapper : ObjectMapper	
JacksonRepresentation(MediaType, T)	
JacksonRepresentation(Representation, Class<T>)	
JacksonRepresentation(T)	
createObjectMapper() : ObjectMapper	

Figure 4.19 Properties and methods of the `JacksonRepresentation`

```

Mail mail = new Mail();
mail.setStatus("received");
mail.setSubject("Message to self");
mail.setContent("Doh!");
mail.setAccountRef(new Reference(getReference(), "...").getTargetRef()
    .toString());

return new JacksonRepresentation<Mail>(mail);
}

@Put
public void store(Representation rep) throws IOException {
    JacksonRepresentation<Mail> mailRep = new JacksonRepresentation<Mail>(
        rep, Mail.class);
    Mail mail = mailRep.getObject();

    System.out.println("Status: " + mail.getStatus());
    System.out.println("Subject: " + mail.getSubject());
    System.out.println("Content: " + mail.getContent());
    System.out.println("Account URI: " + mail.getAccountRef());
}

```

Output JSON element values

```

graph TD
    A["Create mail bean"] --> B["Wrap bean with Jackson representation"]
    B --> C["Parse JSON representation to get mail bean"]
    C --> D["Output JSON element values"]

```

Not surprisingly the `MailClient` class, with logic unchanged since the beginning of the chapter, works equally well. When you consider how close this example is to the JAXB one, it's tempting to ask whether it would be possible to unify both approaches to produce XML or JSON at the same time. In fact this is indeed possible, and we cover that in section 4.5 when introducing the converter service—but there are a few steps to climb before that.

Template representations are useful in producing HTML representations in a way comparable to JSP technology.

4.4 Applying template representations

So far in this chapter, you've exchanged data-oriented representations using the XML and JSON languages, but we've barely explained how to produce HTML representations to be displayed in a web browser. You need a solution to realize the promise (made in chapter 1) of web applications unifying web services and websites.

Because Restlet applications can be deployed standalone, outside Servlet containers, we can't rely on the standard Java Server Pages (JSP) technology. But that's not an issue because we have powerful alternatives: dynamic representations powered by the FreeMarker and Apache Velocity template engines. We call them template representations because they're built around the skeleton of the target document, filled with instructions to insert dynamic values provided by a data model.

Figure 4.20 shows a new variant of our Homer mail resource, a dynamically built HTML document. Keep in mind that template representations can be successfully used to produce any kind of textual representations, such as XML representations or even Java code. HTML is a particularly appropriate target language.

For reference, the following listing shows the HTML mail document that we want the Homer mail resource to expose.

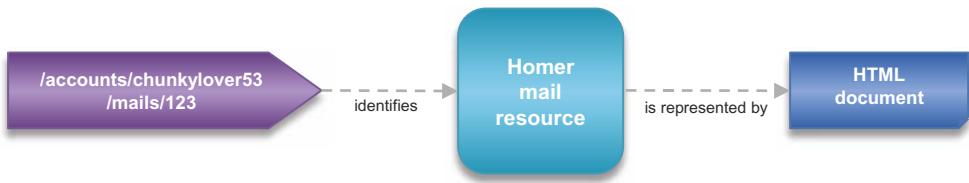


Figure 4.20 Example resource exposing an HTML document as representation

Listing 4.18 The target HTML mail representation

```

<html>
<head>
    <title>Example mail</title>
</head>
<body>
<table>
    <tbody>
        <tr>
            <td>Status</td>
            <td>received</td>
        </tr>
        <tr>
            <td>Subject</td>
            <td>Message to self</td>
        </tr>
        <tr>
            <td>Content</td>
            <td>Doh!</td>
        </tr>
        <tr>
            <td>Account</td>
            <td><a href="http://www.rmep.org/accounts/chunkylover53/">Link</a></td>
        </tr>
    </tbody>
</table>
</body>
</html>
  
```

In the next subsection we put those ideas into practice by using FreeMarker to dynamically generate the preceding HTML representation.

4.4.1 Using the FreeMarker extension

FreeMarker is a powerful open source template engine with a comprehensive syntax allowing simple value insertion, loop control, conditions, custom macros, and directives. We recommend it as an alternative to JSP to produce HTML or other textual representations.

The template corresponding to the example HTML mail in the next listing takes the same structure as the target HTML representation and uses \${variableName} to

insert the content provided with an object data model. This data model can be hierarchical or even be an XML document. (The complete syntax of FreeMarker is explained at www.freemarker.org.)

Listing 4.19 The FreeMarker template HTML representation

```
<html>
<head>
    <title>Example mail</title>
</head>
<body>
<table>
    <tbody>
        <tr>
            <td>Status</td>
            <td>${mail.status}</td>
        </tr>
        <tr>
            <td>Subject</td>
            <td>${mail.subject}</td>
        </tr>
        <tr>
            <td>Content</td>
            <td>${mail.content}</td>
        </tr>
        <tr>
            <td>Account</td>
            <td><a href="${mail.accountRef}">Link</a></td>
        </tr>
    </tbody>
</table>
</body>
</html>
```

With the template file ready, we can now use the FreeMarker extension for Restlet, located in `org.restlet.ext.freemarker`, to produce the target representation. This extension contains the `TemplateRepresentation` class, which works by wrapping a template object and a data model.

Various constructors are also available, providing different ways to create or obtain the template. Either you provide it as a Restlet representation, as in listing 4.20, or you indicate only a template name.

In the latter case FreeMarker will load the template from a local directory that you indicate with a `Configuration` instance, which can be reused for several `TemplateRepresentation` instances. This approach is more scalable because FreeMarker can use its caching strategies to accelerate the resolution of your templates.

For the data model you can provide Java beans or collections such as maps and lists. There's also a `setDataModel(Request, Response)` method able to wrap a Restlet call to allow access to request and response values using shortcut variable names (see the `org.restlet.util.Resolver` interface for details on these shortcuts).

Listing 4.20 Mail server resource using the FreeMarker extension

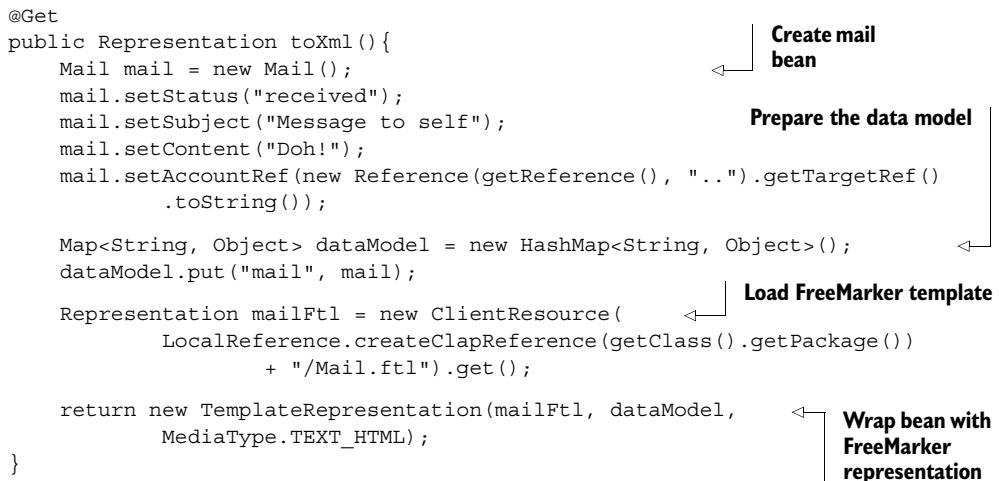
```

@Get
public Representation toXml() {
    Mail mail = new Mail();
    mail.setStatus("received");
    mail.setSubject("Message to self");
    mail.setContent("Doh!");
    mail.setAccountRef(new Reference(getReference(), "...").getTargetRef()
        .toString());

    Map<String, Object> dataModel = new HashMap<String, Object>();
    dataModel.put("mail", mail);
    Representation mailFtl = new ClientResource(
        LocalReference.createClapReference(getClass().getPackage()
            + "/Mail.ftl").get());

    return new TemplateRepresentation(mailFtl, dataModel,
        MediaType.TEXT_HTML);
}

```



The code implements a GET method to return an XML representation. It creates a new Mail object, sets its status to "received", subject to "Message to self", and content to "Doh!". It then creates a data model map containing the mail object and returns a TemplateRepresentation using a ClientResource for the FTL template "Mail.ftl".

In listing 4.20 we only implemented the GET method because it's rarely useful to consume HTML representations. We cover the processing of HTML form posts in section 7.1.1. Our MailClient was also slightly modified to write the HTML representation on the console. Launch the application again and go to the `http://localhost:8111/accounts/chunkylover53-mails/123` URI and see the HTML page produced.

In addition to this FreeMarker extension, a similar extension is provided for the Apache Velocity template engine.

4.4.2 Using the Velocity extension

Velocity is another popular open source template engine provided by the Apache Foundation, similar to FreeMarker but with a slightly different syntax and set of features. Listing 4.21 shows the template corresponding to our example HTML mail. It takes the same structure as the target HTML representation and uses the \$variableName to insert the content provided with an object data model. This data model can be hierarchical or even an XML document. (For more information, the complete syntax of Velocity is explained at <http://velocity.apache.org>.)

Listing 4.21 The Velocity template HTML representation

```

<html>
<head>
<title>Example mail</title>
</head>
<body>
<table>
    <tbody>
        <tr>
            <td>Status</td>

```

```

        <td>$mail.status</td>
    </tr>
    <tr>
        <td>Subject</td>
        <td>$mail.subject</td>
    </tr>
    <tr>
        <td>Content</td>
        <td>$mail.content</td>
    </tr>
    <tr>
        <td>Account</td>
        <td><a href="$mail.accountRef">Link</a></td>
    </tr>
</tbody>
</table>
</body>
</html>
```

With the template file ready, you can now use the Velocity extension for Restlet, located at `org.restlet.ext.velocity`, to produce the target representation. This extension contains the `TemplateRepresentation`, which works exactly like the FreeMarker extension, as shown in the following listing.

Listing 4.22 Mail server resource using the Velocity extension

```

@Get
public Representation toXml() throws Exception {
    Mail mail = new Mail();                                Create mail
    mail.setStatus("received");
    mail.setSubject("Message to self");
    mail.setContent("Doh!");
    mail.setAccountRef(new Reference(getReference(), "...").getTargetRef()
        .toString());                                     Prepare the data model

    Map<String, Object> dataModel = new HashMap<String, Object>();
    dataModel.put("mail", mail);                           Load Velocity template

    Representation mailVtl = new ClientResource(          ←
        LocalReference.createClapReference(getClass().getPackage()
            + "/Mail.vtl").get();                         Wrap bean
    return new TemplateRepresentation(mailVtl, dataModel,   with Velocity
        MediaType.TEXT_HTML);                            representation
}
```

At this point, you've learned various techniques to exchange XML, JSON, and HTML representations for the example Homer mail resource, but each time you had to develop a separate Restlet application, with a new `MailServerResource` class. This isn't exactly the idea of unified web applications proposed in chapter 1. The key feature missing to realize our goal is the topic of the next section: HTTP content negotiation.

4.5 Content negotiation

The modern web supports many kinds of clients, each with different capacities and expectations: classic desktop web browsers, mobile web browsers, native mobile applications, programmatic (not human-driven) clients like search bots, and so on. Wouldn't it be nice to avoid building a different version of your application for each client? The good news is that HTTP has a content negotiation feature (frequently shortened to conneg) that can help you unify your developments. Let's now look at how HTTP conneg works before learning how to apply it within the Restlet Framework.

4.5.1 Introducing HTTP content negotiation

HTTP allows any resource to have multiple representations, called variants, and specify how a client and a server can determine which variant is the most appropriate for a given interaction. You might prefer to use HTML when exchanging representations with a web browser and XML or JSON with a programmatic client wanting structured data that's easy to manipulate.

Let's illustrate those concepts with our RESTful mail example. In figure 4.21 we reuse the type of diagrams introduced in figure 1.4 when explaining the relationships between resources, identifiers (URIs), and representations. We also apply it to our Homer mail resource described in appendix D and used throughout this chapter.

What the figure says is that the Homer mail resource is identified by a single URI, /accounts/chunkylover53-mails/123, and exposes three variants, each corresponding to a different type of content:

- HTML variant for retrieval by web browsers
- XML variant for consumption by programmatic clients
- JSON variant typically for retrieval by AJAX applications

HTTP uses a system where possible types of content are described by media types. A *media type* has a name and defines how conforming content should be interpreted. A

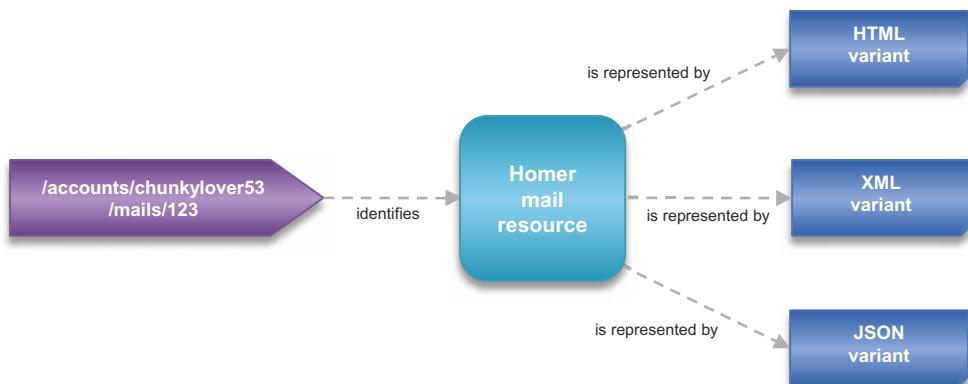


Figure 4.21 Account resource identified by a URI and represented by three variants

number of standard media types have been defined (see table 4.4 for examples), and you can define your own if needed. The Restlet API naturally supports media types with the `org.restlet.data.MediaType` class, which declares many constants as well. Note that you can always create your own instances if no constant matches your needs.

Table 4.4 A few examples of common standard media types

Name	Description
text/plain	Some plain text
text/html	An HTML document
application/xml	An XML document
application/json	A JSON document
image/jpeg	An image in JPEG format
image/png	An image in PNG format
application/pdf	A PDF document

When issuing an HTTP request, a client can include an *Accept* header that lists the acceptable media types for the representation that the server could send back in the response. It's even possible to specify the degree of preference for each acceptable media type, using the *q* parameter to associate weights to media types.

The server can then use this information to generate the best representation possible for that particular client. The following header specifies that the client expects either an XML or a JSON representation, with a preference for JSON:

```
Accept: application/json; q=1.0, application/xml; q=0.5
```

In addition to the *Accept* header, HTTP defines other useful headers related to other dimensions of content negotiation:

- The *Accept-Language* header specifies preferred languages for the response (for example, English or French).
- The *Accept-Encoding* header specifies preferences for content encoding (for example, to indicate support for receiving gzip-compressed responses).
- The *Accept-Charset* header specifies supported character sets such as UTF-8.
- The *User-Agent* header can also be used to drive content generation. It identifies the client (for example, discriminating between Internet Explorer and Firefox) and lets you adapt your content to its specificities or limitations.

Content negotiation is a powerful feature of HTTP, and you should consider using it for your applications. In the next subsections we put it into practice with the Restlet Framework, first showing how resource variants can be declared and then configuring the client to retrieve the desired variants.

4.5.2 Declaring resource variants

Since version 2.0 of the Restlet Framework, there are two ways for a server resource to declare the variants that it exposes. The first is programmatic, using the modifiable variants property, and the second is based on Restlet annotations (discussed in subsection 4.5.4).

If you declare some variants, your ServerResource subclass needs to override the get(Variant) method instead of the get() one, or the put(Representation, Variant) instead of put(Representation). Note that content negotiation can be used for all methods, not just GET ones, but they only apply to the response entity.

Let's illustrate this feature with the same example, but this time trying to expose and consume the mail as XML or JSON representations depending on client preferences. For XML, you use XStream instead of JAXB because it can work with the same Mail class already used with Jackson for JSON representations in subsection 4.3.2. Note in listing 4.23 how you override the doInit() method to declare the supported variants.

Then you test the variant parameter of the get(Variant) method to know which variant is preferred by the client. The selection of the best variant matching the client preferences is done transparently by the Restlet Framework!

Listing 4.23 Mail server resource supporting XML and JSON representations

```

@Override
protected void doInit() throws ResourceException {
    getVariants().add(new Variant(MediaType.APPLICATION_XML));
    getVariants().add(new Variant(MediaType.APPLICATION_JSON));
}

@Override
protected Representation get(Variant variant) throws ResourceException {
    Representation result = null;
    Mail mail = new Mail();
    mail.setStatus("received");
    mail.setSubject("Message to self");
    mail.setContent("Doh!");
    mail.setAccountRef(new Reference(getReference(), "...").getTargetRef()
        .toString());
    if (MediaType.APPLICATION_XML.isCompatible(variant.getMediaType())) {
        result = new XstreamRepresentation<Mail>(mail);
    } else if (MediaType.APPLICATION_JSON.isCompatible(variant
        .getMediaType())) {
        result = new JacksonRepresentation<Mail>(mail);
    }
    return result;
}

@Override
protected Representation put(Representation representation, Variant variant)
    throws ResourceException {
    Mail mail = null;
}

```

The code listing includes several annotations and comments:

- A callout box labeled "Declare two variants supported" points to the line `getVariants().add(new Variant(MediaType.APPLICATION_XML));` and `getVariants().add(new Variant(MediaType.APPLICATION_JSON));`.
- A callout box labeled "Create mail bean" points to the declaration of the `Mail` object.
- A callout box labeled "Use this order to prevent NPEs" points to the sequence of setting properties: `setStatus`, `setSubject`, `setContent`, and `setAccountRef`.
- A callout box labeled "Wrap bean with XStream representation" points to the creation of an `XstreamRepresentation` for XML.
- A callout box labeled "Wrap bean with Jackson representation" points to the creation of a `JacksonRepresentation` for JSON.

```

if (MediaType.APPLICATION_XML.isCompatible(representation
    .getMediaType())) {
    mail = new XstreamRepresentation<Mail>(representation,
    Mail.class).getObject();
    System.out.println("XML representation received");
} else if (MediaType.APPLICATION_JSON.isCompatible(representation
    .getMediaType())) {
    mail = new JacksonRepresentation<Mail>(representation, Mail.class)
        .getObject();
    System.out.println("JSON representation received");
}
if (mail != null) {
    System.out.println("Status: " + mail.getStatus());
    System.out.println("Subject: " + mail.getSubject());
    System.out.println("Content: " + mail.getContent());
    System.out.println("Account URI: " + mail.getAccountRef());
    System.out.println();
}
return null;
}

```

In the `put(Representation, Variant)` method, we test the value of the representation's media type, not the variant's media type. The variant only defines the preferred response type. You could, for example, submit a representation in XML and receive HTML as a result. In the next subsection we continue the example on the client side by configuring the client preferences to retrieve a specific variant.

4.5.3 Configuring client preferences

You can define your preferences in different ways as a client when issuing requests so that a server can make the best guess regarding the variant you want to obtain via content negotiation.

As shown in subsection 4.5.1, the standard HTTP way is to set the `Accept*` headers. The Restlet API facilitates this, thanks to a set of `ClientInfo.accepted*` properties that are instances of `List<Preference<T extends Metadata>>`. Subclasses of `Metadata` are defined for all the dimensions of content negotiation: `CharacterSet`, `Encoding`, `Language`, and `MediaType` metadata classes, all from the `org.restlet.data` package.

The advantage of using those properties is that you can set fined-grained preferences, with various levels of qualities for each media type you accept. But in many cases you only want to retrieve a single specific media type. In this case it's easy to use helper methods directly on the `ClientResource` class, as illustrated in the next listing.

Listing 4.24 Mail client selecting XML, then JSON variants

```

public static void main(String[] args) throws Exception {
    ClientResource mailClient = new ClientResource(
        "http://localhost:8111/accounts/chunkylover53-mails/123");

```

```
Representation mailRepresentation = mailClient
    .get(MediaType.APPLICATION_XML);
mailClient.put(mailRepresentation);

mailRepresentation = mailClient.get(MediaType.APPLICATION_JSON);
mailClient.put(mailRepresentation);
}
```

In this example you issue two GET calls, the first requesting the XML variant and the second the JSON variant. If you run your server application and then this mail client, the server displays the following output on its console:

```
XML representation received
Status: received
Subject: Message to self
Content: Doh!
Account URI: http://localhost:8111/accounts/chunkylover53/

JSON representation received
Status: received
Subject: Message to self
Content: Doh!
Account URI: http://localhost:8111/accounts/chunkylover53/
```

In addition, since version 2.1 the `ClientResource` class provides the convenient shortcut `accept (Metadata...)` method that lets you add new preferences with a 1.0 quality. Another way to express your preferences is to rely on the `TunnelService` (briefly presented in section 2.3.4). For example you can enter the following URI in your browser to retrieve the JSON representation:

```
http://localhost:8111/accounts/chunkylover53-mails/123?media=json
```

The `json` value passed in the query parameter at the end of the URI must correspond to one of the declared prefixes in the `MetadataService`, which defines default extension names for common media types. To end this chapter we combine the power of `conneg` with the converter service that we also encountered in chapter 2, resulting in even simpler code.

4.5.4 Combining annotated interfaces and the converter service

In chapter 2 we used the converter service and annotated Java interfaces to simplify the implementation of example account resources exchanging simple strings as representations.

What we'd like now is to use the same mechanism with more complex structures, like our `Mail` bean class. In figure 4.22 we enhance our previous one to introduce the representation bean (the `Mail` class in our case) and its relationship with the resource's state and the variants that can be serialized from it.

We start by defining the annotated `MailResource` interface in the following listing that will be implemented on the server side and used as a proxy on the client side.

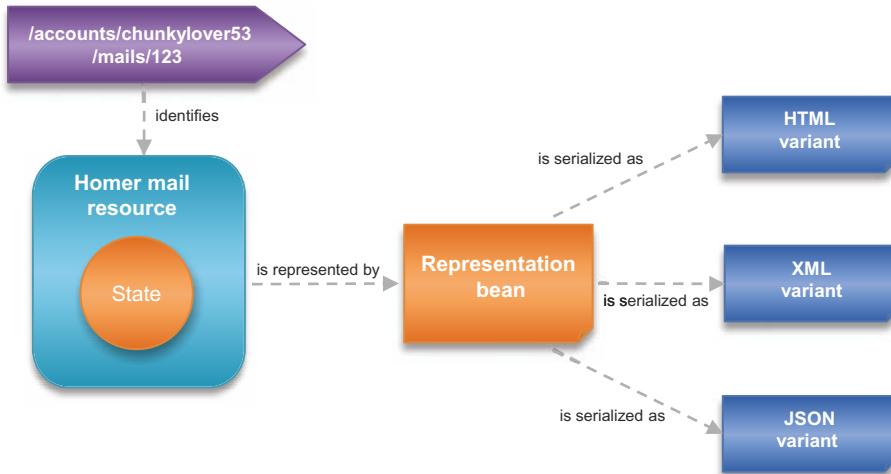


Figure 4.22 Account resource represented by bean serialized in three variants

Listing 4.25 Annotated MailResource interface

```
public interface MailResource {
    @Get
    public Mail retrieve();
    @Put
    public void store(Mail mail);
}
```

As mentioned in chapter 1, the `@Get` and `@Put` annotations come from the `org.restlet.resource` package and define the binding between Java and HTTP methods. Let's implement the server side (see the next listing). Again, the code is extremely straightforward. We abstracted away many Restlet concepts, but note that we're still extending the `ServerResource` class, giving us access to the whole call context, such as the resource reference used in the `accountRef` property.

Listing 4.26 Mail server resource implementing the MailResource interface

```
public class MailServerResource extends ServerResource
implements MailResource {
    public Mail retrieve(){
        Mail mail = new Mail();
        mail.setStatus("received");
        mail.setSubject("Message to self");
        mail.setContent("Doh!");
        mail.setAccountRef(new Reference(getReference(), "...").
getTargetRef().toString());
        return mail;
    }
}
```

```

public void store(Mail mail) {
    System.out.println("Status: " + mail.getStatus());
    System.out.println("Subject: " + mail.getSubject());
    System.out.println("Content: " + mail.getContent());
    System.out.println("Account URI: " + mail.getAccountRef());
    System.out.println();
}
}

```

The nice thing about this code is that all the serialization and deserialization is done transparently by the Restlet Framework. In order to work as expected, it's essential to properly configure your classpath to have the `org.restlet.ext.jackson` extension first, covering JSON serialization, then the `org.restlet.ext.xstream` extension, covering XML serialization. This is because the XStream also has the ability to convert to JSON, and not only to XML, but is less powerful than Jackson for this purpose.

At this point you can already launch the mail server application and try to get the mail resource from your browser. Adjusting the media query parameter should let you obtain either JSON or XML representation produced by Jackson or XStream. What's remarkable is that we obtained transparent serialization of the Mail beans as well as the ability to negotiate their format using standard HTTP con-neg. You could potentially support additional formats by adding new extensions with the proper converter to your classpath! The following listing shows how the MailClient looks.

Listing 4.27 Mail server resource implementing the MailResource interface

```

MailResource clientResource = ClientResource.create(
    "http://localhost:8111/accounts/chunkylover53-mails/123",
    MailResource.class);
clientResource.store(clientResource.retrieve());           ← Create the dynamic client proxy

```

← Directly invoke interface methods

If you need access to the `ClientResource` instance backing the dynamic proxy created, you can make your interface extend the `org.restlet.resource.ClientProxy` to have access to an `getClientResource()` method automatically implemented.

You can now run the client to confirm that the server works the same as for alternative approaches presented earlier in the chapter. We dramatically reduced the number of lines of code while keeping the full power of the Restlet API in hand if needed.

4.6 Summary

In this chapter you explored in depth how representations are supported by the Restlet Framework, from the low level `Variant`, `RepresentationInfo`, and `Representation` classes to the higher level `JacksonRepresentation` or `XstreamRepresentation` classes, providing an automatic binding between HTTP entities and Java classes.

You learned how to manipulate XML representations with powerful features such as XPath selections, XML schema validation, or XSLT transformation. You also saw two

ways to deal with JSON representations and used template representations with engines such as FreeMarker to produce HTML representations dynamically.

Then you learned about the power of HTTP content negotiation and two main ways to support it in Restlet, through explicit declaration or transparently with the converter service and annotated Java interfaces. In the last case you obtained a remarkable reduction in terms of number of lines of source code.

Abstracting away the parsing and formatting logic makes you dependent on binding libraries such as JAXB, XStream, and Jackson for properly handling the evolution of your representation beans. For simple evolutions like the addition or removal of properties, you should be pretty safe, but you need to be aware of the impact of deeper structural changes.

In the end the various abstraction mechanisms provided are very attractive due to the huge productivity gains, but they couple your client and server code (while still retaining the ability to use any HTTP client using standard content negotiation). For most applications this is an acceptable trade-off, although others will need to keep a finer control on those aspects by using the more explicit approach that we've seen before.

With the Restlet Framework you're free to choose the approach that best fits your requirements, even keeping the ability to mix both approaches if needed. In the next chapter we discuss a totally different, but nonetheless very important, subject: how to secure your Restlet applications.



Securing a Restlet application

This chapter covers

- Securing the communication
- Authenticating the remote user
- Assigning roles to the authenticated user
- Authorizing the user to perform actions on the system
- Ensuring end-to-end integrity of the data

In chapter 3 you saw how to deploy a Restlet application on premises, but this application was freely accessible to any client. In the real world free accessibility is rarely desirable, and a point comes when you need to take security into account. This chapter covers how to secure a Restlet application. We address the issue of securing the communication between the client and the server by using transport level security that can ensure confidentiality and integrity of the exchange.

Then we go through three related notions: authentication, assignment of roles, and authorization. The section on authentication guides you through verifying the identity of the remote user. The section on role assignments shows how remote users may be mapped into the system's overall identity management structures, specifically via roles in the application. The section on authorization shows how to

grant or deny users permission to perform an action, depending on their authentication status and on the action they wish to perform.

Finally we show how to protect against accidental modification during the exchange of representations using the mechanisms that ensure end-to-end integrity of the data. We begin by explaining how to implement transport confidentiality and integrity with Restlet.

5.1 **Ensuring transport confidentiality and integrity**

The most common way of securing communication over the internet is to use the Transport Level Security (TLS) protocol, or the Secure Socket Layer (SSL) protocol from which TLS is derived. TLS can be used to ensure the confidentiality and integrity of the communication between a client and a server, making the communication immune to eavesdropping or alteration by a third party or through network failure.

This section covers TLS and SSL and describes how to configure a Restlet component to use them. First we explain how the TLS protocol works, particularly in the context of HTTP, and the purpose of certificates. Then we describe what keystores are, how to generate private and public keys, and how to handle certificates in Java. Finally we explain how Restlet is configured to use certificates and how to configure custom SSL contexts for specialized applications.

5.1.1 **Understanding TLS and SSL**

The purpose of TLS is to ensure the security of the communications between a client and a server. It provides applications with secure sockets, which have been designed to match normal sockets as closely as possible. Correct configuration of TLS is necessary in order to ensure the confidentiality and integrity of the communication.

TLS is the Internet Engineering Task Force's (IETF) standard for a transport layer security protocol. TLS v1.0 is based on SSL v3. As a result TLS is still often referred to as SSL in a number of frameworks and applications. For the purpose of this book, TLS and SSL can be considered to be the same except when an explicit version is given. In Java, most of the classes related to the configuration of TLS/SSL use the SSL prefix.

TLS normally relies on Public Key Cryptography, which uses a pair of keys to encrypt and decrypt messages. The pair of keys consists of a private key, known only to the person or machine by which it was generated, and an associated public key, which may be distributed publicly. Two main operations can be performed: encryption and signing. Encryption is used to make the content of a message secret and is done using the public key; decryption can then only be performed using the corresponding private key. Signing is used to assert the authenticity of a message and involves similar operations, but is done using the private key; validating the signature is done using the corresponding public key. Here the action of signing means using a private key to encrypt the result of applying a hash function to the content of the message being sent.

A fundamental requirement to avoid man-in-the-middle attacks is to enable the client to verify the identity of the server with which it's communicating. Without this

verification, the client could be talking to an impostor relaying, eavesdropping, and possibly altering the messages exchanged with the legitimate server. The mechanism to verify the identity of a remote server relies on certificates (usually based on the X.509 standard), which bind a public key to an identity and are distributed as part of a Public Key Infrastructure (PKI). For this reason, the server must be configured to use a certificate.

A certificate is a signed statement that includes a public key and other information such as date of validity, Subject Distinguished Name (Subject DN), and the Issuer Distinguished Name (Issuer DN). The Subject is the entity to which the certificate is issued; it's the entity that has the private key associated with the public key in the certificate. The Issuer is the entity that asserts that the information in the certificate, in particular the identity of the subject, is correct. The issuer is usually a Certification Authority (CA), which may be a commercial company or may be a local CA created within your institution or company. PKIs describe the relationships and trust models between the CAs and are associated with legal documents describing the intended use of various X.509 attributes (depending on CA policies).

The verification process in a PKI relies on the certificate consumer to be configured with a set of certificates it trusts *a priori*: the trusted anchors. Verifying a certificate then consists of building a chain between that certificate and one of the trusted anchors; there may be intermediate certificates in the chain.

Alternative trust models exist, such as FOAF+SSL, that may require more specialized SSL settings. Certificates can also be self-signed, in which case the trust model has to be established by some other means (for example, someone you trust gives you this certificate in person). Figure 5.1 illustrates the HTTPS connection process.

When a user agent connects to an HTTPS server, it first verifies the certificate of the server during the TLS handshake (before any HTTP data is exchanged). Web browsers are generally bundled with a number of CA certificates (the trusted anchors), often from commercial or governmental CAs. The Oracle Java SE also comes with a set of trust anchors (the default truststore in Java's terminology), but the reference guide for the Java Secure Socket Extension (JSSE), which is the part of the Java SE responsible for handling TLS, recommends checking the content of that truststore when deploying applications.

Once the certificate has been verified against a set of trusted anchors, the client must also verify that the certificate matches the name of the host it intends to connect to. To be acceptable, the host name must be in the Subject Alternative Name DNS entry (an X.509 extension) or, if this extension is absent, in the Common Name (CN) field of the certificate's Subject DN.

Only once these verifications have been performed may the TLS handshake proceed, including the exchange of random session keys. Upon completion of the handshake, the HTTP communication may begin on top of the TLS layer. TLS protects the HTTP communication from eavesdropping, using encryption, and from third-party alteration, because such attacks would make the communication terminate abruptly (which would cause an exception in Java).

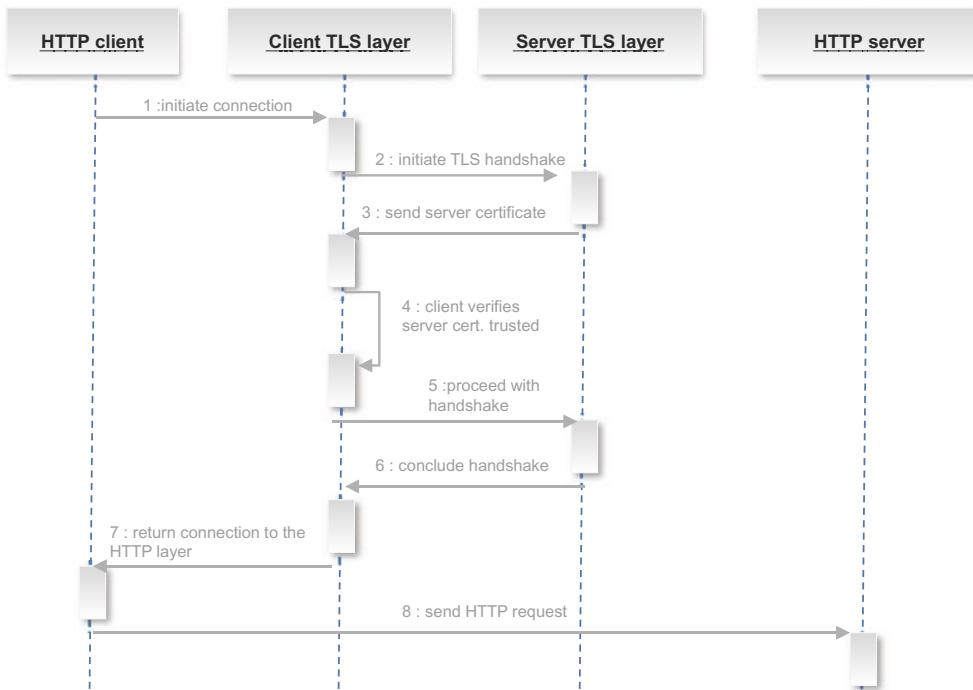


Figure 5.1 HTTPS sequence diagram with TLS/SSL

In addition, clients may also send a certificate to the server during the TLS handshake, when the server wishes to use client-certificate authentication. The verification process on the server side is similar except that there is no requirement to check the host name.

5.1.2 Storing keys and certificates

In general, special files are used to store certificates and private keys. For TLS two kinds of files are used, keystores and truststores, which can also be used in Java code via the `java.security.KeyStore` class. The names may appear confusing, but both keystores and truststores are instances of `KeyStore`.

In this context, the *keystore* is the store that has the information local to the application; the keystore contains the server certificate and its private key on a server and the client certificate and its private key on a client.

In contrast, the *truststore* is the store used for making trust decisions regarding certificates presented by remote peers. The truststore is used by the client to verify the certificates of the servers it connects to and by the server to verify the client certificates it receives (if the server is configured to ask for client certificates).

The Oracle JRE is provided with a default truststore that contains a number of commercial CA certificates (similar to browsers). The Oracle Java SE supports a number of types of keystores. The default type is JKS, but PKCS12 (PKCS#12 format) is also

supported and can be used for importing and exporting .p12 files with a number of tools such as web browsers or OpenSSL. Java provides a command-line tool to manipulate keystores called keytool. Other third-party tools exist.

The following subsections give an overview of the keytool operations used to create and manage certificates for use with Restlet.

5.1.3 Generating a self-signed certificate

This section describes how to generate a key pair using keytool, which also creates a self-signed certificate implicitly. Note that this tool has been greatly enhanced in Java 5.

The following command creates a 2048-bit RSA key pair and creates a self-signed certificate for CN=simpson.org,OU=Simpson family,O=The Simpsons,C=US where CN is the common name (fully qualified domain name or IP address), OU the organizational unit, O the organization, and C the country code, using an SHA1 signature digest. Then it stores it into the server alias of a keystore file called serverKey.jks:

```
keytool -keystore serverKey.jks -alias server -genkey -keyalg RSA  
-keysize 2048 -dname  
"CN=simpson.org,OU=Simpson family,O=The Simpsons,C=US"  
-sigalg "SHA1withRSA"
```

Note that you'll be prompted for passwords for the keystore and the key itself. Let's enter password as the example value. This certificate can then be exported as an independent certificate file server.crt, using this command (providing the same password):

```
keytool -exportcert -keystore serverKey.jks  
-alias server -file serverKey.crt
```

The certificate file can then be distributed and imported explicitly in browsers, whereas the private key remains in the keystore file. By design, private keys can't be recovered from public keys or certificates, so backing up the keystore at this stage is recommended. In the following subsection you see how to have this key material certified by a certification authority (CA).

5.1.4 Generating a certificate request

This section describes how to generate a certificate signing request (CSR), which may be required by the CA that will provide your certificate. (Alternatively, some CAs have web interfaces that allow for the key material generation within the browser.)

A CSR binds a public key to a requested identity and attributes and is as such similar to a certificate. But it can't be used as a certificate. CAs may choose to emit certificates that have different attributes or DN structures than those that were requested, depending on their policies. A CSR is signed by the private key corresponding to its public key, therefore proving that whoever generated the CSR owns this private key.

A prerequisite for the generation of a CSR is a key pair, generated as described in the previous section. Then the CSR can be created with this command:

```
keytool -certreq -keystore serverKey.jks  
-alias server -file serverKey.csr
```

Alternatively, tools such as OpenSSL can be used to generate certificates, certificate requests, and CAs. OpenSSL users may want to start with the man-page for CA.pl.

Next you must import the resulting certificate created by the CA into the keystore to be able to use it.

5.1.5 Importing a trusted certificate

After approval of the certificate request, the CA will provide a certificate file, usually in PEM or DER format. It needs to be imported back into the keystore to be used as a server certificate:

```
keytool -import -keystore serverKey.jks
        -alias server -file serverKey.crt
```

This command is also used for importing CA certificates into a special keystore that's going to be used as a truststore—on the client side, for example. In this case the -trustcacerts options may also be required:

```
keytool -import -keystore clientTrust.jks -trustcacerts
        -alias server -file serverKey.crt
```

This trusted certificate may also be imported explicitly into your browser or used by a programmatic HTTPS client. This is useful if you're deploying your own infrastructure, or during development phases.

5.1.6 Enabling HTTPS in Restlet

Enabling HTTPS on a Restlet server is only relevant to standalone Restlet server connectors. If your Restlet application is running within a Servlet container, the container's connectors must be configured according to its documentation.

Configuration regarding HTTPS can be set using the parameters of the server's context. Table 5.1 lists all useable parameters in this context.

Table 5.1 Parameters of the server context related to the use of HTTPS

Parameter name	Description
keystorePath	Specifies the path for the keystore used by the server
keystorePassword	Specifies the password for the keystore containing several keys
keystoreType	Specifies the type of the keystore
keyPassword	Specifies the password of the specific key used
truststorePath	Specifies the path to the truststore
truststorePassword	Specifies the password of the truststore
truststoreType	Specifies the type of the truststore
sslContextFactory	Specifies a custom SslContextFactory implementation
needClientAuthentication	Indicates whether to require client certificate authentication
wantClientAuthentication	Indicates whether you want client certificate authentication

Next we cover a basic configuration of the keystore for an HTTPS-based server. The use of the `sslContextFactory` parameter is described in subsection 5.1.7.

The code in the following listing illustrates how to set up an HTTPS server using a certificate stored in `serverKey.jks`, as described in the previous sections.

Listing 5.1 Basic configuration of the keystore on a server

```
Component mailServer = new Component();
Server server = mailServer.getServers().add(Protocol.HTTPS, 8183);
Series<Parameter> parameters = server.getContext().getParameters();
parameters.add("keystorePath",
    "src/org/restlet/example/book/restlet/ch05/serverKey.jks");
parameters.add("keystorePassword", "password");
parameters.add("keystoreType", "JKS");
parameters.add("keyPassword", "password");

mailServer.getDefaultHost().attach(new MailServerApplication());
mailServer.start();
```

After having configured and added the HTTPS server to the component ①, you need to set the parameters related to HTTPS in order to configure the associated keystore ②. They correspond to its path, password, and type. Another password also needs to be set for the associated key.

NOTE In order to run this example, you need to ensure that you correctly generated the SSL keystore in the previous section. If you're using version 2.0 of the framework, you also need to add HTTPS client and server connectors to your classpath, such as `org.restlet.ext.net.jar` and `org.restlet.ext.jetty.jar`, and its dependencies.

You should then be able to start your server and point your browser to `https://localhost:8183/accounts/chunkylover53-mails/123`. Remember that the certificate must be trusted by the browser; otherwise you will get a warning message. Even if you are getting a warning from the browser (perhaps because you are running in a test environment where you have not configured your trusted certificates), the browser should let you see the certificate and verify that it's the one you have indeed configured on the server, then manually accept it.

Let's now see how to invoke this HTTPS server with Restlet on the client side. This time, because you use the server certificate to encrypt the communication, you don't need to provide a keystore. But because you used a self-signed certificate so far, you need to explicitly say that you trust it using the previously created truststore on the client side, as detailed in the following listing.

Listing 5.2 Basic configuration of the truststore on a client

```
Client client = new Client(new Context(), Protocol.HTTPS);
Series<Parameter> parameters = client.getContext().getParameters();
parameters.add("truststorePath",
    "src/org/restlet/example/book/restlet/ch05/clientTrust.jks");
```

```

parameters.add("truststorePassword", "password");
parameters.add("truststoreType", "JKS");

ClientResource clientResource = new ClientResource(
    "https://localhost:8183/accounts/chunkylover53-mails/123");
clientResource.setNext(client);
MailResource mailClient = clientResource.wrap(MailResource.class);
mailClient.store(mailClient.retrieve());
client.stop();

```

1 Set HTTPS parameters

2 Add HTTPS client connector

After having created and configured the HTTPS client 1, you need to explicitly add it to the ClientResource instance that you'll use to communicate with your previous HTTPS server 2. The parameters correspond to the client-side truststore. You can now launch this HTTPS client after making sure that the HTTPS server is still running to see that everything works as expected.

The settings presented in this section represent the traditional cases for enabling SSL on a server and a client. More advanced settings with custom management of trust or with other types of keystore configurations may also be used, as described next.

5.1.7 Providing a custom SSL context

In addition to the settings described previously, it's possible to provide a Restlet server with customized SSL settings, using its `org.restlet.ext.ssl.SslContextFactory` abstract class (in Restlet Framework version 2.1). This can be configured via a parameter (using the class name) or via an attribute (using an existing instance). Table 5.2 lists all implementations of this interface provided by Restlet.

Table 5.2 Provided implementations of the `SslContextFactory` interface

Implementation	Description
<code>DefaultSslContextFactory</code>	Makes it possible to configure most basic options when building an <code>SSLContext</code>
<code>JsslutilsSslContextFactory</code>	Corresponds to a wrapper for the <code>SSLContextFactory</code> of jSSLutils
<code>PkixSslContextFactory</code>	Uses <code>PKIXSSLContextFactory</code> from jSSLutils and can be configured via parameters

To configure it via a parameter, pass the fully qualified class name of an implementation of `SslContextFactory` into the `sslContextFactory` parameter of the connector's context. An instance of this class will be created and its `init()` method called using the connector's parameters.

To configure it via an instance, pass the configured instance into the `sslContextFactory` attribute of the connector's context. Configuring the `SslContextFactory` can be used for FOAF+SSL authentication or grid proxy certificates. The Restlet SSL extension (see the `org.restlet.ext.ssl` package) provides a `JsslutilsSslContextFactory` which wraps jSSLutils's `SSLContextFactory` that implements some of these cases.

jSSLUtils

jSSLUtils is a small library designed to assist users who need SSL settings that are often not configurable by using the default text parameters in Java.

It provides a consistent way of setting SSL-related parameters in Restlet, Jetty, and Apache Tomcat, such as methods for configuring PKCS#11 keystores (typically based on hardware cryptographic devices) and explicit configuration of Certificate Revocation Lists (CRL), for example.

It also provides extensions to alter the trust management, for example to accept grid proxy certificates (a type of certificate used for delegation of credentials in grid computing) or FOAF+SSL certificates (used to authenticate a WebID [3] using semantic web and social networking technologies).

This project is hosted at <http://code.google.com/p/jsslutils/>.

The following snippet describes how to specify a custom SSL context factory at the Restlet client side:

```
final SSLContext customSslContext = (...)  
client.getContext().getAttributes().put("sslContextFactory",  
    new SslContextFactory() {  
        public SSLContext createSslContext() throws Exception {  
            return customSslContext;  
        }  
  
        public void init(Series<Parameter> parameters) {  
            // customSSLContext ignores standard parameters.  
        }  
    });
```

At this point, you know how to ensure the confidentiality and the integrity of exchanges between clients and servers using TLS, SSL, and HTTPS. The next step is to explain how servers can authenticate those clients to ensure that they are indeed who they say they are.

5.2 **Authenticating users**

This section describes how to authenticate users, or check the identity of a client that connects to a server. Authentication is a prerequisite for authorization, which consists of making and enforcing an authorization decision depending on authentication information.

Figure 5.2 describes elements involved when implementing authentication with Restlet from client to server sides. Note that in HTTP, authentication is most often synonymous with challenge authentication, which means that the server first specifies the type of credentials that it's expecting from the client in order to successfully authenticate subsequent requests. Then, the client can respond by providing the proper credentials to the server.

Let's first describe how to configure authentication on the client side.

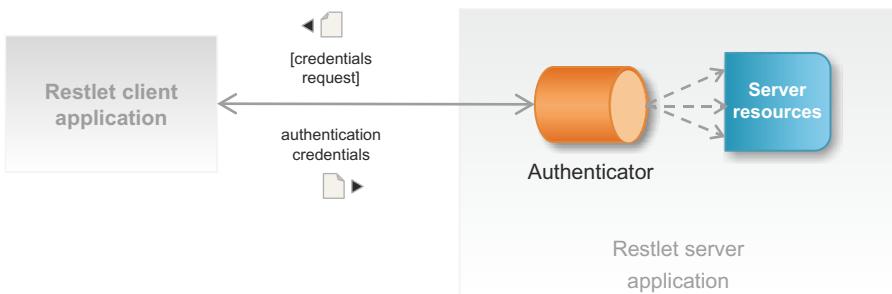


Figure 5.2 Entities involved when using authentication with Restlet on both client and server sides

5.2.1 Providing authentication credentials on the client side

On the client side, several classes from the `org.restlet.security` package of the Restlet API are involved regarding authentication:

- `ChallengeRequest`—Corresponds to the authentication challenge sent by an origin server to a client. Upon reception of this request, the client should send a new request with the proper `ChallengeResponse` set. When used with HTTP connectors, this class maps to the `WWW-Authenticate` header. Note that multiple challenge requests can be sent to a client at the same time, in case several schemes are supported.
- `ChallengeResponse`—Corresponds to the authentication response sent by a client to an origin server. This is typically following a `ChallengeRequest` sent by the origin server to the client. Sometimes, it might be faster to preemptively issue a challenge response if the client knows for sure that the target resource will require authentication. When used with HTTP connectors, this class maps to the `Authorization` header.
- `ChallengeScheme`—Corresponds to the challenge mechanism used to authenticate remote clients, such as HTTP Basic.

Let's now describe how to add client credentials to requests on the client side based on these classes.

SETTING AUTHENTICATION CREDENTIALS IN A REQUEST

Securing a request with Restlet is done using the `ChallengeResponse` class. After initializing the class, you can set it on the entity making the request, either the `Request` itself or a `ClientResource` instance, using the `setChallengeResponse` method.

The constructor of the `ChallengeResponse` class accepts three parameters:

- *The challenge scheme*—Specifies the scheme used to authenticate remote clients.
- *The identifier*—Corresponds to the user identifier, such as a login name or an access key.
- *The secret*—Corresponds to the user secret, such as a password or a secret key.

Restlet provides an important range of built-in security schemes. These are defined as constants in the `ChallengeScheme` class. Table 5.3 lists all these schemes available in version 2.1.

Table 5.3 Security schemes declared by the `ChallengeScheme` class

Name	Description
CUSTOM	Custom scheme based on IP address, query parameters, and so on
FTP_PLAIN	Plain FTP scheme
HTTP_AWS_QUERY	Amazon Web Services scheme using a URI query parameter
HTTP_AWS_S3	Amazon Web Services digest-like HTTP scheme for S3
HTTP_AZURE_SHAREDKEY	Microsoft Azure Shared Key scheme
HTTP_AZURE_SHAREDKEY_LITE	Microsoft Azure Shared Key lite scheme
HTTP_BASIC	Standard HTTP Basic scheme
HTTP_COOKIE	Special scheme using HTTP cookies
HTTP_DIGEST	Standard HTTP Digest scheme
HTTP_NTLM	Microsoft NTLM HTTP scheme
HTTP_OAUTH	OAuth HTTP scheme
POP_BASIC	Basic POP scheme
POP_DIGEST	Digest POP scheme
SDC	Google Secure Data Connector scheme
SMTP_PLAIN	Plain SMTP scheme

We won't describe all of these schemes in this chapter; we focus on HTTP Basic in this section and on HTTP Digest in the next section. Some of the schemes are described in other chapters, the S3 and SDC ones in chapter 8, respectively in sections 8.7 and 8.8.3.

Support for OAuth 2.0 and OpenID 2.0

Two new extensions related to security were added to version 2.1 of Restlet Framework, based on a contribution from Ericsson Labs. They provide support for draft versions of OAuth 2.0 (supporting delegated authentication to web APIs) and OpenID 2.0 (interoperable authentication, client and server sides).

Let's implement HTTP Basic authentication for a request. Imagine that your mail server is now secured and requires you to provide a username and a password, as we'll

explain in section 5.2.4, “Verifying user credentials.” The following code snippet describes how to preemptively provide those credentials:

```
ChallengeResponse authentication = new ChallengeResponse(
    ChallengeScheme.HTTP_BASIC, "chunkylover53", "pwd");
clientResource.setChallengeResponse(authentication);
```

As you can see, the `ChallengeResponse` class is the central class to provide those credentials in a request. Restlet internally relies on an authentication helper to convert this information into a proper HTTP header.

There are cases where authentication is a bit complex and requires some exchanges between client and server. In the next section, we’ll describe the case of the HTTP Digest authentication and how Restlet manages it.

RECEIVING SECURITY CREDENTIALS IN A RESPONSE

Some security schemes are more complex than others to implement on the client side because they require a preliminary exchange with the server before being able to authenticate a request. That’s the case with digest authentication.

The first step is to send the request without any authentication hints. The server will send back HTTP error 401 (Client Unauthorized), telling that you need an authentication. In this response, the HTTP `WWW-Authenticate` header is present and gives unique and transient information to build the authentication credentials for the next requests. Figure 5.3 gives details about interactions between client and server regarding digest authentication.

Information to build authenticated requests and send them back from the server are available with Restlet using the `ChallengeRequest` class. To get the corresponding instances, you need to iterate over the list returned by the `getChallengeRequests()`

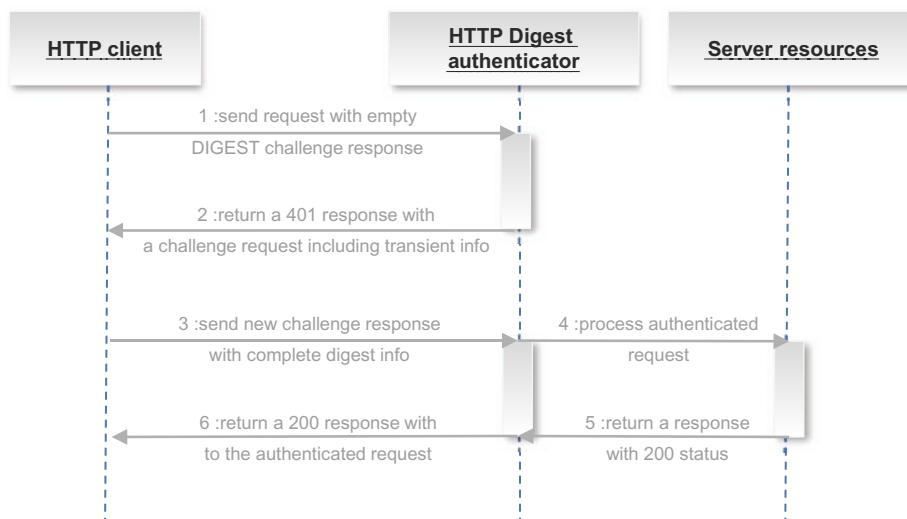


Figure 5.3 Interactions between client and server sides during HTTP Digest digest authentication

method. The one with the HTTP Digest scheme is the one you're prepared to support in this example. This instance will be used to instantiate the ChallengeResponse for subsequent requests to secured resources in addition to username and password. Listing 5.3 describes how to implement HTTP Digest authentication based on security hints received from the server on the first unauthenticated request.

Listing 5.3 Two-step client authentication with HTTP Digest

```
...
ClientResource clientResource = new ClientResource(
    "https://localhost:8183/accounts/chunkylover53-mails/123");
clientResource.setNext(client);
MailResource mailClient = clientResource.wrap(MailResource.class);

try {
    mailClient.retrieve();
} catch (ResourceException re) {
    if (Status.CLIENT_ERROR_UNAUTHORIZED.equals(re.getStatus())) {
        ChallengeRequest digestChallenge = null;

        for (ChallengeRequest challengeRequest : clientResource
            .getChallengeRequests()) {
            if (ChallengeScheme.HTTP_DIGEST.equals(challengeRequest
                .getScheme())) {
                digestChallenge = challengeRequest;
                break;
            }
        }

        ChallengeResponse authentication = new ChallengeResponse(
            digestChallenge, clientResource.getResponse(),
            "chunkylover53", "pwd");
        clientResource.setChallengeResponse(authentication);
    }
}
mailClient.store(mailClient.retrieve());
...
```

The first step is to extract security hints sent back by the server after the first unauthenticated request. Based on them, you create a new ChallengeResponse instance, adding the username and password. Once set on the ClientResource instance, the request can be sent again. This time around it will be successfully authenticated.

Before dealing with the server-side security support in Restlet and testing the previous client-side code, let's look at how the framework allows proxy authentication.

SUPPORT FOR PROXY AUTHENTICATION

Restlet also supports proxy authentication that is sometimes required in large organization in order to leave the intranet and access the web at large. This support is based on the proxyChallengeResponse property of Request. This property holds credentials that contain the authentication information of the user agent for the proxy. These credentials are sent to the server using the Proxy-Authorization HTTP header.

This feature is also used by the SDC support within Restlet in order to provide credentials necessary to authenticate with the secured channel. For more details, refer to chapter 8, section 8.9, “Accessing intranet resources with Restlet’s SDC extension.”

Let’s dive now into the server side of Restlet authentication support.

5.2.2 The `org.restlet.security.Authenticator` class

The `Authenticator` class is a `Filter` subclass that implements the mechanisms for authenticating remote users. Its function is to associate a user identity with a request using a `Verifier`. In addition, it may use the optional `Enroler`, if present, to associate the user with roles. Figure 5.4 illustrates the main `Authenticator` subclasses as well as the Restlet classes and interfaces involved in the authentication process that we’ll describe in this subsection.

The `Authenticator` is an abstract class that requires the definition of the `authenticate(Request, Response)` method in its concrete subclasses. All the preceding details about challenge requests and responses are useful when implementing this method. Typically, authentication is performed by presenting the client with a challenge, via an HTTP header mechanism.

The `ChallengeAuthenticator`, described in the next subsection, is a concrete `Authenticator` that implements the challenge mechanism and uses a `Verifier`, described in section 5.2.4, to verify the response to such challenges.

Some authenticators only rely on existing information associated with the request, without requiring an additional challenge. This is the case with the `CertificateAuthenticator`, described in section 5.2.5, which supports the authentication via trusted TLS client certificates.

By default, authenticators don’t forward the request to the next Restlet in the routing chain if the request isn’t authenticated successfully. But they may be configured for optional authentication using the optional Boolean property, in which case the request will go through even if no authentication information was obtained.

Authenticators aren’t to be confused with authorizers, covered in section 5.4, which make authorization decisions about whether the request will be able to proceed, with or

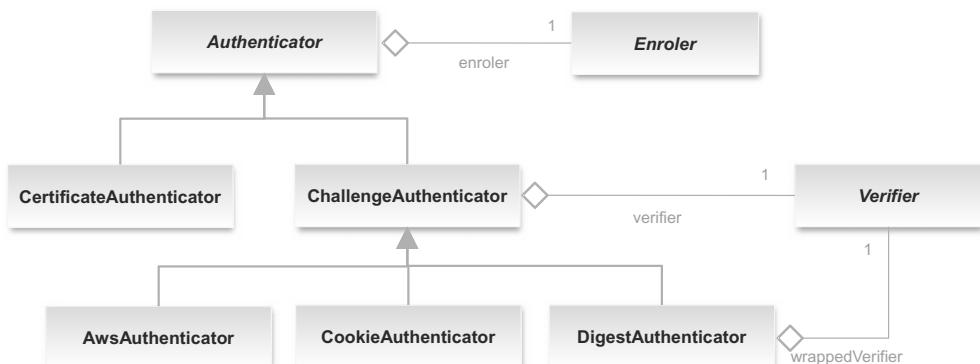


Figure 5.4 Hierarchy of authenticator classes

without user authentication. Typically, an Authenticator is immediately chained to an Authorizer, which is then chained to the protected Restlet.

Let's now describe the authenticator that uses the challenge mechanism described previously to authenticate requests.

5.2.3 Challenge-based authentication

The ChallengeAuthenticator class provides mechanisms to authenticate client requests via a challenge. The verification of the credentials provided by the client in response to this challenge is delegated to its Verifier, as described in the next subsection. The ChallengeAuthenticator expects to be initialized with a ChallengeScheme, a realm name (used by a number of challenge schemes), and a Verifier.

The purpose of the Verifier is to ascertain that the user's response to the challenge is the one that was expected (ensuring that users are who they say they are). Because authenticators are filters, they use the Restlet routing system described in section 2.4.

The simplest ChallengeAuthenticator is one that uses HTTP Basic authentication, where the user identifier and password are almost sent as clear text via standard HTTP headers. This mode triggers a popup dialog in web browsers. Once the identifier and passwords reach the ChallengeAuthenticator, it can use a Verifier to check whether they match what is expected, as described in the next section. Note that as the password is sent in the clear, this scheme is only acceptable when used over HTTPS. The following listing shows how to protect your example mail server application with the HTTP Basic scheme.

Listing 5.4 Protecting resources with HTTP Basic

```
public Restlet createInboundRoot() {
    Router router = new Router(getContext());
    router.attach("/accounts/{accountId}/mails/{mailId}", MailServerResource.class);

    ChallengeAuthenticator authenticator = new ChallengeAuthenticator(           ←
        getContext(), ChallengeScheme.HTTP_BASIC, "My Realm");                   ← Create
    MapVerifier verifier = new MapVerifier();                                     ← | Set credentials verifier
    verifier.getLocalSecrets().put("chunkylover53", "pwd".toCharArray());
    authenticator.setVerifier(verifier);                                         ← | Chain authenticator
    authenticator.setNext(router);                                              ← | with router
    return authenticator;
}
```

Even though in this example you used the MapVerifier class to store the authentication secrets in an easy way, keep in mind that this verifier isn't secure by itself because all passwords are available in clear from the JVM memory.

Another common authentication mechanism is HTTP Digest, also using standard HTTP headers. It's stronger than Basic, using a hash signature computed from the password as a shared key. It can be used over regular HTTP but has documented weaknesses

in brute force attacks. Because the mechanism required for HTTP Digest authentication is more complex and requires cryptographic functions, this is implemented in the `DigestAuthenticator`, a subclass of `ChallengeAuthenticator` present in the `org.restlet.ext.crypto` extension. The next listing adapts the example to use HTTP Digest authentication.

Listing 5.5 Protecting resources with HTTP Digest

```
public Restlet createInboundRoot() {
    Router router = new Router(getContext());
    router.attach("/accounts/{accountId}/mails/{mailId}",
        MailServerResource.class);

    DigestAuthenticator authenticator = new DigestAuthenticator(
        getContext(), "My Realm", "My Server Key");
    MapVerifier verifier = new MapVerifier();
    verifier.getLocalSecrets().put("chunkylover53", "pwd".toCharArray());
    authenticator.setWrappedVerifier(verifier);
    authenticator.setNext(router);
    return authenticator;
}
```

You might have noticed that the previous snippet called the `setWrappedVerifier()` method instead of the `setVerifier()` one used in the HTTP Basic case because `DigestAuthenticator` needs to wrap your regular verifier into a `DigestVerifier` one to do all the HTTP Digest computation, as illustrated in figure 5.4.

In the next section, we explain what a verifier is and how to verify the credentials in a `ChallengeResponse` obtained by the `ChallengeAuthenticator` object.

5.2.4 Verifying user credentials

Although the `ChallengeAuthenticator` is the Restlet filter presenting the challenge and receiving its response, the verification of the response provided by the user is delegated to the `Verifier` interface, described in the figure 5.5.

The `verify` method is responsible for checking if the user can be authenticated using credentials present in the request. This method returns an integer describing the corresponding result. Some constants are present in the interface to describe all possible results.

Three implementations of the `Verifier` interface are described in this section: the `SecretVerifier` (the simplest form), the `DigestVerifier` (used for HTTP Digest authentication), and the `JaasVerifier` (that relies on the standard Java Authentication and Authorization Service).

Let's look at those three types of verifiers in more detail.

SECRET VERIFIER

The `SecretVerifier` is an abstract `Verifier` implementation that can check the identifier and secret (typically a password) provided by the user. Upon success, it associates the `Request` with a new `User` built from the identifier value. The `SecretVerifier`

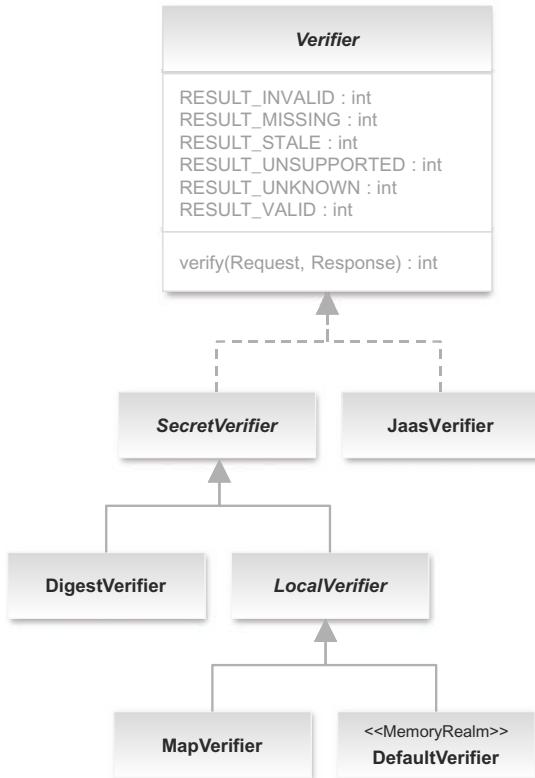


Figure 5.5 Hierarchy of credentials verifiers

doesn't require shared knowledge of the secret, but at least needs a way to verify that secret, for example via a one-way digest of the password. It can be implemented as a map of identifiers and passwords (see the `MapVerifier` class) or check the secret against other mechanisms, such as Apache Httpd's htaccess files or an LDAP store.

The following snippet describes a simple implementation of the `SecretVerifier` class that checks a hard-coded username and password. The verification process could also be done using a database or a directory:

```

public class SimpleSecretVerifier extends SecretVerifier {
    @Override
    public int verify(String identifier, char[] secret) {
        return ("chunkylover53".equals(identifier)) && compare(
            "pwd".toCharArray(), secret)) ?
            RESULT_VALID : RESULT_INVALID;
    }
}
  
```

Let's deal with another kind of verifier, the one dedicated to secret digest-based authentication.

SECRET DIGEST VERIFIER

Due to the nature of HTTP Digest authentication, the `DigestAuthenticator` requires the `Verifier` to know the secret to verify. This is an extra constraint compared with the `SecretVerifier`. The `LocalVerifier` is a subclass of `SecretVerifier` that not only verifies that a secret is valid, but can also get this shared secret in clear text in order to compute a digest from it that can be compared with the digest provided by the user, following the standard HTTP Digest mechanism.

As you saw in listing 5.5, such a `SecretVerifier` is then wrapped in the `DigestVerifier` associated with the `DigestAuthenticator`. Let's deal with the Restlet security support based on the JAAS technology.

JAAS VERIFIER

In circumstances where your Restlet application is deployed as part of a more complex system, it may be useful to rely on the Java Authentication and Authorization Service (JAAS) to verify the user's credentials with JAAS login modules. If you aren't familiar with JAAS, you might want to read more about this standard Java API first, or skip this subsection as it's a pretty advanced topic.

The `LoginModule` interface is an abstraction for a pluggable authentication mechanism in the JAAS architecture. It's responsible for verifying the user credentials and populating the JAAS `Subject` passed to it with various `Principals` when authentication is successful. A principal is a general way of modeling an authenticated entity, such as an individual or a role, as described in section 5.3.1. Then, a `LoginModule` can communicate with the application requesting authentication via `Callbacks` and the application's `CallbackHandler`.

The `JaasVerifier`, part of the `org.restlet.ext.jaas` extension, provides a `CallbackHandler` that supports the `NameCallback` and the `PasswordCallback`, obtained from the `ChallengeResponse`'s identifier and secret, respectively. Therefore, the `JaasVerifier` can be configured to use any `LoginModule` that requires either or both of these `Callbacks`, which are common. There are a number of `LoginModules` provided as part of various libraries or containers. Some, like the `LdapLoginModule`, are provided with Java SE. Although documenting JAAS and `LoginModules` in detail is outside the scope of this book, here is an example showing how to use the `LdapLoginModule` with Restlet. The `JaasVerifier` can be plugged into any `ChallengeAuthenticator` from which it can extract a secret; for example one with HTTP Basic authentication, as illustrated here:

```
JaasVerifier verifier = new JaasVerifier("MailServerApplication");
verifier.setUserPrincipalClassName("com.sun.security.auth.UserPrincipal");
authenticator.setVerifier(verifier);
```

In addition, JAAS must be configured, either programmatically or via a policy file passed via the `java.security.auth.login.config` system property. In this case, the policy file could be along these lines (you are invited to look up the Java API documentation of `LdapLoginModule` for further details):

```
MailServerApplication {
    com.sun.security.auth.module.LdapLoginModule REQUIRED
```

```

userProvider="ldap://ldap.example.net/"
authIdentity="uid={USERNAME},ou=people,dc=example,dc=net"
useSSL="false"
debug="true"
};


```

Here, `MailServerApplication` is the JAAS application name, which must match the name given when constructing the `JaasVerifier`. This login module will provide new principals when the authentication is successful. In particular, this module adds an instance of `com.sun.security.auth.UserPrincipal` containing the username and an `LdapPrincipal`. Each `LoginModule` can add any number of principals to the `Subject`. Each of these instances may be from different concrete implementations of `Principal`. They may represent the users or various attributes such as roles. To distinguish what these instances represent, the `JaasVerifier` can be configured with a class name to indicate from which instance to extract the Restlet User name. In the previous example, using `setUserPrincipalClassName()`, the request's `User` is set to the name of the first principal returned of class `com.sun.security.auth.UserPrincipal`.

Although the `JaasVerifier` doesn't do any mapping of potential role principals obtained via a `LoginModule` to Restlet roles, these principals are retained with the `ClientInfo` list of principals and may be used by an `Enroler`.

Even though this is the most common authentication mechanism in HTTP, challenge authentication isn't the only one available. Another popular one is based on SSL certificates and is presented next.

5.2.5 Certificate-based authentication

Certificate-based authentication is a mechanism that doesn't rely on a challenge at the HTTP or application layer, because this is done via the TLS handshake underneath.

Client certificate authentication is done via the configuration, on the server side, of the TLS stack to request a certificate from the client. The server must be configured with a trust store that contains the trust anchors used to verify a client certificate and additionally configured to want (or need) a client certificate when the client connects. If the `wantClientAuthentication` parameter is set to true, the connection will be established even if the client doesn't present a certificate. In contrast, if the `needClientAuthentication` parameter is set to true, the TLS handshake will fail if no certificate is presented, without moving the connection to the HTTP layer. Presenting a certificate that isn't trusted will make the connection abort in either mechanism. Note that if both parameters are set to true, it's equivalent to requiring a client certificate:

```
parameters.add("wantClientAuthentication", "true");
```

In both cases the server asks for a client certificate. The `wantClientAuthentication` parameter indicates that it's an optional request, but `needClientAuthentication` will abort the SSL handshake if no client certificate is presented, without any HTTP exchange allowed, and therefore without the possibility of an error page being presented.

Client certificate authentication will often require you to configure a trust store to indicate which CA certificates you trust to have issued client certificates for your system. This can be done by setting the following additional properties on the context:

```
parameters.add("truststorePath",
    "src/org/restlet/example/book/restlet/ch05/serverTrust.jks");
parameters.add("truststorePassword", "password");
parameters.add("truststoreType", "JKS");
```

Configuring a client certificate allows complete trusted communication. From the client to the server, requests are encrypted using the server certificate. This is already the case without a client certificate. What is different now is that the response is encrypted using the client certificate so only this client can decrypt the corresponding content.

Such configuration can also be defined within Web navigators. Firefox contains its own certificate store that can be managed through the Preferences – Advanced – Manage certificates section. Internet Explorer directly relies on the Windows certificate store.

In addition, the `javax.net.ssl.SSLContext` may be configured via customized instances of `org.restlet.engine.security.SslContextFactory` to change the trust settings. The Java SE default settings use the PKIX algorithm, whereby certificates are verified against CA certificates in the trust store; this is also the traditional model used by browsers. In this model, a hierarchical chain is built between the certificates to verify a certificate in the trust store. A certificate is verified if it has been emitted using a certificate in the trust store or via an intermediary that has; there may be a number of intermediate certificates in the chain (see section 5.1.1 for details).

Other TrustManagers may be configured if you need alternative trust models that aren't handled by default in Java, for example to accept grid proxy certificates (RFC 3280) or FOAF+SSL certificates, using a custom `SslContextFactory`, as described in section 5.1.6.

Once the authentication has been successful at the TLS layer, it needs to be handled at the HTTP layer in Restlet. This may be done by extending the `Authenticator` class to take the certificate chain presented in the request's `clientInfo.certificates` property and mapping it to a user `Principal`.

You can now adapt the mail example application to this authentication scheme. First, you have to create a key store for the client side using this command line:

```
keytool -keystore clientKey.jks -alias client -genkey -keyalg RSA
        -keysize 2048 -dname
        "CN=friends.simpson.org,OU=Simpson friends,O=The Simpsons,C=US"
        -sigalg "SHA1withRSA"
```

Now, you can export the new certificate and import it into a new `serverTrust.jks` truststore as explained in sections 5.1.3 and 5.1.5. Then, you need to configure the MailClient HTTPS connector with both a keystore (`clientKey.jks`) and a truststore (`clientTrust.jks`). On the server side, you only need to replace, in the `MailApplication`, the challenge-based authenticator with the `CertificateAuthenticator` (added to the SSL extension in version 2.1) as illustrated in the next listing.

Listing 5.6 Protecting resources with trusted TLS client certificates

```

public Restlet createInboundRoot() {
    Router router = new Router(getContext());
    router.attach("/accounts/{accountId}/mails/{mailId}",
        MailServerResource.class);

    Authenticator authenticator = new CertificateAuthenticator (
        getContext());
    authenticator.setNext(router);
    return authenticator;
}

```

Create authenticator

Chain authenticator with router

Once the authentication of the user has been completed, whether via a client certificate or via an HTTP challenge, as described in this section, it's generally useful to obtain additional information regarding the user, so as to be able to make an authorization decision. Typically, this additional information consists of roles that the user may have in the organization or the system, as described next.

5.3 Assigning roles to authenticated users

Many systems rely on Role-Based Access Control (RBAC), in which individuals are assigned roles in the system, which are then used to make the authorization decision. This allows for the decoupling of the individuals from the authorization they're granted.

For example, instead of authorizing Homer to read a particular document because this document is part of the payroll application and Homer also is the head of the Simpson's family finances, the CFO role could be defined and granted to Homer so that if it's no longer him, then the next user in that role will still have access to this document.

This section presents the `org.restlet.security.Enroler` interface and its associated classes aimed to associate roles with user requests. It also presents the structures available to model organizations, groups of users, and roles.

5.3.1 Request principals

The Java security framework relies on the `java.security.Principal` interface for granting permissions. The `Principal` interface has only one method, `getName()`, and represents an authenticated entity, in the broad sense of the term: this can be an individual, a service, a role, and so forth.

On the Restlet API side, the request's `clientInfo` property can contain three kinds of principals: one user, a list of roles, and a list of additional principals. User and Role classes in the `org.restlet.security` package are two predefined principals in the Restlet API. All principals can be accessed and modified via their accessors on an instance of `ClientInfo` (see the `getUser()`, `getRoles()`, and `getPrincipals()` methods). The association of roles based on a user is done via the `Enroler` interface, described in the next subsection.

5.3.2 The `org.restlet.security.Enroler` interface

The Authenticator class typically invokes the Enroler specified via the enroler property after a successful authentication in order to associate roles with the request's clientInfo property. Its single method, `enrole(ClientInfo)`, is expected to add roles to the ClientInfo instance based on the authenticated user and principals it already refers to. It may also be used to add other principals if necessary.

Such information may be obtained independently of the act of verifying the remote user's identity, although it should only be associated with an authenticated entity. This provides some flexibility regarding modeling of roles and other user-related data. The Restlet API provides a level of abstraction to model these roles and a default Enroler, described in the next subsection.

Alternatively, more customized retrieval of role data can be achieved, for example, by querying an LDAP directory based on the Distinguished Name authenticated via a TLS client certificate.

5.3.3 Organizations, users, and groups

The structure of an organization, in particular users, groups, and roles, may be modeled via the `org.restlet.security.Realm` class, more specifically, the `MemoryRealm`.

In this model, a number of users may be grouped into an `org.restlet.security.Group` instance, and groups can also be nested. Both users and groups may be associated with roles. The groups are useful for allocating roles to a set of users, as the role in the system may depend on whether users belong to a particular group. Users and groups tend to represent entities within the scope of an organization, whereas roles are mapped from users and groups within the context of a particular application. One benefit of this separation is the increased portability of the application between heterogeneous deployment environments.

Let's put this feature in practice in the next listing, defining the Simpsons family as an organization composed of users within the `MailServerComponent` introduced in section 3.3.1. At the same time, you complete the example in order to fully use annotated interfaces covered in section 4.5.4. You can see the result in the source code provided along this book.

Listing 5.7 Defining a memory realm with users and mapping to roles

```
public MailServerComponent() throws Exception {
    ...
    MailServerApplication app = new MailServerApplication();
    getDefaultHost().attachDefault(app);
    MemoryRealm realm = new MemoryRealm();
    User homer = new User("chunkylover53", "pwd", "Homer", "Simpson",
        "homer@simpson.org");
    realm.getUsers().add(homer);
    realm.map(homer, app.getRole("CFO"));
    realm.map(homer, app.getRole("User"));
}
```

The code is annotated with three callouts:

- A callout pointing to the line `getDefaultHost().attachDefault(app);` with the text **Attach application to virtual host**.
- A callout pointing to the line `MemoryRealm realm = new MemoryRealm();` with the text **Configure security realm**.
- A callout pointing to the line `realm.map(homer, app.getRole("User"));` with the text **Give Homer CFO, User roles**.

```

User marge = new User("bretzels34", "pwd", "Marge", "Simpson",
    "marge@simpson.org");
realm.getUsers().add(marge);
realm.map(marge, app.getRole("User"));

User bart = new User("jojo10", "pwd", "Marge", "Simpson",
    "bart@simpson.org");
realm.getUsers().add(bart);
realm.map(bart, app.getRole("User"));

User lisa = new User("lisal1984", "pwd", "Marge", "Simpson",
    "lisa@simpson.org");
realm.getUsers().add(lisa);
realm.map(lisa, app.getRole("User"));

app.getContext().setDefaultEnroler(realm.getEnroler());
app.getContext().setDefaultVerifier(realm.getVerifier()); ←
...
}

```

Set realm's
default
enroler,
verifier

On the client side, you don't need to change anything to your MailClient provided in listing 3.2 besides adding the challenge authentication (switching back to HTTP Basic) exactly like in section 5.2.1. You can launch the test client to confirm that the example still works, but note that although the authenticated Homer is given the CFO role, it isn't taken into account when processing. In section 5.4, we'll cover how to enforce authorization policies, but for now let's explain how the realm's enroler and verifier can be used within the application.

5.3.4 The default enroler and verifier

The default enroler of the application's context is the enroler used by the authenticators within that application (unless specified otherwise). It's called after each successful authentication to allocate roles to the request, whether or not it's configured with your main realm's enroler (as shown previously).

It's convenient to set such an enroler for your application, so as to avoid having to repeat this code explicitly for each authenticator it may contain. In the same spirit, there is a default verifier that can be set on the Context class that is used by default in ChallengeAuthenticator instances when no custom verifier is provided.

5.4 Authorizing user actions

This section describes how to authorize access, following the authentication of the user. This is where the decision of granting or denying access is made and enforced.

5.4.1 The org.restlet.security.Authorizer class

The Authorizer is a Filter subclass that provides the mechanism to authorize requests. Its function is to grant or deny the request access to the next Restlet it protects, depending on the authentication information passed to it.

In particular, the authentication information may include a user name, roles, and various principals, as described in section 5.3, which may be used, for example, by the

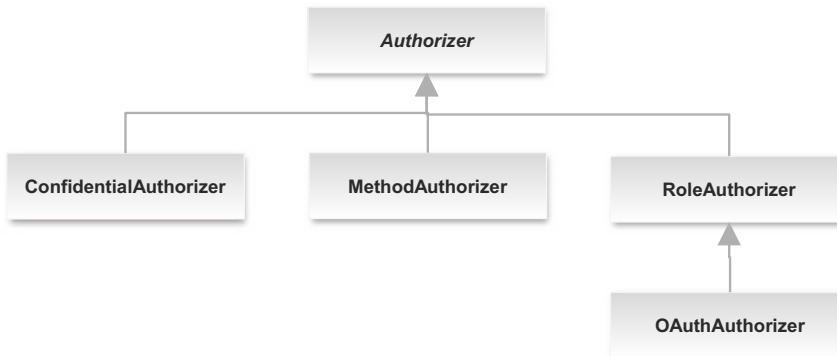


Figure 5.6 Hierarchy of authorizers

`RoleAuthorizer` (see section 5.4.2). In addition, the authorization can also depend on the request itself; for example, the `MethodAuthorizer` (see section 5.4.3) makes its authorization decisions depending on the HTTP method.

In figure 5.6 hereafter, you can see the `Authorizer` subclasses provided by the Restlet Framework, including `ConfidentialAuthorizer` to only authorize secure requests, such as those coming through an HTTPS connector.

In next subsection, we'll look more closely at the role authorizer.

5.4.2 The role authorizer

As described in section 5.3, permissions are often granted to roles rather than a specific user, allowing for more flexibility in the management of the permissions within an organization. Permissions tend to be granted to people depending on their role with respect to the application, which in turn may depend on the position within the organization, rather than being tied to a particular individual. The mapping of roles to users is done using the `Enroler`, described in section 5.3, before the request is passed on to the authorizer.

The `RoleAuthorizer` is an authorizer that makes its decision depending on whether the client has been allocated certain roles. This authorizer is configured with a set of authorized and forbidden roles. A client will be allowed through only if it has at least one of the authorized roles and none of the forbidden roles.

A `RoleAuthorizer` that would only authorize users that have the CFO role would be configured as follows:

```

RoleAuthorizer authorizer = new RoleAuthorizer();
authorizer.getAuthorizedRoles().add(getRole("CFO"));
authorizer.setNext(router);
  
```

This uses the `getRole(String)` method of `Application` to get the registered instance of `Role` from its name. Note that you also need to declare the supported roles in the `MailServerApplication`'s constructor like this:

```

getRoles().add(new Role("CFO"));
getRoles().add(new Role("User"));
  
```

Alternatively, some authorizers may depend on the method used, but not necessarily on the identity or roles of the user. The next subsection presents the method authorizer.

5.4.3 The method authorizer

The MethodAuthorizer is an authorizer that makes its decision depending on the request method and on whether the client is anonymous or authenticated. This authorizer is configured with a set of methods allowed for anonymous users and a set of methods allowed for authenticated users. A MethodAuthorizer that would let any GET user do a request but allow POST only for authenticated users would be configured as illustrated here:

```
MethodAuthorizer authorizer = new MethodAuthorizer();
authorizer.getAnonymousMethods().add(Method.GET);
authorizer.getAuthenticatedMethods().add(Method.GET);
authorizer.getAuthenticatedMethods().add(Method.POST);
authorizer.setNext(router);
```

In some cases, knowing only the method used in the request to perform the action is too coarse to be able to make an authorization decision. The next subsection explains how to implement fine-grained authorization, where the decision may also depend on the state of the resource.

5.4.4 Fine-grained authorization

Finer-grained authorization may be performed within a given resource; for example, if authorization also depends on the content of this resource. This is often done in a bespoke manner, although the authorizers described previously may be used independently of being filters (using their authorize method).

In an email application, each account has a different owner. Only Homer ought to be able to read his emails. When the client sends a request to read emails from a given account, it's only once the data about this particular account has been retrieved by the ServerResource that an authorization decision can be made.

In addition, authorization regarding a particular resource may depend on the state of that resource at a given moment in time. In these cases, finer-grained authorization that depends on the state of the resource itself is required. This could not be achieved by a filter that would not have prior knowledge of the resource data.

Although this may require some customized implementation depending on the data this kind of resource models, utility methods such as `ServerResource.isInRole(String roleName)` can assist when performing fine-grained authorization within the resource as illustrated here:

```
public String represent() {
    String result = AccountsServerResource.getAccounts()
        .get(this.accountId);

    if (isInRole("CFO")) {
        return result + " (CFO access)";
    } else {
```

```

        return result;
    }
}

```

We'll now complete our overview of authorization support in Restlet by covering how to use the JVM security manager, combined with JAAS.

5.4.5 Using Java security manager

It can be desirable to benefit from the Java security manager's ability to sandbox a Restlet Application within its host virtual machine. Although the Restlet Framework doesn't provide an explicit security manager configuration, this section provides pointers and examples regarding how this can be done.

If you intend to use the security manager, it's recommended that you read its official Java documentation, in particular the section on the syntax of policy files. Activating the default security manager can be done using this system property flag (no value is required):

```
-Djava.security.manager
```

The policy file may be configured using this flag:

```
-Djava.security.policy="src/org/restlet/example/book/restlet/ch05/sec4/
server/jaas.policy"
```

This section is based on a short example, whereby a server resource is allowed to read the server's home directory only if the user connecting to it has the *CFO* role. An example policy file is shown in listing 5.8. The first grant block grants permissions with the basics for the Restlet server to run. The second block grants the permission to read the server's home directory to subjects with the CFO role. (This is only an example, which is probably too permissive for serious applications.)

Listing 5.8 Example policy file

```

grant {
    permission java.lang.RuntimePermission "*";
    permission java.net.NetPermission "*";
    permission java.util.logging.LoggingPermission "control";
    permission java.util.PropertyPermission "*", "read";
    permission java.net.SocketPermission "*", "listen,accept,resolve";
    permission javax.security.auth.AuthPermission "modifyPrincipals";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
};

grant principal org.restlet.security.Role "CFO" {
    permission java.io.FilePermission "${user.home}", "read";
};

```

In order to associate the user's principals to the request, the static `JaasUtils.doAsPrivileged()` method may be used within a `ServerResource`, as illustrated next.

Listing 5.9 Running sensitive code as a privileged user

```

public class FileServerResource extends ServerResource {
    @Get("txt")
    public Representation retrieve() throws ResourceException {
        StringBuilder result = null;

        PrivilegedAction<StringBuilder> action = new
        PrivilegedAction<StringBuilder>() {
            public StringBuilder run(){
                File dir = new File(System.getProperty("user.home"));
                String[] filenames = dir.list(new FilenameFilter(){
                    public boolean accept(File dir, String name) {
                        return !name.startsWith(".");
                    }
                });
                StringBuilder sb = new StringBuilder(
                    "Files in the home directory: \n\n");
                for (String filename : filenames) {
                    sb.append(filename);
                    sb.append("\n");
                }
                return sb;
            }
        };
        try {
            result = JaasUtils.doAsPrivileged(
                getRequest().getClientInfo(), action);
        } catch (AccessControlException ace) {
            setStatus(Status.CLIENT_ERROR_FORBIDDEN);
        }
    }
    return (result == null) ? null : new StringRepresentation(result);
}

```

Action requiring CFO role to run

Invoke privileged action

Return home dir files listing

If you attach this `FileServerResource` to the router of your `MailServerApplication` using the `/accounts/{accountId}/files` URI template, you can issue GET calls and compare the result with and without proper authentication of the `chunkylover53` user, which grants to the authenticated user a principal of class `org.restlet.security.Role`, with the name `CFO`. Then, the action of reading the home directory is allowed, as instructed in the second grant block in the example policy file.

In the next section, we move away from the topic of authentication and authorization, to focus on ensuring the integrity of the data transfers.

5.5 Ensuring end-to-end integrity of data

In some cases, one of the risks that may be associated with an application is due to the potential failures that would cause the data to be altered before, during, or after the communication. This section presents how to ensure end-to-end integrity of data, by using digests of the representations. These digests aren't cryptographically signed,

therefore they protect only against accidental modifications of the representations, not against man-in-the-middle attacks, which could potentially replace the digest too.

This section briefly introduces the concepts related to digests in general and within the context of HTTP (more specifically the Content-MD5 header). Then, we show how to use them with the Restlet Framework.

5.5.1 Ensuring representation integrity

HTTP provides a way to ensure the integrity of a representation by using a header that contains the digest of the representation. The digest is the result of a one-way function that would change almost completely given a minor alteration of the representation. Commonly used digest algorithms include MD5 and SHA-1, although both are now considered to have weaknesses. Users willing to ensure additional security against active attackers will need to check the robustness of the digest algorithm they use when they implement their systems.

The Content-MD5 header, defined in the HTTP specification, is optional and may be present in responses and requests that have an entity. This header must contain the MD5 digest of the representation in the associated entity. It may be verified by any parties in the exchange, but must not be modified by intermediates, in particular by proxy servers. The next listing shows an example HTTP response that includes a Content-MD5 header.

Listing 5.10 Example HTTP response with Content-MD5 header

```
HTTP/1.1 200 The request has succeeded
Date: Thu, 27 May 2010 20:31:39 GMT
Server: Restlet-Framework
Content-Length: 12
Content-MD5: 7Qdih1MuhjZehB6Sv8UNjA==
Content-Type: text/plain; charset=UTF-8

Hello World!
```

The recipient of this representation may verify the integrity of the representation by computing its digest and comparing it with the reference digest in the Content-MD5 header. Computing the digest and attaching it to the representation can be done in the Restlet Framework using the `DigesterRepresentation`, which wraps an existing representation and provides the utility methods for sending and verifying the digests, as described next.

5.5.2 Representation digesting

The operations used for digesting representations as a sender and as a receiver are similar. In both cases, the representation must be fully read and the digest must be computed. The main distinction between the two operations is that the receiver must compare the digest it computed from the representation it has received with the digest that was received with it. Both operations can be handled with a `DigesterRepresentation`, more specifically its `computeDigest()` and `checkDigest()` methods.

These two methods only finalize the computation; consuming the representation via the `DigesterRepresentation` methods is an essential part of the computation.

As a sender, setting the digest of a representation in the `Content-MD5` header can be achieved by wrapping this representation in a `DigesterRepresentation`. This is a four-step process, as illustrated in listing 5.11:

- 1 The initial representation must be wrapped into a `DigesterRepresentation`.
- 2 Its content must be fully read, as this contributes to the computation of the digest (in this particular example, the content is read and discarded via `exhaust()`).
- 3 The digest must be computed and finalized, using `computeDigest()`.
- 4 The digest must be set as a property of the representation.

Listing 5.11 Example setting the representation's digest as a sender

```
public class VerifiedServerResource extends ServerResource {
    @Get
    public Representation represent() throws Exception {
        DigesterRepresentation result = new DigesterRepresentation(
            new StringRepresentation("hello, world"));
        result.exhaust(); ← Wrap String-
        result.setDigest(result.computeDigest()); ← Representation
        return result;
    }
}
```

As `exhaust()` consumes and discards the representation entirely, this technique can't be used for representations for which the source may only be read once, otherwise the content would be entirely lost. The next section describes how to use the `DigesterRepresentation` without losing content.

As a receiver, verifying the digest of a representation is a three-step process. First, the received representation must be wrapped with a `DigesterRepresentation`. Second, its content must be read fully. Third, the digest must be computed and compared with the existing `Content-MD5` header. This last step is done via the `checkDigest()` method:

```
DigesterRepresentation digesterRepresentation =
    new DigesterRepresentation(response.getEntity());
digesterRepresentation.exhaust();
boolean correctDigest = digesterRepresentation.checkDigest();
```

Again, this example discards the content representation received using `exhaust()`. The following section describes how not to lose content when digesting a representation.

5.5.3 Digesting without losing content

The computation of the digest of a representation requires processing the entire content of this representation. The `DigesterRepresentation` class is a wrapper that can let you use a representation and compute its digest while the content is being read. Therefore, using a `DigesterRepresentation` via its `getStream()`, `getReader()`, `getText()`, or `write()` methods will transparently contribute to the computation of the digest of the

wrapped representation. Once the entire representation has been processed, the `computeDigest()` method will return the computed digest, or `checkDigest()` will verify the representation against the associated digest property.

This can be achieved when processing a representation that is being received, while using its content. Unfortunately, when sending the representation, this requires the representation to be read twice from its beginning, because the related HTTP header has to be sent before the content. This problem could be overcome by using trailer headers, which aren't supported yet in Restlet Framework 2.1.

The following example writes the representation received onto `System.out`. Because the `write()` method is called on the wrapping `DigesterRepresentation` and not the original representation, the digest is computed while the representation is being consumed:

```
Representation responseEntity = response.getEntity();
DigesterRepresentation digesterRepresentation =
    new DigesterRepresentation(responseEntity);
digesterRepresentation.write(System.out);
boolean correctDigest = digesterRepresentation.checkDigest();
```

This lets the application use the content of the representation while computing its digest at the same time. The `checkDigest()` method then finalizes the digest computation and checks it against the value obtained from the `Content-MD5` header. If no digest was associated with the original representation, then `checkDigest()` will return `false`.

5.6 *Summary*

In this chapter, you have learned how to enable a number of security features in a Restlet application. These can be categorized into three themes: protecting the communication between the client and the server; dealing with user authentication, identity management, and authorization; and protection against accidental network failures.

You saw how to enable HTTPS on a Restlet server to protect the communication between the client and the server by configuring TLS/SSL and creating certificates.

You also learned how to authenticate remote users, or verify their identity using authenticators and verifiers. The API for authentication is flexible and allows for simple use cases (for example, a list of usernames and passwords) as well as larger organizational structures (for example, authentication and identity management via a company directory).

Then, you learned how to make and enforce an authorization decision based on the information about the client, its user, and the action they're trying to perform. Finally, you saw how to protect the representations against accidental alterations, using representation digests.

After covering security, let's now continue our road to the deployment of a Restlet application with chapter 6 which will discuss other essential tasks which are to document the RESTful web API it exposes and to deal with the versioning needs of an application during its life cycle.



Documenting and versioning a Restlet application

This chapter covers

- Documenting your web API
- When to version your web API
- Web Application Description Language (WADL)

Now that you've seen how to secure Restlet applications, it's time to continue your exploration of the tasks required to get ready to roll out. This chapter covers how to document and version the RESTful web API of your Restlet application.

First we discuss the use cases, pitfalls, and recommendations when documenting and versioning your web API. Documenting is essential in helping a development team communicate during active development or maintenance and helping users learn how to use it. Versioning ensures that existing clients won't break and alienate your user community.

We cover the Web Application Description Language (WADL) that's well-suited for documentation of RESTful web APIs and the Restlet extension for WADL, including `WadlApplication`, `WadlServerResource`, and other artifacts. You learn how to progressively describe parts of a WADL document, such as the application title and description, the list of resources, their names, URI paths, methods, and representations. To illustrate the use of this extension you'll reuse the sample RESTful mail application developed in previous chapters.

Finally you'll see how easy it is to automatically convert the WADL documents into a user-friendly HTML document.

6.1 **The purpose of documentation and versioning**

This section explores the need for documenting and versioning your RESTful web APIs with common uses cases, pitfalls when tackling such a project, and best practices to follow.

6.1.1 **Use cases**

The first and most common use case is to provide human-readable documentation of your web API to developers on the client side so they can read it online or print it. After a learning phase they'll continue to use it as a reference document while developing their client programs. Those developers are mainly looking for the URI entry points to your API resources, the supported methods, and a description of their representations.

The second use case is to coordinate among a large development team for construction or maintenance purposes. Imagine that you need to develop version 2.0 of an existing web API, without any previous knowledge. You'd naturally look for the equivalent of Javadocs for this API to learn not just how to use it but also how it was designed and implemented. This knowledge should be complementary to technical specifications and properly commented code as an essential deliverable of any RESTful web project.

The third use case is allowing the automatic generation of client programs, or at least part of them. This use case requires a formal and precise description of your REST API and is often criticized as reproducing the issues of the previous SOAP-based web services. But it can be provided for convenience purposes, in addition to a regular HTTP access.

Finally the need to maintain several versions of the same API occurs when the existing clients aren't under the control of the project, such as for open web APIs, and when you don't want to break clients by forcing them to use a newer API.

6.1.2 **Pitfalls**

In theory there's no need to generate client SDKs for a RESTful web API because clients should automatically discover the API through hypermedia-driven navigation; nor is there a need to document it because it should be self-describing. A website is a good comparison: web browsers don't need to read a user guide to display or interact with websites. This principle is hard to respect in practice, though, unless you limit yourself to standard hypermedia-driven media types such as HTML/XHTML, Atom/AtomPub, or RDF (we discuss this topic in greater detail in chapter 10 when covering Restlet support for hypermedia and the Semantic Web).

But when you use your own XML- or JSON-based media types (for example, by using XStream or Jackson extensions provided by Restlet), you gain in terms of productivity. This is particularly true for the first phases of your project, where iterations are short and reactivity is key, but you'll reduce the capacity of your clients to adapt to

future changes of your API. Change is common among web APIs, even those labeled “REST,” but by creating new XML or JSON dialects you introduce coupling between clients and servers that can lead to brittle interfaces and the need for proper versioning to prevent evolution headaches. Also most if not all web APIs available today provide users with documentation to help learn and interact with them.

Another common problem you’ll face when documenting a web API is the need to keep it synchronized with the code developed. This is true during the elaboration and construction phases of a project, but also later on when evolutions are being pushed into production. Having documentation that doesn’t reflect the reality of your API will certainly cause trouble for your users.

Let’s review some best practices to help you solve or work around these issues.

6.1.3 Recommendations

If you’re developing an open web API that can be consumed by clients outside your control, the first recommendation is to reuse existing and proven hypermedia types such as HTML/XHTML, Atom/AtomPub, and RDF and extend them if necessary. This design choice will make your API more RESTful, easier to document and consume (higher-level clients for those media types exist in many environments), more reusable, and easier to evolve. But this choice comes with a development cost: you won’t be able to use transparent and bi-directional serialization mechanisms such as those provided by XStream, Jackson, JiBX, or JAXB.

If your API relies on custom media types not specifically designed with hypermedia and independent evolution of client and server in mind, the second recommendation is to add a version number to your URIs such as `http://api.mycompany.com/v1.0/`. This way you’ll be able to maintain older versions of your API while offering a new version in a parallel URI space for API enhancements.

If you use custom media types for productivity reasons, try to make it as hypermedia-capable as possible, at least by providing hyperlinks to related resources. One way to do it is to add HREF attributes to XML elements, as in HTML. Hyperlinking, as opposed to forms and scripting, is rather easy to support, so make sure to use it extensively (for example, to point to items in a collection resource).

Regarding documentation, unless you target only browsers used by humans you’ll need to provide more formal documentation and keep it perpetually up to date, ideally by maintaining it close to your source code, as for Javadocs. If you target robots and other programmatic clients only, then consider using HTTP content negotiation to provide alternative HTML representations of your resources. Doing so will be useful for the developer of those clients to navigate, learn, and test your web API.

In HTTP, the standard way to request a description of a resource is via the OPTIONS method. If you invoke it on a root URI, or with the “`*`” special URI, it should return a description of the target URI subspace, typically of the whole application.

At this point you may feel like it adds a lot of work to follow those recommendations in a real project, so let’s look at how the Restlet Framework can help. In the next

section we introduce WADL and explain how Restlet supports it to document your web APIs.

6.2 Introducing WADL

WADL is a useful XML vocabulary designed by Marc Hadley as a more RESTful way to describe web applications than its WSDL 1.1 predecessor, which had roots in the SOAP world. Note that WSDL 2.0 added better support for RESTful web APIs but still isn't as elegant as WADL for our purpose. WADL was submitted in 2009 to the W3C by Sun Microsystems [4].

To get more concrete, the following listing shows a simple WADL example for the mail application developed in previous chapters. Though the example is small, it already provides useful information, such as the list of resources, their URIs, and the supported HTTP methods.

Listing 6.1 Sample WADL description

```
<?xml version="1.0" standalone="yes"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc title="RESTful Mail Server application">
        Example application for 'Restlet in Action' book
    </doc>
    <resources base="http://localhost:8111/">
        <resource>
            <method name="GET">
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
        </resource>
        <resource path="accounts/">
            <method name="GET">
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
            <method name="POST">
                <request>
                    <representation mediaType="text/plain"/>
                </request>
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
        </resource>
        <resource path="accounts/{accountId}">
            <method name="DELETE"/>
            <method name="GET">
                <response>
                    <representation mediaType="text/plain"/>
                </response>
            </method>
        </resource>
    </resources>
</application>
```

WADL element can be documented

Collection of resources with base URI

Account resource and URI path

Response with text/plain media type

```

<method name="PUT">
  <request>
    <representation mediaType="text/plain"/>
  </request>
</method>
</resource>
</resources>
</application>

```

This example is pretty simple, but WADL can also be used to describe expected request and response messages, including query parameters and representation media types. Let's now explore the Restlet extension for WADL provided in the `org.restlet.ext.wadl.jar` file. This extension is composed of a set of description classes such as `ApplicationInfo`, `ResourcesInfo`, `ResourceInfo`, `RequestInfo`, and so on corresponding to the WADL information model. In addition the `WadlRepresentation` class can be used to either parse an existing WADL document or generate one based on a given `ApplicationInfo` or `ResourceInfo` instance.

The two most useful classes, `WadlApplication` and `WadlServerResource`, can help to produce WADL descriptions.

6.3 The WadlApplication class

The most common use case for this extension is to describe a complete Restlet application as a WADL document. That's the purpose of the `WadlApplication` class, which extends `org.restlet.Application` (illustrated in figure 6.1) and intercepts incoming calls on the base URI of the application with the OPTIONS method. It provides several protected methods, such as `createWadlRepresentation(ApplicationInfo)`, that can be extended to customize their default behavior, even though in most cases you won't need to look into this.

Imagine that you want to describe the API of the sample mail application developed in chapter 3. You update the `MailServerApplication` to make it extend `WadlApplication` and see how it behaves. To test this let's write a simple client program that outputs the WADL document retrieved on the console.

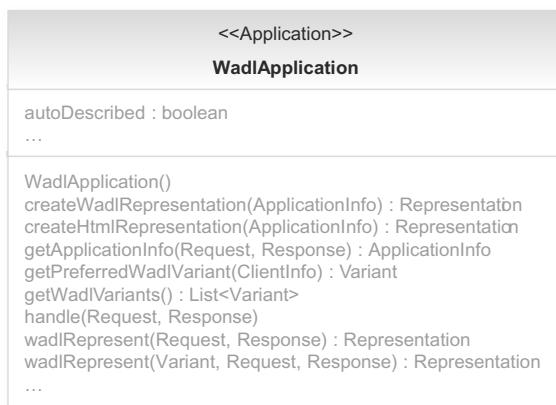


Figure 6.1 The `WadlApplication` class can be used to describe an application in WADL.

```
public static void main(String[] args) throws Exception {
    ClientResource service = new ClientResource("http://localhost:8111");
    System.out.println(service.options().getText());
}
```

You could also use any kind of HTTP client, such as curl or a web browser via the tunnel service, using the URI `http://localhost:8111?method=options`. If you first launch the `MailServerComponent`, including the modified `MailServerApplication`, and then test the client program, the console will display (as you may have guessed) the same WADL document as the one in listing 6.1. With little effort, you already have a useful result!

By automatically introspecting the application, the `WadlApplication` class was able to detect the available resources and their URI templates, the methods supported by each, and the request and response representations supported. Note that the name and description properties of the application are also automatically extracted into the WADL documentation in the root `application` element.

This introspection process is capable of recursively traversing filters and routers until it reaches a leaf `ServerResource` subclass and then instantiates them—which, by the way, can help you detect implementation issues. As you know, other sorts of leaves can be found in a Restlet routing graph, typically `Restlet` instances. For those situations, to provide a description of those leaves in the resulting WADL document, you can either have the object implement the `WadlDescribable` interface or wrap it into a `WadlWrapper` instance. You'll then only have to override the `getResourceInfo()` method and build your own WADL resource descriptor using the `ResourceInfo` class available in the WADL extension.

Next we introduce the `WadlServerResource` class to further describe each resource of the mail application in WADL.

6.4 The `WadlServerResource` class

Even though the `WadlApplication` can already discover quite a bit of useful information on your server resources, there's a limit to what can be guessed during introspection. To go beyond this limit, you need to explicitly provide additional information. For this purpose the WADL extension includes `WadlServerResource`, a subclass of `ServerResource`, which can be extended to describe anything that's supported by the WADL specification.

In this section we introduce the class and its properties, update the server resources from the sample mail application to improve their description, and illustrate how descriptions of a single resource can also be retrieved.

6.4.1 Overview of properties and methods

To discover this essential class of the WADL extension, you will first get an overview of its main properties and methods. We already covered the ones inherited from `ServerResource` in section 2.5. For this purpose we illustrate this class in figure 6.2 as a UML class diagram, including its three properties and its sets of methods.



Figure 6.2 The `WadlServerResource` class can be used to further describe resources in WADL.

First the `autoDescribing` Boolean property indicates whether the resource should directly support the `OPTIONS` method. By default it's enabled and allows a client to retrieve a WADL description snippet only for the resource that's the target of the request, and not the whole web API. We illustrate this possibility later in the section.

The two other `name` and `description` properties allow easy documentation of your resources with the possibility of using dynamic information from the request. Note that this dynamic aspect wouldn't be possible if you used regular Javadocs comments or special Java annotations.

Now to get an overview of the class methods. As you can see, many `describe*`() methods should be viewed as callbacks to be invoked by a parent `WadlApplication`, giving you a chance to customize the values returned by default.

For example if you merely want to update the description of the `GET` method with additional documentation, you need to override only `describeGet(MethodInfo)`, invoking `super.describeGet(methodInfo)` (unless you want to completely bypass the default behavior), and then add your own values to the `MethodInfo` parameter.

Two methods, `createHtmlRepresentation(ApplicationInfo)` and `createWadlRepresentation(ApplicationInfo)`, allow you to customize the WADL and HTML representation generated when `OPTIONS` is directly invoked on a `WadlServerResource` subclass. Finally, `canDescribe(Method)` allows you to remove the description of a specific method—for example, if the user doesn't have the required role.

For further explanations, including on the default behavior, refer to the Javadocs. You'll now put this useful class into practice.

6.4.2 Improving description of existing server resources

To improve the description of the sample mail application, you'll make the three server resources extend the `WadlServerResource` class instead of `ServerResource`. Start with the `AccountServerResource` and reuse the `accountId` attribute in the name and description properties, as illustrated in the following listing.

Listing 6.2 WADL-enhanced account resource with a dynamic name and description

```
public class AccountServerResource
    extends WadlServerResource
    implements AccountResource {
    private int accountId;
    @Override
    protected void doInit() throws ResourceException {
        String accountIdAttribute = getAttribute("accountId");
        if (accountIdAttribute != null) {
            this.accountId = Integer.parseInt(accountIdAttribute);
            setName("Resource for mail account '" + this.accountId + "'");
            setDescription("The resource describing mail account "
                + "number '" + this.accountId + "'");
        } else {
            setName("Mail account resource");
            setDescription("The resource describing a mail account");
        }
    }
    public String represent() {
        return AccountsServerResource.getAccounts().get(
            this.accountId);
    }
    public void store(String account) {
        AccountsServerResource.getAccounts().set(
            this.accountId, account);
    }
    public void remove() {
        AccountsServerResource.getAccounts().remove(this.accountId);
    }
}
```



Dynamic documentation easily provided

The following listing continues updating the sample resources to provide static documentation for the `AccountsServerResource` class.

Listing 6.3 WADL-enhanced accounts server resource with static documentation

```
public class AccountsServerResource extends WadlServerResource implements
    AccountsResource {
```

```

private static final List<String> accounts
    = new CopyOnWriteArrayList<String>();

@Override
protected void doInit() throws ResourceException {
    setName("Mail accounts resource");
    setDescription(
        "The resource containing the list of mail accounts");
}

public static List<String> getAccounts(){
    return accounts;
}

public String represent(){
    StringBuilder result = new StringBuilder();
    for (String account : getAccounts()) {
        result.append((account == null) ? "" : account).append('\n');
    }
    return result.toString();
}

public String add(String account) {
    getAccounts().add(account);
    return Integer.toString(getAccounts().indexOf(account));
}
}

```

Finally you take care of the RootServerResource in listing 6.4, which provides a name and a description but also sets the autoDescribing property to false. Why do this? If you remember, WadlApplication is intercepting OPTIONS calls to the root URI of the application to automatically describe the whole web API exposed to clients. This action is done when a client requests the root URI of the application and there is no other reply by underlying resources.

If you let the RootServerResource autodescribe itself in reply to OPTIONS calls, then you'd only be able to retrieve the snippet covering the root resource, not the whole API description.

Listing 6.4 WADL-enhanced root server resource

```

public class RootServerResource extends WadlServerResource implements
    RootResource {

    @Override
    protected void doInit() throws ResourceException {
        setAutoDescribing(false);
        setName("Root resource");
        setDescription("The root resource of the mail server application");
    }

    public String represent() {
        return "Welcome to the " + getApplication().getName() + " !";
    }
}

```

Now that you've associated a name and description to each resource, let's see how you could describe representations more precisely than with a text/plain media type. For this, you'll override some of the `describe()` methods provided by the `WadlServerResource`. First override the `describe(ApplicationInfo)` method in `AccountsServerResource` to add a global description of the textual representation that you use for accounts, as illustrated in the following snippet. Note that it sets an account identifier on this representation description to be able to reference it in other descriptions:

```
@Override
protected void describe(ApplicationInfo applicationInfo) {
    super.describe(applicationInfo);
    RepresentationInfo rep =
        new RepresentationInfo(MediaType.TEXT_PLAIN);
    rep.setIdentifier("account");
    applicationInfo.getRepresentations().add(rep);

    DocumentationInfo doc = new DocumentationInfo();
    doc.setTitle("Account");
    doc.setTextContent("Simple string containing the account ID");
    rep.getDocumentations().add(doc);
}
```

Then you want to describe the fact that both `AccountsServerResource` and `AccountServerResource` rely on this textual media type to represent an account in the sample application. For this purpose you'll override another `describe(...)` method as illustrated in this snippet:

```
@Override
protected RepresentationInfo describe(MethodInfo methodInfo,
    Class<?> representationClass, Variant variant) {
    RepresentationInfo result = super.describe(methodInfo,
        representationClass, variant);
    result.setReference("account");
    return result;
}
```

Finally you want to describe the remaining representation type returned by the GET method of `RootServerResource`. Again, you override the same method, but because this representation isn't reused elsewhere, it's directly described:

```
@Override
protected RepresentationInfo describe(MethodInfo methodInfo,
    Class<?> representationClass, Variant variant) {
    RepresentationInfo result = super.describe(methodInfo,
        representationClass, variant);
    result.setMediaType(MediaType.TEXT_PLAIN);
    result.setIdentifier("root");

    DocumentationInfo doc = new DocumentationInfo();
    doc.setTitle("Mail application");
    doc.setTextContent("Simple string welcoming the user " +
        "to the mail application");
```

```
        result.getDocumentations().add(doc) ;
        return result;
    }
```

It's been a while since you tested the WADL description of the application, so launch the test client again. The result displayed in the following listing shows some nice improvements to the documentation.

Listing 6.5 Improved WADL application description

```
<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="wadl2html.xslt"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc title="RESTful Mail Server application">
        Example application for 'Restlet in Action' book
    </doc>
    <representation id="account" mediaType="text/plain">
        <doc title="Account">Simple string containing the account ID</doc>
    </representation>
    <resources base="http://localhost:8111/">
        <resource>
            <doc title="Root resource">
                The root resource of the mail server application
            </doc>
            <method name="GET">
                <response>
                    <representation id="root" mediaType="text/plain">
                        <doc title="Mail application">
                            Simple string welcoming the user to the
                            mail application
                        </doc>
                    </representation>
                </response>
            </method>
        </resource>
        <resource path="accounts/">
            <doc title="Mail accounts resource">
                The resource containing the list of mail accounts
            </doc>
            <method name="GET">
                <response>
                    <representation href="#account"/></response>
                </method>
                <method name="POST">
                    <request>
                        <representation href="#account"/></request>
                    <response>
                        <representation href="#account"/></response>
                    </method>
                </resource>
                <resource path="accounts/{accountId}">
                    <doc title="Mail account resource">
                        The resource describing a mail account</doc>
                    <method name="DELETE"/>
                </resource>
            </resource>
        </application>
    </?xml-stylesheet?>
```

```

<method name="GET">
    <response>
        <representation href="#account"/></response>
    </method>
    <method name="PUT">
        <request>
            <representation href="#account"/></request>
        </method>
    </resource>
</resources>
</application>

```

Let's see if it's as easy to describe a single resource instead of the whole application.

6.4.3 Describing a single resource

Earlier in the chapter we mentioned describing a single resource instead of the whole application. You can now demonstrate this feature without any additional development. Modify the test client to address the collection of accounts and see the result:

```

public static void main(String[] args) throws Exception {
    ClientResource service = new ClientResource("http://localhost:8111");
    System.out.println(service.getChild("/accounts/").options().getText());
}

```

In the following listing you can see the resulting WADL description that should be displayed in your console.

Listing 6.6 WADL resource description snippet

```

<?xml version="1.0" standalone="yes"?>
<?xml-stylesheet type="text/xsl" href="wadl2html.xslt"?>
<application xmlns="http://wadl.dev.java.net/2009/02">
    <doc title="Mail accounts resource"/>
    <representation id="account" mediaType="text/plain">
        <doc title="Account">Simple string containing the account ID</doc>
    </representation>
    <resources>
        <resource path="accounts/">
            <doc title="Mail accounts resource">
                The resource containing the list of mail accounts
            </doc>
            <method name="GET">
                <response>
                    <representation href="#account"/></response>
                </method>
                <method name="POST">
                    <request>
                        <representation href="#account"/></request>
                    <response>
                        <representation href="#account"/></response>
                    </method>
                </resource>
            </resources>
        </application>

```

You can also override other `WadlServerResource` methods, such as `describeGet` (`MethodInfo`) to describe the GET methods; see figure 6.2 and Javadocs of the WADL extension for details.

So far we've retrieved WADL descriptions in their original XML form, but that's not easily readable by a human using a browser to learn about your cool web API. There's another built-in feature of this WADL extension: the ability to automatically convert WADL documents to an HTML equivalent.

6.5 Automatic conversion to HTML

The WADL documents you've produced so far are nice for formal descriptions or when those descriptions need to be automatically processed, but in general your users will look for something more user-friendly, typically a web page.

For this purpose the WADL extension embeds an XSLT document that was developed by Marc Nottingham from Yahoo! This XSLT transformation can be automatically applied by the `WadlApplication` and `WadlServerResource` classes, relying on standard HTTP content negotiation.

NOTE Because this WADL-to-HTML stylesheet uses the nonstandard EXSLT functions library, you need to have a recent version of Xalan-Java in your classpath instead of the default one bundled in Java SE. Version 2.6.0 has been tested successfully, but version 2.4.1 and above should work fine as well.

If you open `http://localhost:8111/?method=options` in your browser, you'll see the web page in figure 6.3. Try `http://localhost:8111/accounts/?method=options` to see

RESTful Mail Server application

Example application for 'Restlet in Action' book

- [Resources](#)
 - <http://localhost:8182/>
 - <http://localhost:8182/accounts/>
 - <http://localhost:8182/accounts/{accountId}>
- [Representations](#)
 - [Mail application \(text/plain\)](#)
 - [Account \(text/plain -\)](#)

Resources

Root resource

The root resource of the mail server application

Methods

GET

available response representations:

- [Mail application \(text/plain\)](#)

Figure 6.3 The partial WADL documentation converted to HTML

an HTML snippet describing the `AccountServerResource` only. The application knows it has to return HTML due to the browser preferences set in the `Accept` HTTP header.

Note that the `method` query parameter is interpreted by the `TunnelService` discussed in previous chapters and allows the sending of `OPTIONS` requests from your browser. If you want to use the `GET` method instead, override the `WadlApplication.handle(Request, Response)` method and invoke the WADL generation logic directly via the `wadlRepresent(Request, Response)` method.

With no further development you can provide a clear documentation of the API that will always stay synchronized with your source code.

The WADL extension can also use a WADL document—for example, provided by a third-party tool—as an input for configuring a Restlet application. This isn’t as powerful as the XML configuration mechanism introduced in chapter 3, especially a configuration based on Spring, but this can come in handy in some situations. For additional details, have a look at the constructors of the `WadlComponent` and `WadlApplication` classes.

6.6 Summary

This chapter explored how to document and version a RESTful web API exposed by a server-side Restlet application. We presented common use cases, such as the need to learn an API and have a reference document to look at when using it, or the need to coordinate among a large team of developers. Versioning is also used when the web API or the technical environment can’t transparently handle the independent evolution of clients and servers.

You also saw main pitfalls such as coupling too many clients and servers in opposition to REST principles and recommendations such as using existing media types like HTML, Atom, or RDF built to respect REST’s hypermedia principle.

This led you to WADL and its support in the Restlet Framework. You learned how to use the `WadlApplication` class as an alternative parent class to `org.restlet.Application` in order to automatically describe the existing sample mail application. You then used the `WadlServerResource` class and saw how it complements `WadlApplication` to provide more detailed descriptions of resources and representations that constitute your API and to generate a snippet description for a single resource instead of the whole API.

Finally you saw how to convert the XML documents produced by the Restlet extension for WADL into user-friendly HTML documents that display nicely in a web browser.

The second part of this book ends with a chapter packed with Restlet recipes and best practices. Chapter 7 discusses handling of web forms, cookies, static files, error pages feeds, and redirections, as well as how to improve performance and how to modularize your Restlet application to make it ready for production rollout.



Enhancing a Restlet application with recipes and best practices

This chapter covers

- Handling common web artifacts such as forms and cookies
- How servers can redirect clients
- Handling file uploads on the client and server side
- Improving the performance of Restlet applications
- Splitting a large Restlet application into several modules

As you near the end of the second part of this book, you've already read about many important topics that required dedicated chapters, such as dealing with representations and securing and documenting your web API. But many important questions have yet to be covered—questions that still matter when you want to bring your Restlet application closer to production.

First you'll see how to deal with common Restlet developer needs, such as handling web forms, cookies, and file uploads. We also explain how Restlet can replace a classic web server by serving static files, how to customize default error pages, and how to deal with web feeds.

Then we explain how to redirect client calls, at first manually and then in a more sophisticated way with the `Redirector` class, with both client-side and server-side redirections. We continue exploring Restlet with features and best practices to improve performance such as streaming and compression of representations, caching, and conditional processing.

Finally you learn how to manage complexity with the server dispatcher, the RIAP pseudoprotocol, and internal routing to modularize your applications. For each development topic we provide small, reusable example code to copy and paste in your own application to save you time.

7.1 Handling common web elements

Even though the Restlet Framework is quite suitable for building RESTful web APIs accessible by programmatic clients, you can also use it to build regular websites.

In this section we illustrate those features and show how to manage web forms and cookies and how to serve static files without needing a separate HTTP server such as Apache HTTPD. And you learn how to deal with web feeds and how to customize error pages. Let's get started!

7.1.1 Managing forms

Web forms are a very common way to ask a user to enter or edit information using a web browser. Several popular technologies are available (such as XForms and PDF Forms) that can be used with Restlet (because they submit XML documents), but the most popular one covered here is HTML forms [5].

In this section you'll see how to present an HTML form in a browser and how to process its submission on the server side. You'll also see how a simple Restlet client can also submit forms without needing a browser. As an illustration we'll start and enhance the example started in section 4.4.1 related to FreeMarker template representations.

First you need to modify the mail template to return an HTML form instead of a read-only HTML table, as illustrated in the following listing, allowing the mail subject to be edited as a text field and the content to be edited as a text area.

Listing 7.1 Mail-editing form as a FreeMarker template

```
<html>
<head>
    <title>Example mail</title>
</head>
<body>
    <form action="?method=PUT" method="POST">
        <table>
            <tbody>
                <tr>
                    <td>Status</td>
                    <td>${status}</td>
                </tr>
            </tbody>
        </table>
    </form>
</body>
```

Submit form content
on mail resource

```

<tr>
    <td>Subject</td>
    <td><input type="text" name="subject" size="80"
               value="${subject}"></td>
</tr>
<tr>
    <td>Content</td>
    <td><textarea name="content" rows="10"
               cols="80">${content}</textarea></td>
</tr>
<tr>
    <td/>
    <td><input type="submit" value="Save"></td>
</tr>
</tbody>
</table>
</form>
</body>
</html>

```

Text field to edit mail subject

Text area to edit mail content

Save button POSTs form

Because the purpose of the form is to update the state of a mail resource, you want to use the proper HTTP PUT method—but PUT isn’t allowed by HTML 4, so you take advantage of the TunnelService (introduced in section 2.3.4) by adding a `method` query parameter.

NOTE At the time of this writing, HTML 5 is still a working draft, but after initially supporting methods other than GET and POST as valid values for the form `method` attribute (such as PUT and DELETE), the later drafts dropped this possibility for unknown reasons.

Continue modifying the original example by adding a Java method supporting PUT, as illustrated here:

```

@Put
public String store(Form form) {
    for (Parameter entry : form) {
        System.out.println(entry.getName() + " = " + entry.getValue());
    }
    return "Mail updated!";
}

```

Now start the original `MailServerApplication` class again and point your browser to `http://localhost:8111/accounts/chunkylover53-mails/123` to display the mail form. The output from Firefox is shown in figure 7.1.

Now in your browser change the subject to “Message to Jérôme,” change the content to have several lines and special characters, and click the Save button. On the server side the console will first display the raw form value and then the result of the iteration over the higher-level `Form` object, as illustrated here:

```

subject=Message to Jérôme
content=Doh!
Allo?

```

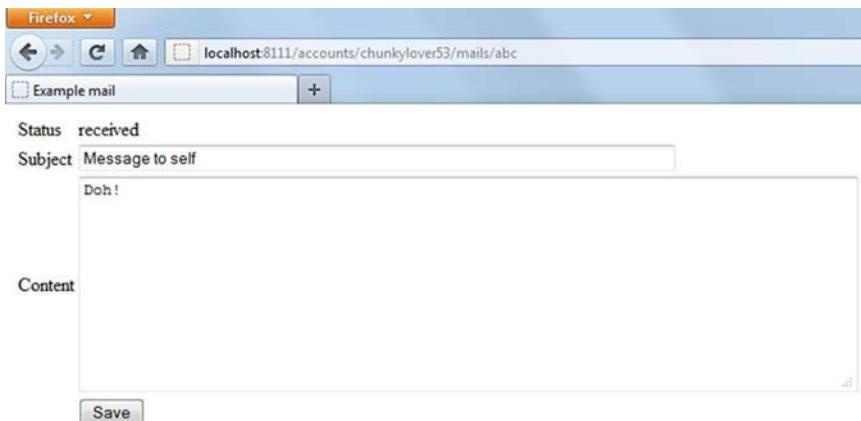


Figure 7.1 Displaying the mail-editing form

As you can see, the raw value is a long string containing a sequence of encoded form values separated by & characters. Using the `org.restlet.data.Form` allows you to automatically decode this structure and navigate it as a list of `org.restlet.data.Parameter` instances, each of which is a pair of `String`s: a parameter name and a parameter value.

There are many interesting methods in the `Form` class, such as `getValuesArray(String name)`, which returns all values with the same name in a `String` array, or the `getFirstValue(String name)` which is convenient when you assume that only one value should be available for a given parameter name.

But that's not all with the `Form` class! It can also create new forms from scratch and update them. To illustrate this, write a simple programmatic client to the `MailServerResource`, as in the following snippet:

```
public static void main(String[] args) throws Exception {
    ClientResource mailClient = new ClientResource(
        "http://localhost:8111/accounts/chunkylover53-mails/123");
    Form form = new Form();
    form.add("subject", "Message to Jérôme");
    form.add("content", "Doh!\n\nAllo?");
    System.out.println(form.getWebRepresentation());
    mailClient.put(form).write(System.out);
}
```

If you run this client, it will display the raw and encoded form sent and then the “Mail updated!” message in console. Note how you were able to pass the `Form` instance directly to the `ClientResource.put(Object)` method, which transparently relies on the `getWebRepresentation()` method and on the converter service.

Now that we've covered processing basic web forms, we explore the support for setting and reading cookies in the Restlet Framework in the next section. We come back to the topic in section 7.4 when explaining how to process file uploads.

7.1.2 Managing cookies

Even though REST discourages the use of cookies [6], there are situations where they're still needed. The most common case is for browser-based authentication, when using regular HTTP authentication isn't desired for usability reasons, or to support Web Single Sign-On (SSO) mechanisms. Other common use cases are personalization of representations and navigation tracking.

The Restlet API has extensive support for both reading cookies sent by a client on the server side and setting cookies on the server for storage on the client side, allowing control of expiration, applicable URI path, and domain name.

For this purpose it provides two classes: `org.restlet.data.Cookie` to exchange existing cookies between clients and servers, and `org.restlet.data.CookieSetting` to let servers create or update new cookies on clients. Setting a cookie requires a client to store it for a given duration and send it again to the same server in subsequent requests. It also provides two related modifiable list properties on `Request`: `cookies` and `cookieSettings`. You'll now use those classes and properties to build a cookie-based authentication mechanism.

Listing 7.2 shows a naive `NaiveCookieAuthenticator` class extending the `ChallengeAuthenticator` class (introduced in chapter 5 during the discussion of security aspects). This filter can authenticate a client based on a special cookie and challenge the client by displaying a login HTML form and intercepting its submission to set or update the cookie with an expiration time set to 30 seconds.

Listing 7.2 Simple cookie-based authentication

```
public class NaiveCookieAuthenticator extends ChallengeAuthenticator {
    public NaiveCookieAuthenticator (Context context, String realm) {
        super(context, ChallengeScheme.HTTP_COOKIE, realm);
    }
    [... similarly for other constructors ...]
    @Override
    protected int beforeHandle(Request request, Response response) {
        Cookie cookie = request.getCookies().getFirst("Credentials");
        if (cookie != null) {
            String[] credentials = cookie.getValue().split("=", 2);
            if (credentials.length == 2) {
                String identifier = credentials[0];
                String secret = credentials[1];
                request.setChallengeResponse(new ChallengeResponse(
                    ChallengeScheme.HTTP_COOKIE, identifier, secret));
            }
        }
        } else if (Method.POST.equals(request.getMethod())
            && request.getResourceRef().getQueryAsForm()
                .getFirst("login") != null) {
            Form credentials = new Form(request.getEntity());
            String identifier = credentials.getFirstValue("identifier");
        }
    }
}
```

```

        String secret = credentials.getFirstValue("secret");
        request.setChallengeResponse(new ChallengeResponse(
            ChallengeScheme.HTTP_COOKIE, identifier, secret));

        // Continue call processing to return the target
        // representation if authentication is successful
        // or a new login page
        request.setMethod(Method.GET);
    }

    return super.beforeHandle(request, response);
}

@Override
public void challenge(
    Response response, boolean stale) {
    Representation ftl = new ClientResource (
        LocalReference.createClapReference(getClass().getPackage()
            + "/Login.ftl").get());
    response.setEntity(new TemplateRepresentation(ftl, response
        .getRequest().getResourceRef(), MediaType.TEXT_HTML));
    response.setStatus(Status.CLIENT_ERROR_UNAUTHORIZED);
}

@Override
protected void afterHandle(Request request, Response response) {
    super.afterHandle(request, response);
    Cookie cookie = request.getCookies().getFirst("Credentials");

    if (request.getClientInfo().isAuthenticated() &&
        (cookie == null)) {
        String identifier
            = request.getChallengeResponse().getIdentifier();
        String secret = new String(request.getChallengeResponse()
            .getSecret());
        CookieSetting cookieSetting = new CookieSetting(
            "Credentials",
            identifier + "=" + secret);
        cookieSetting.setAccessRestricted(true);
        cookieSetting.setPath("/");
        cookieSetting.setComment(
            "Unsecured cookie-based authentication");
        cookieSetting.setMaxAge(30);
        response.getCookieSettings().add(cookieSetting);
    }
}
}

```

Set authentication cookie

Authenticate via login page/status

One of the benefits of extending `ChallengeAuthenticator` is that you can take advantage of default behavior like the use of `SecretVerifier` to check passwords. The following listing shows a portion of the updated `MailServerApplication` presented in the previous section; the `createInboundRoot()` method now creates the `NaiveCookieAuthenticator` and its associated verifier.

Listing 7.3 Guarding the mail server application

```

@Override
public Restlet createInboundRoot() {
    Router router = new Router(getContext());
    router.attach("/accounts/{accountId}/mails/{mailId}",
        MailServerResource.class);

    MapVerifier verifier = new MapVerifier();
    verifier.getLocalSecrets().put("chunkylover53", "pwd".toCharArray());

    NaiveCookieAuthenticator() authenticator = new NaiveCookieAuthenticator(
        getContext(), "Cookie Test");
    authenticator.setVerifier(verifier);
    authenticator.setNext(router);
    return authenticator;
}

```

Now launch the main method of the `MailServerApplication` and try to retrieve the previous email. The first time, you should see a login page where you can enter `chunkylover53` as the identifier and `pwd` as the password and click the Login button to effectively display the mail form. If you reload the same page immediately you should see it directly. But if you wait more than 30 seconds, the login page should be presented again because the authentication cookie expired.

If you try to launch the previous `MailClient`, you should get a 401 (Unauthorized) status in the console because authentication credentials are missing. To solve this you need to authenticate like a web browser, sending the expected `Credentials` cookie with a value of `chunkylover53(pwd)` as illustrated in the following listing.

Listing 7.4 Authenticating the mail client

```

public static void main(String[] args) throws Exception {
    ClientResource mailClient = new ClientResource(
        "http://localhost:8111/accounts/chunkylover53-mails/123");

    mailClient.getRequest().getCookies()
        .add(new Cookie("Credentials", "chunkylover53=PWD"));

    Form form = new Form();
    form.add("subject", "Message to Jérôme");
    form.add("content", "Doh!\n\nAllo?");

    mailClient.put(form).write(System.out);
}

```

If you launch this authenticating client, the “Mail updated!” message should be displayed in the console instead of the previous error, illustrating how to programmatically send cookies on the client side.

WARNING Again, this example is a naïve implementation to illustrate the use of cookies; it’s unsecure as login and password are stored in the clear. In a real application those values must be encrypted to prevent attacks. In version 2.1

of the Restlet Framework, a more robust `org.restlet.ext.crypto.CookieAuthenticator` class is available.

Now that you know how to deal with cookies, you can continue your exploration by learning how to serve static files and potentially replace an Apache HTTPD server.

7.1.3 **Serving file directories**

Even though the Restlet Framework is frequently used to expose or consume web APIs, and therefore dynamic data, you can also use it to develop any kind of web application, blurring the lines between websites and web services. You already saw those ideas in chapter 1, and now it's time to put this feature in action.

The main class supporting this feature is the `org.restlet.resource.Directory` class that can serve static files in a way similar to a regular web server but in a lighter and embeddable way. To use it you need to give it the root URI, from which it will serve the static files, and attach it to a Router that will make it visible to your clients.

The following listing creates an HTTP server exposing a web service on the `http://localhost:8111/hello` URI, as well as the directory of temporary files on your machine on the `http://localhost:8111/home/` URI.

Listing 7.5 Merging websites and web services

```
public Restlet createInboundRoot() {
    Router router = new Router(getContext());
    String rootUri = "file:///\" + System.getProperty("user.home");
    Directory directory = new Directory(getContext(), rootUri);
    directory.setListingAllowed(true);
    router.attach("/home", directory);
    router.attach("/hello",
        HelloServerResource.class);
    return router;
}
```

← Expose files
as website

← Attach hello
web service

The example sets the `listingAllowed` property to `true` in order to display the content of the directory as an HTML listing if no predefined index file is present, like a regular Apache HTTPD server would do. Figure 7.2 lists other properties you can set and methods you can override to display a customized representation of this index page.

Note also that you can change the way entries are sorted in the listing using the `comparator` property, as detailed in table 7.1, or by directly calling the `useAlphaComparator()` or `useAlphaNumComparator()` method. The latter sorting algorithm is the default, but you may prefer to revert to the former one for performance reasons.

Thanks to Restlet's pluggable client connectors, you can use any URI scheme in the `rootRef` property to retrieve and serve files from various types of locations, such as the following:

- Local file directories using the built-in FILE client
- Local JAR or ZIP files using the built-in JAR and ZIP clients
- Classpaths using the built-in CLAP client
- WAR archives using the org.restlet.ext.servlet extension
- FTP sites using the org.restlet.ext.net extension

When configuring a `Directory` it's important to keep in mind that not all these sources are born equal. For example the ClassLoader Access Protocol (CLAP) client doesn't have the ability to list the content of directories due to technical limitations. In this case turning on the directory listing by setting the `listingAllowed` Boolean property to `true` will have no effect. Because connectors such as CLAP and WAR are inherently read-only, enabling modification on a `Directory` by setting the modifiable

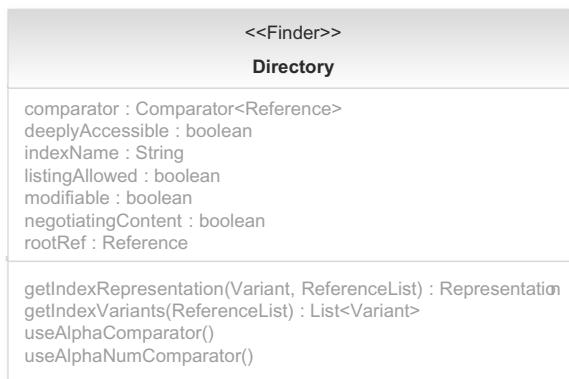


Figure 7.2 Class diagram of Directory

Table 7.1 Directory class properties

Name	Description	Default value
comparator	Comparator object used to sort URI references displayed on index pages	Friendly Alphanum algorithm based on Dave Koelle's original idea [7]
deeplyAccessible	Indicates whether the subdirectories are deeply accessible	true
indexName	The index name, without file extensions	index
listingAllowed	Indicates whether the display of directory listings is allowed when no index file is found	false
modifiable	Indicates whether modifications to local resources (most likely files) are allowed	false
negotiatingContent	Indicates whether the best file variant is automatically negotiated with the client based on its preferences	true
rootRef	The root URI from which the relative resource URIs will be looked up	Undefined

property will return an error. Speaking of errors, now you'll see how Restlet deals with them and how you can customize error pages.

7.1.4 Customizing error pages

Dealing with errors is an important part of the web developer's job. Errors can come from your own program or its dependencies, from your host machine, from a remote client or server, or from the network. One of the reasons for the Web's scalability comes from a design choice made by its creator, Tim Berners-Lee [8], to allow for broken hyperlinks and those famous 404 (Not Found) error pages.

When developing your Restlet application, you should strive to use stable URI names that aren't likely to change. You should also try to detect bugs in advance with proper testing practices (covered in detail in section 7.5), but you can't anticipate all of them, particularly those outside of your reach.

For programmatic clients it's generally sufficient to return the proper response status code, but for regular web browsers you need to be ready to present a friendlier error page. The good news is that the Restlet Framework generates such pages for you automatically when your application or component returns an error status in a response. This feature is managed by the `org.restlet.service`

`.StatusService` class and the associated `statusService` properties on `Component` and `Application`. Figure 7.3 shows the default error page displayed when a resource doesn't exist.

Let's see how you can customize this page on a per-application basis or on a per-component basis. The easiest way to customize this representation is to change the home page URI by setting the `homeRef` property on `StatusService`, but generally you want to change the whole document by overriding the `getRepresentation(Status, Request, Response)` method, as illustrated in the following listing. In this case we decided to rely on FreeMarker again to provide a template error page that's dynamically filled with data from the application and specific error status.

Listing 7.6 Changing the default status service

```
public class MailStatusService extends StatusService {

    @Override
    public Representation getRepresentation(
        Status status, Request request, Response response) {
        Map<String, String> dataModel = new TreeMap<String, String>();
        dataModel.put("applicationName",
            Application.getCurrent().getName());
        dataModel.put("statusName", response.getStatus().getName());
    }
}
```

Create
data
model

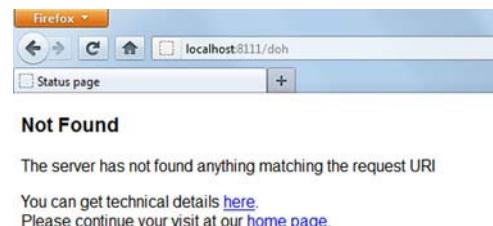


Figure 7.3 Default error page

```

dataModel.put("statusDescription", response.getStatus()
    .getDescription());
Representation mailFtl = new ClientResource (
    LocalReference.createClapReference(getClass().getPackage())
    + "/MailStatus.ftl").get();
return new TemplateRepresentation(
    mailFtl, dataModel, MediaType.TEXT_HTML);
}
}

```

Load FreeMarker template

Wrap bean with FreeMarker representation

The missing piece is the `MailStatus.ftl` template document in the following listing, which displays the application name and a logo of the Restlet Framework.

Listing 7.7 Content of the `MailStatus.ftl` template

```

<html>
<head>
    <title>Mail status page</title>
</head>

<body style="font-family: serif;">
    <p>An error was detected in the <b>${applicationName}</b>
        application.</p>
    <p><b>${statusName}</b> : ${statusDescription}</p>
    <p>
        <a href="http://www.restlet.org">
            </a>
        </p>
</body>
</html>

```

The next step is to configure this customized status service, which you achieve by updating the `MailServerApplication` constructor to set the `statusService` property, as in the following snippet:

```

public MailServerApplication () {
    setName("RESTful Mail Server application");
    setDescription("Example application for 'Restlet in Action' book");
    setOwner("Restlet SAS");
    setAuthor("The Restlet Team");
    setStatusService(new MailStatusService());
}

```

Configure status service

You can now try the invalid `http://localhost:8111/doh` URI in a browser again to observe the new error page in figure 7.4. You could go much further in term of user-friendliness, but at this point there's no special limit to your creativity.

Beyond the customization of the error representations, it's also possible to change the association between exceptions caught in Resource subclasses by the `doCatch(Throwable)` method and Status instances by overriding the `StatusService.getStatus(Throwable, UniformResource)` method. By default it returns the 500 (Internal Error) status, but you could define a `NotFoundException` class that would map to the

404 status. Note that in that case, you still have access to the underlying exception via the `Status.getThrowable()` method, so you could easily display more information about the cause of the error status.

The next section covers advanced forms that need to upload files from an HTML page and retrieve those files on the server side.

7.1.5 Handling file uploads

When you need to upload a file from a web browser to a web application, your HTML form uses a special `multipart/form-data` media type. This changes the way your form content is encoded so that on the server side you can't use the `org.restlet.data.Form` class anymore; it only supports the `application/x-www-form-urlencoded` media type that you saw in the first section.

Currently, the Restlet Framework relies on an `org.restlet.ext.fileupload` extension to parse multipart forms on the server side. This extension depends on the Apache Commons FileUpload library (<http://commons.apache.org/fileupload/>) provided in the `/lib` directory of your Restlet Framework installation, along with two additional dependencies (`org.apache.commons.io` and `javax.servlet`).

To illustrate the use of this feature, you'll enhance the mail-editing form developed in section 7.1.2 to allow the upload of an attachment file. First modify the mail form template to change the form encoding and add the file upload field, as in the following listing.

Listing 7.8 Adding a file attachment field to the mail-editing form

```
<html>
<head>
<title>Example mail</title>
</head>
<body>
    <form action="?method=PUT" method="POST" enctype="multipart/form-data">
        <table>
            <tbody>
                <tr>
                    <td>Status</td>
                    <td>${status}</td>
                </tr>
                <tr>
                    <td>Subject</td>
                    <td><input type="text" name="subject" size="80" value="${subject}"></td>
                </tr>
                <tr>
                    <td>Content</td>
                    <td><textarea name="content" ></td>
                </tr>
            </tbody>
        </table>
    </form>
</body>
```

Added multipart encoding

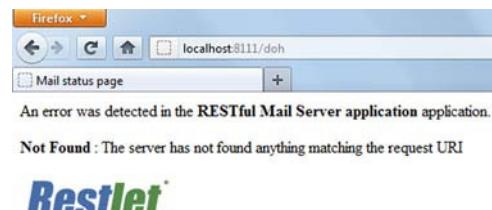


Figure 7.4 Customized error page

```

        rows="10" cols="80">>${content}</textarea></td>
    </tr>
    <tr>
        <td>Attachment</td>
        <td><input
            name="attachment" type="file"/></td>
    </tr>
    <tr>
        <td/>
        <td><input type="submit" value="Save"></td>
    </tr>
</tbody>
</table>
</form>
</body>
</html>
```



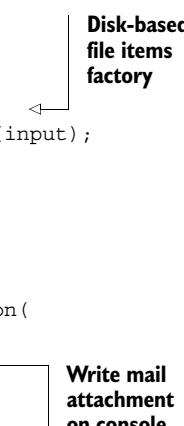
The next step is to update the PUT handling logic in the MailServerResource class. The important thing to note in listing 7.9 is that you need to instantiate an org.restlet.ext.fileupload.RestletFileUpload class and retrieve from it a list of form fields. It's then easy to detect whether a field is a regular form field, like the subject and content fields, or a file upload field that should have a special treatment.

Listing 7.9 Adding a file attachment field to the mail-editing form

```

@Put
public String store(Representation input) throws Exception {
    RestletFileUpload fileUpload = new RestletFileUpload(
        new DiskFileItemFactory());
    List<FileItem> fileItems = fileUpload.parseRepresentation(input);

    for (FileItem fileItem : fileItems) {
        if (fileItem.isFormField()) {
            System.out.println(fileItem.getFieldName() + "="
                + fileItem.getString());
        } else {
            Representation attachment = new InputRepresentation(
                fileItem.getInputStream());
            attachment.write(System.out);
        }
    }
    return "Mail updated!";
}
```



We encourage you to try this example and observe that the console still displays the mail subject and content fields, plus the uploaded attachment file. That was easy! Note also that Apache FileUpload offers various strategies to parse this multipart form, including a streaming API for maximum efficiency that can be accessed via the RestletFileUpload.getItemIterator(Representation) method. For additional information, look at the FileUpload library documentation.

Let's now continue with the second section of the chapter—web feeds.

7.2 Dealing with Atom and RSS feeds

Exposing or consuming web feeds has become a common requirement for web applications, and the Restlet Framework has comprehensive support for them, through two extensions.

The first is the `org.restlet.ext.atom` extension, which provides comprehensive Atom and AtomPub support with dependency only on the Restlet core and on the `org.restlet.ext.xml` extension for XML parsing and writing support. This is a lightweight implementation (about 80 Kb in size, dependencies included) integrating naturally with the Restlet API. If you only require Atom support in your applications, either client side or server side, this is the way to go.

The second extension is `org.restlet.ext.rome`, which integrates with the popular ROME open source library that's heavier (350 Kb in size, dependencies included) but has the ability to parse and write all feed formats, including RSS and Atom, with a single API. If you can't solely rely on Atom and AtomPub, this is the extension to use.

7.2.1 Exposing web feeds

Let's illustrate those extensions with the RESTful mail example by implementing a feed resource associated with an account that exposes its data in both Atom 1.0 (via the Atom extension) and RSS 2.0 (via the ROME extension), so you can compare them side by side. The source code of the server resource is in the following listing.

Listing 7.10 Account feed server resource

```
public class FeedServerResource extends ServerResource {

    @Get("atom")
    public Feed toAtom() throws ResourceException {
        Feed result = new Feed();
        result.setTitle(new Text("Homer's feed"));

        for (int i = 1; i < 11; i++) {
            Entry entry = new Entry();
            entry.setTitle(new Text("Mail #" + i));
            entry.setSummary("Doh! This is the content of mail #" + i);
            result.getEntries().add(entry);
        }

        return result;
    }

    @Get("rss")
    public SyndFeed toRss() throws ResourceException {
        SyndFeed result = new SyndFeedImpl();
        result.setTitle("Homer's feed");
        result.setDescription("Homer's feed");
        result.setLink(getReference().toString());
        List<SyndEntry> entries = new ArrayList<SyndEntry>();
        result.setEntries(entries);

        for (int i = 1; i < 11; i++) {
            SyndEntry entry = new SyndEntryImpl();
            entry.setTitle("Mail #" + i);
            entry.setSummary("Doh! This is the content of mail #" + i);
            entries.add(entry);
        }
    }
}
```

The diagram shows two callout arrows pointing to specific parts of the code. One arrow points to the `@Get("atom")` annotation and the `toAtom()` method, with the text "Atom extension" next to it. Another arrow points to the `@Get("rss")` annotation and the `toRss()` method, with the text "ROME extension" next to it.

```

        entry.setTitle("Mail #" + i);
        SyndDescription description = new SyndContentImpl();
        description.setValue("Doh! This is the content of mail #"+i);
        entry.setDescription(description);
        entries.add(entry);
    }

    return result;
}
}

```

Both representation variants are built similarly; the Atom extension is more compact because there's no separation between Java API interfaces and the implementation classes as there is in ROME.

If you attach this resource to the application's inbound router at the `/accounts/{accountId}/feeds/{feedId}` URI template, you can test them from your web browser. To compare the RSS and Atom variants, you can take advantage of the tunnel service by adding a `?media=rss` or `?media=atom` query string at the end of the URI. Figure 7.5 shows the result obtained in Firefox. The browser recognizes the media type set by Restlet and displays the feed.

Those feed resource implementations are quite minimal, but the extensions provide comprehensive APIs supporting most subtleties encountered with more complex feeds, so you should be pretty safe moving forward on those foundations.

Because modern web browsers tend to provide a nice rendering of web feeds, in either Atom or RSS media types, let's try another way to obtain raw representation using the command line curl or wget tools. Here's raw output obtained:

```

~$ curl http://localhost:8111/accounts/chunkylover53/feeds/xyz?media=atom
<?xml version="1.0" standalone='yes'?>
<feed xmlns="http://www.w3.org/2005/Atom">
    <title type="text">Homer's feed</title>
    <entry>
        <summary>Doh! This is the content of mail #1</summary>
        <title type="text">Mail #1</title>
    </entry>
    ...
</feed>

```

Serving feeds was pretty easy, but what about the client side?

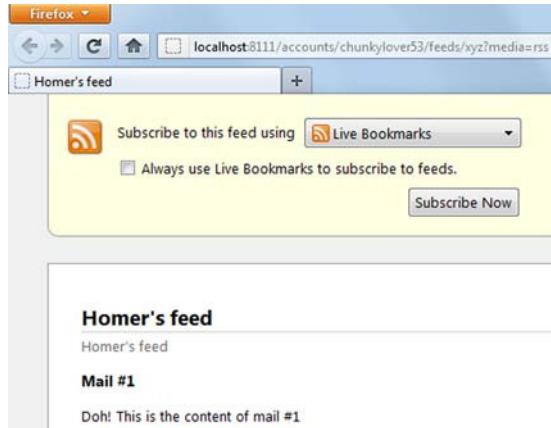


Figure 7.5 Retrieving Restlet web feeds

7.2.2 Consuming web feeds

Unsurprisingly, consumption of Atom and RSS feeds is equally easy in Restlet and relies on the same classes in both cases, as illustrated in the following listing.

Listing 7.11 Consuming account feeds

```
public static void main(String[] args) throws Exception {
    ClientResource mailClient = new ClientResource(
        "http://localhost:8111/accounts/chunkylover53/feeds/xyz");

    Feed atomFeed = mailClient.get(Feed.class);
    System.out.println("\nAtom feed: " + atomFeed.getTitle() + "\n");

    for (Entry entry : atomFeed.getEntries())
    {
        System.out.println("Title : " + entry.getTitle());
        System.out.println("Summary: " + entry.getSummary());
    }

    SyndFeed rssFeed = mailClient.get(SyndFeed.class);
    System.out.println("\nRSS feed: " + rssFeed.getTitle() + "\n");

    @SuppressWarnings("unchecked")
    List<SyndEntry> entries = (List<SyndEntry>) rssFeed.getEntries();

    for (SyndEntry entry : entries)
    {
        System.out.println("Title : " + entry.getTitle());
        System.out.println("Summary: " +
                           entry.getDescription().getValue());
    }
}
```

Again you can see the power and simplicity of the converter service and its ability to automatically negotiate the best representation variant based on the Java type passed to the `ClientResource.get(Class<T>)` method.

As a final note on this topic, keep in mind that using Java APIs like the Atom and ROME extensions gives a very convenient abstraction to expose feeds but isn't always the best way to achieve this task compared to template representations. There's more to say about feeds—including how they're used by higher level web APIs such as OData (see the dedicated Restlet extension available since version 2.0) and GData—but we need to continue exploring Restlet best practices with a section on web redirection followed by optimization and modularization advice.

7.3 Redirecting client calls

Redirections are an essential feature when developing web applications. They can be divided into two categories: client-side redirections and server-side redirections. Client-side redirections are useful to point a client to the new location of a resource, to a fallback resource when the target one has been deleted, or to the original target resource after authenticating the user with a single sign-on mechanism such as OpenID.

Server-side redirections are transparent to the client making the call, but facilitate the internal processing either by delegating to another resource within the component

(*inbound server redirection*), or by delegating to a resource outside the component (*outbound server redirection*, often referred to as a *reverse proxy*).

The Restlet Framework has comprehensive support for all of these cases. First we cover how redirection can be handled manually, and then we introduce the powerful Redirector class.

7.3.1 Manual redirection

Client-side redirections use dedicated HTTP status codes. Those codes are in the Restlet API with their equivalent constants in the org.restlet.data.Status class and set with the org.restlet.Response.status property. For more details using status codes when designing your web API, refer to appendix D.

When writing a server-side resource, you can call the various ServerResource.setStatus (...) methods directly or use the following shortcut methods: redirectPermanent(targetUri), redirectSeeOther(targetUri), and redirectTemporary(targetUri). Those methods accept absolute and relative URIs; relative URIs are resolved against the current base reference of the resource.

HTML redirections

Note that when clients are web browsers, you can also implement client-side redirection by adding an HTML META tag in a document head like this: <meta http-equiv="refresh" content="5; URL=http://my.targetLocation.com">

In this example, the result page will be displayed for five seconds before redirecting the browser to the target URI. Be warned, though, that this approach is outside HTTP and can therefore be overlooked by some clients, including programmatic ones. It might also have a negative impact on search engine referencing and should be used sparingly.

Let's assume that you have an application that contains an old REST resource that was replaced by a newer one at a different URI and even a different HTTP port. You'd like to ensure that existing clients don't receive a 404 status when attempting to retrieve the old resource and at the same time would like to inform them that there's a new location that should be used, updating previous bookmarks it may have. The following listing contains the code of the old resource doing the permanent redirection.

Listing 7.12 Redirecting clients to a new permanent location

```
public class OldServerResource extends ServerResource {
    @Get
    public String redirect() {
        redirectPermanent("http://localhost:8111/");
        System.out.println("Redirecting client to new location...");
        return "Resource moved... \n";
    }
}
```

Set response status to 301

Add explanation entity

The program in the next listing serves both the old resource from listing 7.12 on port 8183 and the new resource (`HelloServerResource`) on port 8111.

Listing 7.13 Server program listing hosting the old and new resource

```
public class RedirectingServer {
    public static void main(String[] args) throws Exception {
        new Server(Protocol.HTTP, 8111,
                   HelloServerResource.class).start();
        new Server(Protocol.HTTP, 8113,
                   OldServerResource.class).start();
    }
}
```

The final step in this example is to write a client program that tries to reach the old resource and verify that the `ClientResource` class and its `followRedirects` feature work as expected, retrieving the result of the new resource. In the following code snippet you can see how simple it is to write such a client:

```
public class RedirectedClient {
    public static void main(String[] args) throws Exception {
        ClientResource resource
            = new ClientResource ("http://localhost:8113/");
        resource.get().write(System.out);
    }
}
```

If you launch this `RedirectedClient` class, you'll observe that the client console displays the “hello, world” string. At the same time, the server console displays the “Redirecting client to new location...” message, proving that it worked as expected. Finally, you can open `http://localhost:8113/` in a web browser to observe the same result, including the change of URI in the address bar.

Even if writing a redirecting server is easy for a single resource, you can guess how tedious it can be to redirect a large set of resources—for example, if a complete application migrated from one domain name to another. In the next section we explain how the Restlet Framework can help solve this use case.

7.3.2 *The org.restlet.Redirector class*

As introduced earlier, the `Redirector` class is a special Restlet subclass that redirects all requests reaching it in two modes: client-side or server-side redirections.

First it can issue for you a response with a client-side redirection, exactly as explained in the previous section, with an additional feature: the target URI reference provided can be a URI template where variables are replaced automatically for you. For example, you can append the URI remaining part (relative to the current base URI) to your target URI, by using this target template: `http://my.targetDomain.com/{rr}`. Table 7.2 lists a few common variables.

Table 7.2 Common URI template variables

Variable name	Description
cia	IP address of the client (<code>request.clientInfo.address</code>)
cig	User agent name (<code>request.clientInfo.agent</code>)
m	Method (<code>request.method</code>)
rr	Remaining part of the URI reference (<code>request.reference.remainingPart</code>)

To obtain a comprehensive list of supported variables, check the Javadocs of the `org.restlet.util.Resolver<T>` class, which is used to wrap a request and response pair as a template data model.

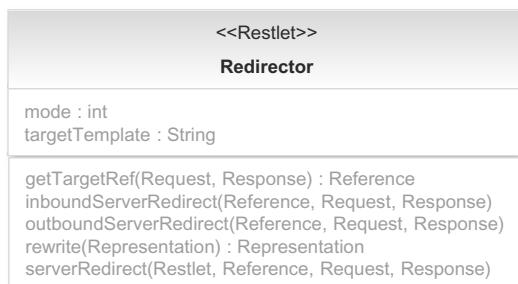
This URI template mechanism lets you easily redirect a subset of your URI space to a new location. It can also be used in the second server-side redirection mode. In this mode, the redirection is transparent from the client point of view and handled entirely on the server side while the user is waiting for a response.

We complete the discussion of the `Redirector` class with the class diagram in figure 7.6.

As you can see, there are two properties: `targetTemplate` for the URI template to redirect to and `mode` to indicate the redirection mode to use with a value in this list of constants:

- `MODE_CLIENT_FOUND` :—Client-side mode for `Status.REDIRECTION_FOUND`
- `MODE_CLIENT_SEE_OTHER` :—Client-side mode for `Status.REDIRECTION_SEE_OTHER`
- `MODE_CLIENT_TEMPORARY` :—Client-side mode for `Status.REDIRECTION_TEMPORARY`
- `MODE_SERVER_OUTBOUND` :—Server-side mode for application’s `outboundRoot` or context’s `clientDispatcher` (using client connectors)
- `MODE_SERVER_INBOUND` :—Server-side mode for context’s `serverDispatcher`

The server outbound mode allows you to define a reverse proxy transferring requests to a remote service, typically by rewriting the request’s URI. The inbound mode is valuable when trying to modularize a large application, as we discuss in section 7.4.

**Figure 7.6 Redirector class diagram**

The `rewrite(Representation)` method is invoked for the server-side redirections. By default it does nothing, but you can override it to provide your own logic—such as to rewrite hyperlinks for response entities received from remote applications.

Let's see an example of this class in action. The following listing provides a search resource relying on Google search to illustrate two things: how to use a target URI template to insert a dynamic variable (the keywords to search) and how to extract an attribute automatically from a request's URI query parameter.

Listing 7.14 Client-side redirection with attribute extraction

```
public class SearchRedirector extends Application {
    public static void main(String[] args) throws Exception {
        Component component = new Component();
        component.getServers().add(Protocol.HTTP, 8111);
        Application application = new SearchRedirector();
        component.getDefaultHost().attachDefault(application);
        component.start();
    }

    @Override
    public Restlet createInboundRoot() {
        Router router = new Router(getContext());
        String target
            = "http://www.google.com/search?q=site:mysite.org+{keywords}";
        Redirector redirector = new Redirector(getContext(), target,
            Redirector.MODE_CLIENT_TEMPORARY);

        Extractor extractor = new Extractor(getContext(), redirector);
        extractor.extractFromQuery("keywords", "kwd", true);

        router.attach("/search", extractor);
        return router;
    }
}
```

As you can see, you use `Extractor`, another class from the Restlet routing system, to automatically add a request attribute by extracting a URI query parameter named `kwd`. Note that this class also works with cookies and posted web forms via additional `extractFrom*()` methods.

Launch the search redirector program and test it by entering the following URI in your web browser: `http://localhost:8111/search?kwd=book`. You should see a redirection to the Google search page with the following search string: `"site:mysite.org book."`

Adding new resources and new features such as web feeds and forms is important to cover the requirements of your application, but this shouldn't be your sole enhancement focus. The next section continues your exploration of best practices with a discussion of Restlet application performance and how to improve it.

7.4 Improving performances

Performance improvement of web applications is a vast topic that would require at least one complete book to do it justice. Our goal here is to explore six powerful features of REST, HTTP, and the Restlet Framework to help you improve the latency and system resource consumption of your Restlet application.

We first explain how to stream large representations that may not fit in available memory (for example, a multigigabyte file) or would consume too much memory under load. Then you see how compression is a simple and powerful way to save bandwidth and reduce latency. We finish with four additional ways to improve performance.

7.4.1 Streaming representations

In most use cases, HTTP requests and responses must contain a header that specifies the length in bytes of the message body, if there is one (note that this length doesn't take into account headers or other control information such as the status code). For instance, a server that answers a request with the ASCII character string "hello, world" will include the following header in its response, as the string is composed of 12 characters:

```
Content-Length: 12
```

Doing this requires knowing the length of the data before you start sending it. But that's not always the case; for example, the data can be generated dynamically and you might not know its length up front. You could generate all the data before starting to send it, because that would allow you to know its size and fill in the `Content-Length` header correctly (or its equivalent `Representation.size` property in the Restlet API).

But if the data is large or takes a long time to generate, you might want to start sending it before it's entirely generated. Streaming data like that is made possible by TCP, which is usually the transport protocol you use for sending HTTP requests and responses over the network. When a TCP connection is established between the client and the server, data can be sent over that connection piece by piece, progressively.

In such situations, to solve the unknown `Content-Length` problem, you can use a technique called content streaming. Instead of indicating the entire content length, you specify that the entity will contain a series of chunks, using a `Transfer-Encoding: chunked` header. Each chunk specifies its own size, and the end of the series is marked with a zero-sized chunk.

When enough data is ready to be sent, you put it in a chunk that you send over the network. That way the recipient can process it while you're generating other chunks. As you can imagine, this ability for HTTP components to easily exchange and process data progressively as it becomes available can be very beneficial to the performance and responsiveness of your applications. Let's now see how the Restlet API supports this.

CONTENT STREAMING EXAMPLE WITH RESTLET

Using the Restlet Framework, performing content streaming is easy. You still need to prepare a representation, as usual. The following listing illustrates the generation of such dynamic content.

Listing 7.15 Server Resource that generates dynamic content

```

public class DynamicContentServerResource extends ServerResource {
    @Get
    public Representation getDynamicContent() {
        Representation result = new WriterRepresentation (
            MediaType.TEXT_PLAIN) {
            @Override
            public void write(Writer writer) throws IOException {
                for (int i = 0; i < 10000; i++) {
                    writer.append("0123456789\n");
                }
            }
        };
        return result;
    }
}

```

The code defines a `DynamicContentServerResource` class that extends `ServerResource`. It overrides the `getDynamicContent()` method to return a `WriterRepresentation` object. This representation is initialized with `MediaType.TEXT_PLAIN`. The `write(Writer)` method is overridden to append 10,000 digits from 0 to 9 to the writer, separated by newlines.

The Restlet Framework provides a broad set of `Representation` subclasses dedicated to common usages. The `WriterRepresentation` is an abstract class that handles the generation of dynamic and potentially large content, ensuring that the chunked encoding is automatically used to communicate with HTTP clients. Subclasses are only required to override the `write(Writer)` method where they can implement the logic of writing the representation's content of any length.

7.4.2 Compressing representations

Another great feature of HTTP that isn't widely used is its ability to compress the content of entities exchanged between components. This feature relies on the `Accept-Encoding` and `Content-Encoding` HTTP headers and the `Representation.encodings` and `ClientInfo.acceptedEncodings` properties of the Restlet API.

Even though some entities—such as most image, audio, and video formats—are already compressed, the average win for textual entities is 70%. The only condition is that both clients and servers support the same encoding algorithm, which is increasingly common, especially for the GZip algorithm.

Since version 2.1, the Restlet Framework is fully capable of automatically compressing and uncompressing entities exchanged with other remote components. This support comes with the `DecoderService` and `EncoderService` properties available to applications. By default the decoding of compressed message entities received is enabled, but encoding is disabled. To turn it on for a Restlet application, call `getEncoderService().setEnabled(true)` in the application constructor.

Compression is configurable through a list of accepted media types (all types by default), combined with a list of ignored media types (archive, audio, image, and video by default). You can also set a minimum size (default value of 1000 bytes) to prevent compression of small entities, where the benefit is small compared to the extra overhead in CPU, memory, and latency.

We recommend you turn on this service in production after proper testing with representative clients. During development the lack of entity readability due to compression can be a drawback when debugging, though. Next we show the importance of partial representations and the RangeService to manipulate large entities.

7.4.3 Partial representations

Suppose that, while receiving the representation of some resource in response to a GET request, your client application suffers a temporary network problem that drops the connection to the server. When the network is back, if the representation is large, wouldn't it be beneficial to be able to request only the missing part of the representation instead of restarting from the beginning?

Or suppose that your client application, for whatever reason, only needs a portion of a particular representation. Wouldn't it be nice to be able to ask the server to send only the needed portion? For large data sets, this can reduce network traffic local storage capacity requirements and improve performance.

For such situations, HTTP offers a Range header to specify the desired range(s) to return in a response, expressed in bytes. Optionally, on the server side, an application can also advertise its support for serving partial content, on a per-resource basis, by including Accept-Ranges headers in response to regular requests. The possible values associated with this header are the names of the units in which ranges can be specified (for example, *bytes*) or the word *none* to express that partial content delivery isn't supported on the target resource. The latter is useful to advise the client not to attempt a range request.

As you can imagine, this ability to exchange partial content is another powerful feature of HTTP, so consider using it when implementing your applications. Let's illustrate range usage with the HelloServerResource class and a client that wants the first five characters of the "hello, world" sentence—that is to say, *hello*. The following listing shows a client that asks for the first five characters of the remote resource.

Listing 7.16 Requesting parts of a resource

```
public static void main(String[] args) throws Exception {  
    ClientResource resource = new ClientResource("http://localhost:8111/");  
    resource.getRanges().add(new Range(0, 5));  
    resource.get().write(System.out);  
}
```

Request first
5 characters

Because the server supports the range feature, a client can obtain a specific portion of a resource's representation using the ranges property, which maps to the Range header. Note that if you ran the same client after launching the DynamicContentServer of the previous section, the client would still receive the full representation. This is merely due to the fact that this server isn't using a Restlet application as a container for its resource, thereby missing the RangeService providing the partial representation processing.

Let's continue exploring HTTP performance optimization mechanisms with caching. For this purpose we first explain how to set cache information on the server side, then how to use conditional methods to revalidate cached responses.

7.4.4 Setting cache information

The purpose of caching is to allow a client or an intermediary, under certain conditions, to keep previous responses sent by the server and reuse them instead of performing real requests over the network. There are two main benefits: reduced latency (the time to wait for the response) and reduced network traffic.

HTTP provides powerful support for caching. Making use of it requires some cooperation from your application because what is cacheable and what isn't is specific to each application (and potentially each resource that it contains). In addition, to benefit from caching, your clients must manage the cached representations. Fortunately most HTTP client libraries, including web browsers, provide automatic cache management.

When a client issues a GET request, it can look in its local cache to see if this request has previously been answered by the server and stored in the cache. If this is the case, and if the cached response is fresh enough, the cached response will be used, without the need for network interactions with the server.

If the cached response appears to be stale, the cache can try to revalidate the response by sending a request to the server and letting it know it already has a cached response. If the server determines that the cached response held by the client is fresh enough, it can avoid further processing and inform the client that its cached response is still fresh. Finally, you can choose to allow certain cached responses to be handed to the client even if they're stale. This can be useful as a fallback mechanism if the client is disconnected from the server.

The way all this works is specified by the HTTP protocol. A server can add an Expires header to a response; this element contains a date and time after which the response should be considered stale. Usually, it's application-dependent. For example, if the state of a given resource changes every day at midnight, the server can return an Expires header set to midnight, along with the resource representation, when serving a GET request on that resource. The client's cache can then use this information in subsequent requests for that resource to determine whether a response it holds is fresh enough to be returned to the client.

A number of additional mechanisms are defined by HTTP to control the caching strategy—for example, a response that must not be cached can be tagged with a Cache-control header with a no-cache value. For in-depth coverage of HTTP caching, we recommend the excellent online tutorial written by Mark Nottingham [9].

When implementing a RESTful system, you should consider making use of these caching mechanisms. You should specify information about the cacheability of responses emitted by your server-side application. The Restlet API has integrated support for all these headers via the following properties:

- `RepresentationInfo.tag` :—E-Tag-based validation
- `RepresentationInfo.modificationDate` :—Date-based validation
- `Representation.expirationDate` :—Date-based cache control
- `Message.cacheDirectives` :—Advanced cache control

We can't illustrate every kind of caching mechanism available, because that's beyond the scope of this book. Instead we focus on a simple way for a server resource to help clients such as browsers optimize their requests.

In listing 7.17, the resource updates its representation with a modification date, an expiration date, and an entity tag (E-Tag). This cache information can then be used by web browsers and intermediary caches such as Squid (www.squid-cache.org) to prevent or shortcut future requests to the same resource.

Listing 7.17 Server resource that time-stamps its representation

```
public class CachingServerResource extends ServerResource {
    @Get
    public Representation represent() {
        Calendar cal = new GregorianCalendar(2012, 4, 17, 10, 10, 10);
        Representation result = new StringRepresentation("<a href="
            + getReference() + ">" +
            + System.currentTimeMillis() + "</a>");
```

The code snippet shows annotations for setting cache metadata. A callout points to the line `result.setModificationDate(cal.getTime());` with the text "Set last modification date". Another callout points to the line `result.setExpirationDate(cal.getTime());` with the text "Expires 3 hours later". A third callout points to the line `result.setTag(new Tag("xyz123"));` with the text "Set E-Tag unique value". A fourth callout points to the line `getResponse().getCacheDirectives().add(CacheDirective.publicInfo());` with the text "Set one cache directive".

```
        result.setMediaType(MediaType.TEXT_HTML);
        result.setModificationDate(cal.getTime());
```

```
        cal.roll(Calendar.HOUR, 3);
        result.setExpirationDate(cal.getTime());
```

```
        result.setTag(new Tag("xyz123"));
        getResponse().getCacheDirectives().add(
            CacheDirective.publicInfo());
        return result;
    }
}
```

Note that if you launch this server and try to access it from a web browser, you won't see the browser cache in action because browsers bypass the cache for localhost sites. You need to deploy the server under a real domain name. As an alternative, verify that this information is available on the client side, as shown in the following listing.

Listing 7.18 Retrieving caching metadata

```
public static void main(String[] args) throws Exception {
    ClientResource resource = new ClientResource("http://localhost:8111/");
    Representation rep = resource.get();
```

The code snippet shows annotations for retrieving cache metadata. A callout points to the line `Representation rep = resource.get();` with the text "Get a representation". Another callout points to the lines `System.out.println("Modified: " + rep.getModificationDate());` and `System.out.println("Expires: " + rep.getExpirationDate());` with the text "Display caching metadata".

```
    System.out.println("Modified: " + rep.getModificationDate());
    System.out.println("Expires: " + rep.getExpirationDate());
    System.out.println("E-Tag: " + rep.getTag());
}
```

In this particular example, on the server side the resource always generates the same representation, so its modification date remains unchanged. But in real cases those values would be adjusted based on the resource lifecycle, including its state changes.

Next you'll see how you can use conditional methods to revalidate cached responses and prevent the lost update problem.

7.4.5 Conditional methods

Sometimes a client wants to ask the server to process a request only if certain conditions are met. HTTP supports headers that allow conditional method processing: If-Match, If-Modified-Since, If-None-Match, and If-Unmodified-Since.

A client might want to retrieve the state of a resource (with a GET method), do some processing, and then update the state on the server (with a PUT method) only if the resource state has not changed in the meantime—for example, after concurrent editing by another client. To accomplish this, the client’s PUT request should include an If-Match header specifying the entity tag included in the response to the previous GET request. An E-Tag is a character string generated by the server that allows it to determine whether two representations of a given resource are equivalent.

CONDITIONAL METHOD EXAMPLE

The following code snippet adds a PUT method to the previous CachingServer-Resource. The implementation displays a message on the server console:

```
@Put
public void store(Representation entity) {
    System.out.println("Storing a new entity.");
}
```

On the client side, in listing 7.19 you retrieve the original representation and do a conditional GET based on the modification date and E-Tag. Then you do two conditional PUTs, one with a matching E-Tag and then one with a nonmatching E-Tag, to demonstrate that it produces an error due to a failed condition.

Listing 7.19 Updating a resource with conditions

```
public class ConditionalClient {

    public static void main(String[] args) throws Exception {
        ClientResource resource
            = new ClientResource ("http://localhost:8111/");
        Representation rep = resource.get();
        System.out.println(resource.getStatus()); ← Get representation

        resource.getConditions().setModifiedSince(
            rep.getModificationDate());
        rep = resource.get();
        System.out.println(resource.getStatus()); ← Get updated representation if modified

        resource.getConditions().setModifiedSince(null);
        resource.getConditions().getNoneMatch().add(new Tag("xyz123"));
        rep = resource.get();
        System.out.println(resource.getStatus()); ← Put new representation with same tag

        resource.getConditions().getNoneMatch().clear();
        resource.getConditions().getMatch().add(rep.getTag());
        resource.put(rep);
        System.out.println(resource.getStatus()); ← Put new representation with different tag

        resource.getConditions().getMatch().clear();
    }
}
```

```
        resource.getConditions().getMatch().add(new Tag("abcd7890"));
        resource.put(rep);
        System.out.println(resource.getStatus());
    }
}
```

Running this client obtains the following output in the console:

```
OK (200) - OK
Not Modified (304) - Not Modified
Not Modified (304) - Not Modified
OK (200) - OK
Exception in thread "main" Precondition Failed (412) - Precondition failed
```

TIP The last line is quite interesting because the framework automatically prevented an update of the resource due to the (simulated) detection of a change in the representation E-Tag. Because this condition isn't met, the PUT request was ignored by the `ServerResource` class. This is a great way to prevent the lost update problem that occurs when several concurrent clients update the same resource concurrently without previously acquiring locks.

Now we describe a last performance best practice, directly inherited from the REST architecture style: living without sessions.

7.4.6 **Removing server-side session state**

In many web applications, including those developed with Java Servlet technology, there's a notion of server-side session used to maintain conversational state between clients and servers. This session is typically identified uniquely by a cookie or a query parameter. For each request received, the server is then able to look up the correct session and update it if necessary (for example, to maintain a shopping cart). At first glance this looks like a convenient feature for developers.

Unfortunately, this leads to numerous problems such as reduced scalability as servers are encumbered by in-memory data. Also, client requests must be processed by servers that contain or have access to the associated session state (also known as *session affinity*) when load balancing becomes necessary—otherwise constant session replication must be used. Reliability is also harder to achieve because session state potentially has to be reliably stored and restored in the case of server failure. Finally, this prevents the HTTP infrastructure and network intermediaries from providing some services because a request can no longer be understood in isolation from this out-of-band conversation.

From an HTTP point of view, each request should contain all the information needed by the server to process it without having to refer to out-of-band information. The Restlet Framework helps you enforce this rule by *not* providing any server-side session mechanism in its Restlet API. If you find yourself in desperate need of Servlet-like sessions, you usually need to rethink your design more RESTfully, relying on the two approaches discussed next, separately or in conjunction.

The first solution is to store this conversational state on the client side, as hypermedia information, cookies, JavaScript logic, HTML 5 offline databases, and so on. Often

people talk about the stateless principle of REST, implying that there can't be any session state between clients and servers. This isn't true; session state *can* be kept on the client side. But the client should send the relevant state along with each request to the server. One way to achieve this is by using the HATEOAS pattern (Hypermedia As The Engine Of Application State) discussed in chapter 10, where possible state transitions are encoded inside the hypermedia representations exchanged between the server and the client. Another way is to develop rich clients with technologies such as GWT (see chapter 9 for details), Flex, or Silverlight.

The second solution is to convert this conversational state into first-class resources in your web API. You could have a caddie resource with its own URI and HTTP methods to manipulate it. Its state could be persisted in a database for a rather long time (not the usual 30 minutes of a Servlet session) and let the user continue shopping a few days later. The scalability issue doesn't disappear in this case but is transformed into a well-understood problem that can be solved using database sharding, replication, or cloud-scale databases (for example NoSQL). The advantage of this approach is that it puts low requirements on the clients, which is important for mobile devices.

You are now done with your review of Restlet best practices for improving performance. Before ending this chapter and the second part of the book, we'd like to wrap things up by consolidating pieces of the example mail component and address its slowly growing complexity by introducing solutions to modularize large Restlet applications.

7.5 **Modularizing large applications**

When we introduced Restlet components in chapter 3, you saw how they can act as containers of several applications. This is the most obvious way to modularize your web presence, creating more or less isolated functional spaces. The advantage is that you can later decide to deploy those applications separately, on different host machines for example. But when an individual application is becoming in itself too complex, maybe due to a growing list of contained resources, you can split it into several applications, still hosting them in the same component and benefiting from intra-JVM communication.

In this section you'll see that Restlet provides a mechanism, similar to Servlet's request dispatcher, called the server dispatcher. Then we introduce the more powerful RIAP pseudoprotocol, inspired by Apache Cocoon. We illustrate how to use RIAP to separate a large application into two parts—a public part and a private part not reachable by clients outside of the current component.

7.5.1 **Server dispatcher**

If you're a Servlet developer, there's a good chance you've already used the Servlet context's request dispatcher to include the result of another Servlet or to forward call processing. The Restlet API comes with a similar mechanism called the server dispatcher,

which works like the client dispatcher but sends requests to the current component as if they came from the outside instead of asking server connectors to process them.

This object can be retrieved from the current context using the `serverDispatcher` property, which is an instance of the `Restlet` class like `clientDispatcher`. To invoke it you need to create a new `Request` object with the proper target resource URI reference and method set. If your local target resource is protected, you need to properly authenticate yourself as there won't be much difference between your call and an outsider call. You can also use `ClientResource` as long as you set the `serverDispatcher` as the value for the next property.

To illustrate these advanced concepts, go back to the example mail server component. In previous chapters you've developed a web application mixing web API aspects and website aspects, with bean representations and annotated interfaces for the former one and template HTML representations for the latter. Although this approach works fine when security requirements and resource granularity are the same, there are cases where it's wiser to split this application into two complementary applications: a web API layer at the bottom exposing the domain data into a reusable RESTful API, and a website layer picking up, splitting, or aggregating various resources from one or several RESTful APIs underneath. In your case, this results in two Restlet applications, as illustrated in figure 7.7: `MailSiteApplication` and `MailApiApplication`. At the same time, you consolidate in those applications many features introduced in the first two parts of the book. We won't review all the resulting code in the manuscript, but you can refer to the source code available on the book's website.

The following listing demonstrates the use of the server dispatcher in this new architecture. You build template HTML representations of mail messages in the `MailSiteApplication` by first retrieving the mail bean from the `MailApiApplication` and then by resolving the FreeMarker template exactly as done in section 4.4.

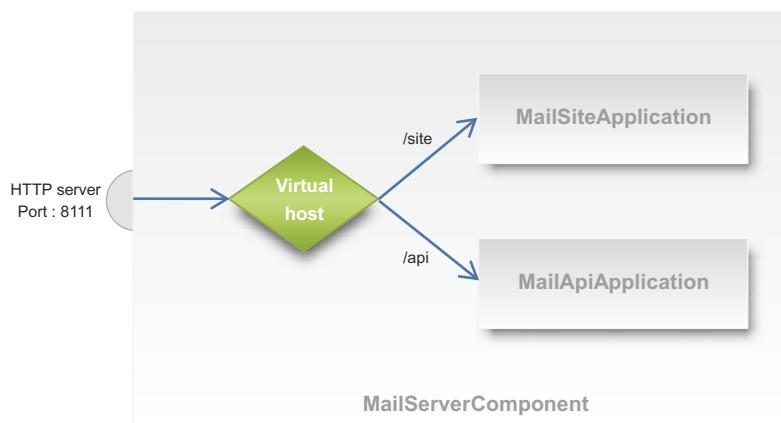


Figure 7.7 Mail server component split into a web API and a website application

Listing 7.20 Optimizing internal calls with the server dispatcher

```

public class MailServerResource extends ServerResource {
    @Get
    Representation retrieve() {
        String accountId = getAttributes().get("accountId");
        String mailId = getAttributes().get("mailId");
        String mailApiUri = getReference().getHostIdentifier()           ←
            + "/api/accounts/" + accountId + "/-mails/" + mailId;
        ClientResource cr = new ClientResource(mailApiUri);           ←
        cr.setNext(getContext().getServerDispatcher());
        MailRepresentation mail = cr.get(MailRepresentation.class);
    }

    Dynamic HTML representation → Representation mailFtl = new ClientResource (
        LocalReference.createClapReference(getClass().getPackage()
            + "/Mail.ftl").get();

        return new TemplateRepresentation(mailFtl, mail,
        MediaType.TEXT_HTML);
    }
}

```

Create API mail URI

Optimized internal call

If you test the new application with the following mail URI, you should first see the login page where you can enter the usual chunkylover53 login and pwd password and then see the HTML page with the mail-editing form filled with the mail data obtained from the mail API:

`http://localhost:8111/site/accounts/chunkylover53-mails/123`

What's remarkable is that by setting the context's server dispatcher as the next property of the client resource, you optimize the call to the API by not even going through the network local loopback. This is more optimal because you save the formatting and parsing of the HTTP messages. If you removed the `setNext(...)` call, the application would still work, but you'd need to add an HTTP client connector to the parent `MailComponent`.

NOTE To test the previous example without manually setting the base URI of the mail resource in the mail API application, you need to use a recent version of the Restlet Framework, in either the 2.0 branch or the 2.1 branch.

Technically speaking, the server dispatcher is providing a shortcut path that forwards your calls right before the virtual host routing in the parent component. Even though this mechanism works very well, it lacks flexibility; it requires you to specify the full absolute resource URI on the request. Also, because it doesn't come with a specific URI scheme, you can't use it as the target of references (in an XSLT representation, for example). This is where the RIAP feature of the Restlet Framework comes into play.

7.5.2 RIAP pseudoprotocol

Restlet Internal Access Protocol (RIAP) is a mechanism that allows various Restlet applications to call upon each other to retrieve resources for further processing. This

feature can be used to access resources contained in the same Restlet component, virtual host, or application, allowing powerful modularization of larger Restlet applications or even calls internal to a single application.

RIAP arose as a solution to support execution of XSLT stylesheets capable of importing and transforming local documents without suffering network costs. It comes with a `riap://` URI scheme that identifies a pseudoprotocol. This terminology is derived from Apache Cocoon; it describes the difference between a “real” or “official” public protocol and the RIAP scheme, which only relates to internal processing of the Restlet system architecture.

RIAP brings a URI notation for interapplication calls that exposes them through the uniform interface. There are three flavors of this protocol, with distinct URI authorities for different use cases:

- `riap://application/` :—Resolves the rest of the URI relative to the current context’s application (applications can use this scheme to call resources within themselves)
- `riap://host/` :—Resolves the rest of the URI relative to the current context’s virtual host
- `riap://component/` :—Resolves the rest of the URI with respect to the internal router of the current context’s component

The following listing updates the previous example to use RIAP instead of the server dispatcher. As you can see, the resulting source code is slightly simpler and offers more flexibility. In terms of performance this is equivalent to the server dispatcher.

Listing 7.21 Optimizing internal calls with the RIAP pseudoprotocol

```
String accountId = getAttribute("accountId");
String mailId = getAttribute("mailId");
String mailApiUri = "riap://host/api/accounts/" + accountId + "/mails/" + mailId;
ClientResource cr = new ClientResource(mailApiUri);
MailRepresentation mail = cr.get(MailRepresentation.class);
```

Create API mail URI

Optimized internal call

Note that internal resources that require protocol-specific attributes of the URI where it’s invoked (like the hostname) might yield errors or unexpected results when called via the RIAP protocol. The main advantage of RIAP is flexible decomposition of your component into smaller, reusable, configurable, and interchangeable applications while assuring optimal efficiency when calling between them.

The RIAP schemes application and host are easy to understand and use, but how to use the component authority is probably unclear at this point. The final section of this chapter covers the internal router and how those modularized applications can be made private to the component, unreachable from the outside world. Note that you can create instances of `Client` and `Server` classes with the RIAP pseudoprotocol and send calls between them inside the same component, which should be unique to the current JVM. This method can be useful when the client side has no direct access

to the current Restlet's context or its client dispatcher, such as inside callback methods of a third-party library invoked from Restlet's server resources.

7.5.3 Private applications

Sometimes application modules should not be accessible directly from the outside world but only via other applications contained in the same component. The Restlet API supports such private applications by providing a `Component.internalRouter` property where applications (or any `org.restlet.Restlet` instance) can be attached. If this application instance isn't attached at the same time to a virtual host, then it's guaranteed to not be reachable from one of the server connectors.

With this additional private router (which can be thought of as a private host), you can effectively modularize a larger component and separate public applications facing the web from private ones, only providing support resources to the former ones.

That's great, but you may think one piece is missing in this puzzle: how can public applications invoke the private ones? Because each application is technically isolated from the parent component except via the `Context` class, you have to rely again on the RIAP feature, this time using the `riap://component/` flavor listed in the previous subsection.

You implement this public/private architecture with the sample `MailServerComponent` as illustrated in figure 7.8. As you can see, we decided to isolate the `MailApiApplication` by attaching it to the internal router under the `/api` root URI path from the `MailSiteApplication` publicly available under the default virtual host attached to the `/site` root URI path.

Reviewing the source code, there are a few differences compared to the previous RIAP example to achieve this design. First you need to modify the constructor of `MailServerComponent`, as illustrated in the following snippet, to change the `MailApiApplication` attachment:

```
getDefaultHost().attach("/site", new MailSiteApplication());
getInternalRouter().attach("/api", new MailApiApplication());
```

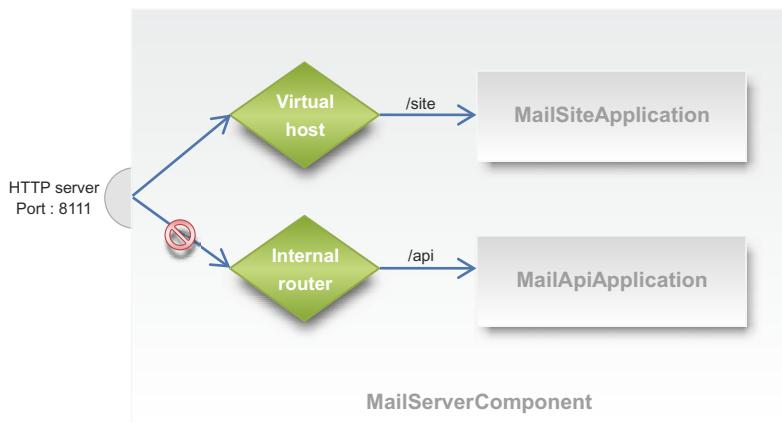


Figure 7.8 Isolating public and private applications

The second point is to update the MailServerResource to modify the way the mail URI is built in order to use the RIAP component scheme:

```
String mailApiUri = "riap://component/api/accounts/" + accountId +
    "/mails/" + mailId;
```

You can now test that the component is working fine by opening `http://localhost:8111/site/accounts/chunkylover53-mails/123` in your browser and verifying that `http://localhost:8111/api/accounts/homer-mails/123` can't be reached anymore!

Before ending this chapter we'll discuss how to persist the state of your Restlet applications in general and more precisely of their resources state.

7.6 Persisting resources state

When you need to provide persistence for your Restlet application, you can choose among dozens of serious technologies. Several types of storage mechanisms are frequently used in addition to relational databases, such as cloud databases like Cassandra, object databases like db4o, graph databases like Neo4j, distributed data grids like Hazelcast, or full-text indexes like Lucene.

For relational databases the most common standard is Java Database Connectivity (JDBC), which can be used directly via its Java API or indirectly via object-relational mapping (ORM) tools such as Hibernate, iBatis, or DataNucleus. Higher-level persistence Java APIs such as JPA and JDO APIs are also reasonable choices.

7.6.1 The JDBC extension

As with presentation technologies, the Restlet Framework tries to stay agnostic regarding your persistence choices because this decision often depends on your project's technical environment and goals.

Restlet provides the `org.restlet.ext.jdbc` extension as an alternative, more RESTful way to use JDBC. This isn't a recommendation over other approaches to managing persistence—it's only one option. The idea is to use a pseudoprotocol based on the JDBC URI scheme. To access a MySQL database the URI template is as follows:

```
jdbc:mysql://{hostname}/{database}
```

This extension offers a Restlet client connector supporting only the POST method in order to issue JDBC requests, using an XML representation as in the following listing.

Listing 7.22 XML request for the JDBC client connector

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<request>
  <header>
    <connection>
      <usePooling>true</usePooling>
      <property name="user">chunkylover</property >
      <property name="password">pwd</property >
      <property name="...">1234</property >
      <property name="...">true</property >
```

Configuring the
JDBC connection

```

</connection>
<start>10</start>
<limit>20</limit>
<returnGeneratedKeys>true</returnGeneratedKeys>
</header>
<body>
  <statement>UPDATE myTable SET myField1="value1" </statement>
  <statement>SELECT msField1, myField2 FROM myTable</statement>
</body>
</request>

```

List of JDBC statements

When you create such a request, you can either use an `AppendableRepresentation` by hand or use FreeMarker or Velocity template representations. This XML request is then parsed and converted to an equivalent JDBC call in Java. The result set and additional properties are also returned as an instance of the `RowSetRepresentation` class, which can be automatically converted to XML, reusing the format defined by the `javax.sql.rowset.WebRowSet` interface. When it's used directly, you can also access the optional list of generated keys and update count.

Let's now continue exploring Restlet support for persistence technologies with the Lucene extension for searching.

7.6.2 *The Lucene extension*

A frequent feature that developers want to add to web applications is search. In Java land and even beyond, the Apache Lucene open source project has become the de facto standard for efficient indexing and retrieval, in particular for textual content.

To help you use Lucene, Restlet comes with an `org.restlet.ext.lucene` extension that facilitates integration with two subprojects:

- *Apache Tika*—A content analysis toolkit
- *Apache Solr*—An enterprise search platform

For Tika the extension provides a `TikaRepresentation` class that facilitates the extraction of textual content from various media types such as PDF, Word, HTML, and XML. It can also extract useful metadata about those textual representations, such as the page count, the author, or the modification data.

For Solr the extension provides a pseudoconnector for this specific URI scheme:

```

solr:///{{command}}
solr://{{coreName}}/{{command}}

```

This client connector lets you interact with an embedded Solr engine from your Restlet application using the Restlet API. You have access to the same web API as the one accessible through HTTP from a standalone Solr installation [10]. You'll now look at some best practices when dealing with resource state persistence before moving on to the third part of the book.

7.6.3 Best design practices

When using any persistence solution with Restlet, you should try loading and storing the state of your resources each time a `ServerResource` instance is created in order to process a new call. The goal is to respect the stateless constraint of REST.

This constraint doesn't mean that your application can't have state, but that the conversation between clients and servers shouldn't be driven by out-of-band information such as cookie-based sessions promoted in the Servlet API.

For efficiency reasons you should consider putting in place database connection pools and data caches in your Restlet application subclasses, accessing those mechanisms by casting the result of the `ServerResource#getApplication()` method. Another option to share state between several resource instances is via the parent `Context` instance, which contains a map of attributes.

When dealing with transient data, such as a shopping cart or transactions running across several web pages, you should rethink your application design to turn this data into resources, such as a shopping cart resource with a specific URI. The state of those resources might be stored in a database like other resources, maybe with a shorter life-cycle but without special treatment.

Far from being a backward step, this stateless constraint can even benefit your end users—they won't experience the typical session expiration issue when taking too long to complete a purchase order, for example.

7.7 Summary

In this chapter you learned best practices that, taken together, make the difference between a prototype Restlet application and a more mature one suitable for production.

You first saw how to complete an application with the usual web elements manipulated in most web APIs and websites, such as HTML forms handling, static file serving, and cookie manipulation. Each feature has been carefully illustrated in the context of the ongoing mail server example.

You also learned two complementary ways to handle web feeds in the Atom and RSS formats using two Restlet extensions, comparing the advantages and disadvantages of each of them. Then you learned how to redirect client calls, either manually by setting the proper HTTP statuses or more automatically and powerfully with the `Redirector` class.

We also discussed performance and modularization aspects in order to deal with applications growing in size and complexity. We looked at HTTP built-in features such as a conditional method processing, entity compression, and caching support. We also introduced Restlet-specific features such as the server dispatcher, the internal router, and the RIAP pseudoprotocol to communicate optimally inside a Restlet component.

You'll now continue exploring further Restlet Framework possibilities, such as deployment in cloud computing, web browsers, and smartphones—and innovative semantic web support.

Part 3

Further use possibilities

P

art 2 gave you the necessary knowledge to turn a Restlet prototype application into a production-ready Restlet application. We went relatively deep inside the Restlet Framework and covered many advanced features. Now, in part 3, we step back and look at the bigger Restlet picture: Restlet as part of the modern multifaceted Web, including mobile web, semantic web, and cloud computing.

Chapter 8 covers how to use Restlet in cloud computing environments such as Google App Engine, Amazon Web Services, and Windows Azure, including deployment of both server-side and client-side web applications and consumption of cloud-based web APIs such as Amazon S3 and APIs based on OData.

Chapter 9 shows that the Restlet Framework is flexible enough to be used in mobile Android devices as well as in web browsers, not only as a Java Applet but also without any plug-ins through the Restlet edition for Google Web Toolkit. We cover all remaining Restlet editions and position the framework as a unique RESTful middleware for mobile cloud computing.

In chapter 10 we explain how web APIs built with Restlet can not only support usual representation media types such as XML, JSON, or HTML but also RDF, the core format of the Semantic Web, ensuring that your Restlet application can be part of the quickly growing Linked Data.

Chapter 11 concludes the book by covering what the Restlet ecosystem has to offer, revealing the roadmap for future Restlet development, and suggesting ways in which you can contribute to this open source project.

At the end of part 3 you should be equipped to develop complex web APIs deployed in popular cloud platforms and accessible from many types of clients, including rich web clients, smartphones, tablets, and even Semantic Web robots!

Using Restlet with cloud platforms



This chapter covers

- Using Restlet in the cloud
- Deploying Restlet applications to various cloud platforms
- Accessing RESTful applications from the cloud using Restlet
- Securely accessing intranet resources from public cloud platforms with Restlet

From the beginning of the book, you've seen how Restlet is an easy, convenient, and powerful framework for implementing and accessing RESTful applications. But Restlet goes further—it's also a complete middleware for connecting heterogeneous remote applications based on REST and HTTP.

In this chapter we describe the benefits of using Restlet for cloud computing in both providing and consuming RESTful web APIs. We first look at how to deploy Restlet applications into various cloud platforms, such as Google App Engine, Amazon Beanstalk, and Windows Azure.

Then we look at how you can use Restlet to access web APIs deployed in the cloud. You can see Restlet as a universal client to consume RESTful applications

(and any kind of HTTP backend). We discuss Restlet's built-in features that ease access to web services such as OData and Amazon S3. We cover OData in depth because Restlet provides comprehensive support for this technology with an abstraction layer, which makes using OData services much easier. We finish the chapter by covering Restlet's support for the Secure Data Connector protocol, which lets cloud applications access intranet resources in a highly secure way. Primarily developed by Google for the Google App Engine platform, SDC is now available to all kinds of platforms, thanks to Restlet.

8.1 Restlet main benefits in the cloud

Before going into the details of Restlet's capabilities and benefits in the context of cloud computing, figure 8.1 introduces the main three layers of cloud computing.

As with regular architectures, each layer builds on top of the layers below it. Table 8.1 describes each cloud layer in more details.

Let's see how Restlet fits in a cloud computing environment, first by improving portability and reducing vendor lock-ins.

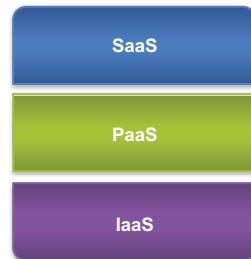


Figure 8.1 Three layers of cloud computing

Table 8.1 Description of cloud computing layers

Layer	Description
Infrastructure as a Service (IaaS)	Delivers computer and storage infrastructure, typically using a virtualized data center, as a service. Rather than purchasing servers, software, data center space, and network equipment, clients instead buy those resources as a fully outsourced service. Typically, the provisioning of those resources doesn't require any manual operation, thanks to the use of web API.
Platform as a Service (PaaS)	Provides a computing platform that facilitates the development, deployment, and management of applications without the cost and complexity of buying and managing the underlying software stack. This layer typically builds on top the IaaS layer.
Software as a Service (SaaS)	Provides ready-to-use applications accessible by users from a web browser. Those applications don't require any installation or maintenance on the user's computers, resulting in great cost savings and higher productivity. The main drawback is the need for fast and reliable internet connectivity.

8.1.1 Better SaaS portability

As you've seen in previous chapters, Restlet pluggable extension design lets you avoid dragging in unnecessary dependencies, ensuring an optimal footprint for your application. In addition to extensions, Restlet Framework provides the concept of edition targeting specific execution environments by handling its specificities and restrictions. In the context of cloud computing, four editions are provided, as listed in table 8.2.

Table 8.2 Restlet editions related to cloud computing

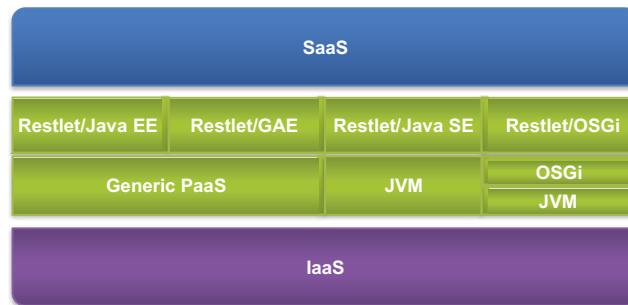
Edition	Description
Edition for Java EE (JEE)	Major PaaS offerings such as AWS Beanstalk, AppFog, Heroku, and CloudBees support the deployment of standard Java EE applications. Restlet edition for Java EE comes with few restrictions besides the need to go through the Servlet API, preventing the use of standalone HTTP server connectors.
Edition for Google App Engine (GAE)	Targets the GAE platform as a service by only supporting JRE classes that are part of GAE's white list. The edition also provides special extensions usable in this context to take advantage of GAE security features without gluing your application too much with GAE's proprietary Java APIs.
Edition for Java SE (JSE)	Targets the lower-level IaaS cloud offerings when you need to have control of the JVM, the network configuration, and the full deployment context such as with Amazon EC2 and Rackspace servers. More work is required from you, but you can use more of the Restlet Framework, including standalone connectors.
Edition for OSGi Environments (OSGi)	Targets the lower-level IaaS cloud offerings like the JSE edition, but provides additional level of dynamicity and versioning management features thanks to its multitenant JVM features. This makes it well-suited for highly available cloud environments.

The combination of editions and extensions helps you consistently execute the same code in different contexts. Although the original Java slogan is “write once, execute anywhere,” that isn’t true for all Java-based environments because of their restrictions. Restlet goes further by trying to support this slogan with cloud platforms as well. This benefit also applies to browsers and mobile devices (covered in the next chapter).

Figure 8.2 illustrates how a SaaS application can be built with the Restlet Framework, based on any of the four mentioned editions, without compromising much portability.

You can also start developing your application locally, without an IaaS, deploy it within a regular Java EE container or simple JVM or OSGi environment, and later decide to migrate it to the cloud without much additional work.

Beyond increased portability of your applications to the cloud and within the cloud, let’s now see how Restlet can also help to consume services from the cloud.

**Figure 8.2 Restlet Framework positioning in cloud layers**

8.1.2 Easy client access to services from the cloud

As you've seen before, Restlet is equally capable of creating web applications accessible via RESTful web APIs and consuming third-party web services. This capability is very important in a cloud context in order to consume services provided by:

- The IaaS layer such as object storage services, queuing services, and managed databases
- Third-party services such as AWS S3, Twitter, Twilio, Bit.ly, Netflix, or LinkedIn via their web APIs
- Secure intranet applications that are hosted behind your organization firewall without changing its configuration

Initially you may want to use the client SDKs provided by those services, adding them to your application stack. But you'll quickly end up with redundant or incompatible dependencies and a heavy footprint.

That's where Restlet comes in handy—by taking advantage of the natural interoperability offered by REST and HTTP. It also provides extensions that support specific security schemes to provide access to Amazon S3 resources and OData services, and even support for Google Secure Data Connector (SDC) in order to be connected to intranet resources, as illustrated in figure 8.3.

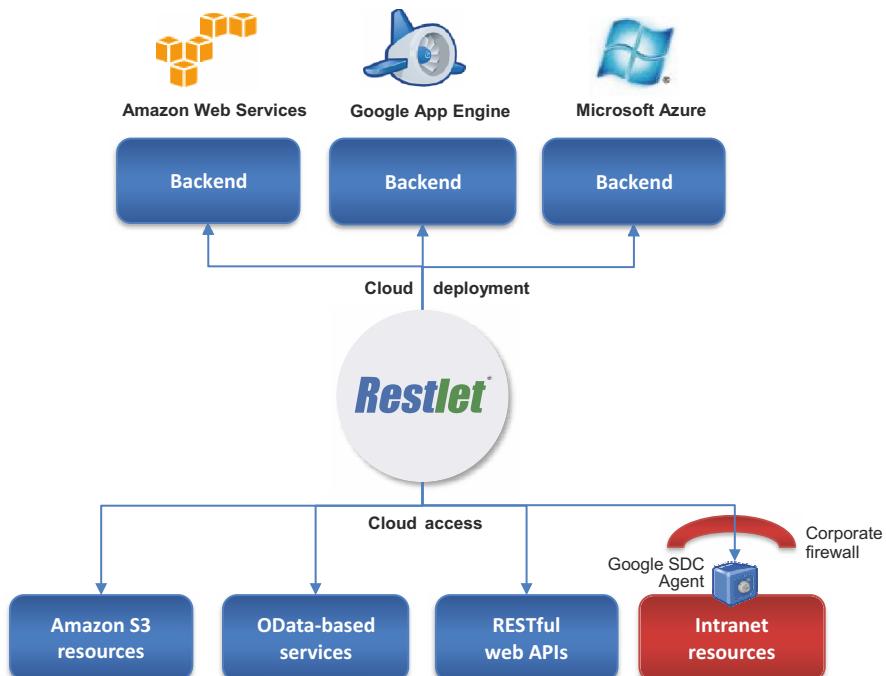


Figure 8.3 Restlet as a cloud middleware to deploy, execute, and connect RESTful applications

Now that you've seen Restlet's overall place in the cloud, let's get more concrete and describe how to deploy Restlet applications to various cloud platforms, starting with Google App Engine (GAE).

8.2 Deployment in Google App Engine

Google App Engine is a generic Platform as a Service that can host any web application and offer it massive scalability in terms of storage and network and computing power. Free accounts are available for limited usage, and you can pay for extra resources (CPU time, bandwidth, storage, and mails sent).

GAE has many benefits but comes with important constraints because you share computing resources with others. GAE makes your web applications live in a sandbox with access to a limited set of base Java APIs. Before describing deployment aspects, let's look more closely at the GAE platform, how it works, and its constraints.

8.2.1 What is GAE?

GAE is the cloud platform offered by Google that enables the development and hosting of web applications on top of Google's infrastructure, using the same technologies and data centers that power Google's own applications. Administration is simplified as well, and you don't need to worry about hardware and scalability. Everything is handled by the platform.

GAE currently supports Go (Google's own C-derived language), Python, and Java to develop applications, but we naturally focus here on the Java support. Figure 8.4 describes the overall architecture of GAE.

Compared to other cloud platforms, such as Amazon Beanstalk (described later in the chapter), GAE makes it easier to write scalable applications but comes with restrictions because it can only run a limited range of applications designed specifically for that platform. Here are some of these restrictions:

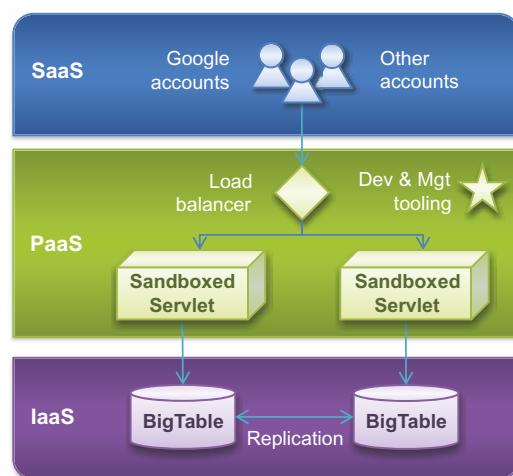


Figure 8.4 Global architecture of the GAE platform

- Read-only access to the filesystem is allowed.
- Only code triggered by HTTP requests, XMPP requests, or the Task service can be executed.
- Only a subset of the classes from the Java Standard Edition is available (the JRE Class White List).
- Sockets aren't under the application's control.
- Applications can't create new threads.
- Incoming HTTP requests have a limited time to complete, typically 60 seconds.
- HTTP chunked encoding isn't supported, so the size of entities sent to the application must be known in advance.

Workaround for chunked encoding provided by Restlet

At this time, GAE ignores request entities that are chunk-encoded. To work around this limitation, you can call the `ClientResource.setRequestEntityBuffering(true)` method. This configuration prevents the use of chunk encoding for request entities by prebuffering them in order to compute their size.

Due to those restrictions, a dedicated edition of the Restlet Framework was developed and is available for download. This edition guarantees that only Java classes from the GAE/J whitelist are used for processing within the Restlet Framework. At the time of writing, a wide range of extensions from the more generic Java EE edition is usable except JDBC, XStream, and a couple of other ones. The complete list is provided in the Javadocs of the GAE edition.

Let's now get started and deploy a Restlet application on the GAE platform.

8.2.2 *Deploying Restlet applications in GAE*

The first step in deploying a Restlet application in GAE is to register an application in the remote console located at <https://appengine.google.com>. In the registration screen, you have to specify an application identifier and a description. The identifier will be part of the root URI of the application. If you choose `reia` for the identifier, the root address for the application will be `http://reia.appspot.com`. You can also configure other configuration hints regarding security, storage, and replication here.

Once the application is configured, it appears in the application list, as shown in figure 8.5.

My Applications

My Applications			
Application	Title	Storage Scheme	Current Version
reia	REIA Application		None Deployed
Create Application You have 9 applications remaining.			

Figure 8.5 Application list containing the Restlet application

To deploy the web application, Google App Engine provides a set of various tools:

- *Command-line tools*—Scripts to upload web application content in the GAE platform and to start the local development server. Versions written in Java and Python are available.
- *Eclipse support*—A plug-in that can upload your application from the Eclipse IDE by clicking a button. This approach requires having a project of type *Web Application*.

In addition to regular Servlet deployment artifacts, described in chapter 3, your Restlet application must include a file named appengine-web.xml in order to be deployed on the Google App Engine platform. This file is located under the WEB-INF directory of your Web application. At a minimum, it contains the application identifier and the version of the content, as described in the following snippet:

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application>reia</application>
    <version>1</version>
</appengine-web-app>
```

Additional configuration properties are available and described at length in GAE documentation, although you will likely want to add the `<threadsafe>true</threadsafe>` line that ensures that concurrent requests can be made to your application, something that Restlet naturally supports, reducing your GAE bill at the end of the month.

The last steps consist of packaging the application within a WAR file and pushing it to GAE. We first use the Java-based command line tool to show how to upload the application to Google App Engine.

USING THE COMMAND LINE TOOL

After downloading the SDK from <http://code.google.com/appengine/downloads.html> and unzipping it, you have access to the appcfg.sh script under the bin folder. This script allows you to upload the application.

The first parameter corresponds to the action to execute, update in this case. The second parameter specifies the folder where the application content is located. The following snippet shows the command to execute in order to upload your Restlet application:

```
<APPENGINE_JAVA_SDK_HOME>/bin/appcfg.sh update <APPLICATION_ROOT>/war
```

Figure 8.6 indicates that a version is now deployed and available.

My Applications

My Applications			
Application	Title	Storage Scheme	Current Version
reia	REIA Application		1 ↗
Create Application			
You have 9 applications remaining.			

Figure 8.6 Application list containing the deployed Restlet application

In this case, the Restlet application is now available at the address <http://reia.appspot.com>. To get the contact with identifier 1, you can use the address <http://reia.appspot.com/contacts/1>.

Let's continue by briefly looking at the deployment option from an Eclipse IDE.

ECLIPSE SUPPORT FOR GAE

Google provides an Eclipse plug-in to facilitate the development of GAE applications. It provides wizards to create a GAE application as an Eclipse project, test it locally, and deploy it to GAE. This plug-in, called Google Plugin for Eclipse, also supports Google Web Toolkit, Google Cloud SQL, and Google APIs development.

It's available at <https://developers.google.com/eclipse/>. We describe its GWT features in more detail in the next chapter, in section 9.1.2.

Before ending the GAE section, we'll describe a convenient feature of Restlet that was added to the GAE edition in order to provide an easier integration of Google Accounts authentication with Restlet's regular security API.

8.2.3 Using Google Accounts authentication

GAE provides a built-in authentication mechanism based on Google Accounts. To help you take advantage of this feature, Restlet provides an extension to integrate it with its security API for authentication and authorization. (The security API is described in sections 5.2 and 5.3 in chapter 5.)

Security in Restlet is based on the `Authenticator` class to authenticate the current user of a request and the `Enroler` interface to determine its granted roles. For using the authentication based on Google Accounts, the `org.restlet.ext.gae` extension provides dedicated classes:

- `GaeAuthenticator` class—Provides integration to the GAE `UserService`.
- `GaeEnroler` class—Adds a Restlet `Role` object to the request's `clientInfo` property only if GAE reports that the user is an administrator.

You can configure these two classes within the `createInboundRoot` method of your Restlet application, as described in the following listing.

Listing 8.1 Configuring Google Accounts-based authentication in Restlet applications

```
public Restlet createInboundRoot() {
    ...
    Authenticator authenticator = new GaeAuthenticator(getContext());
    Enroler enroler = new GaeEnroler("admin", "Administrator");
    authenticator.setEnroler(enroler);
    authenticator.setNext(myRouter);
    return authenticator;
}
```

The diagram illustrates the code flow. A callout labeled 'Instantiate GAE authenticator ①' points to the line 'Authenticator authenticator = new GaeAuthenticator(getContext());'. Another callout labeled 'Instantiate, configure GAE enroler ②' points to the line 'Enroler enroler = new GaeEnroler("admin", "Administrator");'. Arrows from the labels point to their respective code lines.

The `GaeAuthenticator` needs to be instantiated ① and returned by the `createInboundRoot` method to secure the applications. A dedicated enroler needs then to

be instantiated ② with `admin` as the Restlet role name and then set on the previous authenticator.

We'll now tackle the next cloud platform in our list: Amazon Elastic Beanstalk.

8.3 Deployment in Amazon Elastic Beanstalk

At the beginning of 2011, Amazon announced a new offering: AWS Elastic Beanstalk. It provides a very convenient way to deploy and manage applications in the AWS cloud. It truly simplifies deployment and execution of Servlet-based applications, compared to the existing lower-level Amazon EC2 and S3 products.

In this section we describe the features and characteristics of the platform and guide you step-by-step to deploy Restlet-based applications.

8.3.1 What is Elastic Beanstalk?

Amazon Elastic Beanstalk is a complete and generic platform (PaaS) to deploy and execute any Servlet-based applications. When deploying such an application, the service under the covers automatically deploys it on one or more EC2 instances, depending on the current load monitored, each running an Apache Tomcat server. Restlet easily supports such applications using its Java EE edition, including the `org.restlet.ext.servlet` extension.

Figure 8.7 provides an overview of the platform provided by Beanstalk and associated services for deployment, execution, and management. It's comparable to GAE, with fewer technical constraints but with potentially higher costs in production. For comparison, GAE provides a free plan allowing developers to deploy ten applications with no limit in time.

The deployed applications are automatically added to a load balancer to distribute requests more evenly and efficiently across instances in order to help maintain

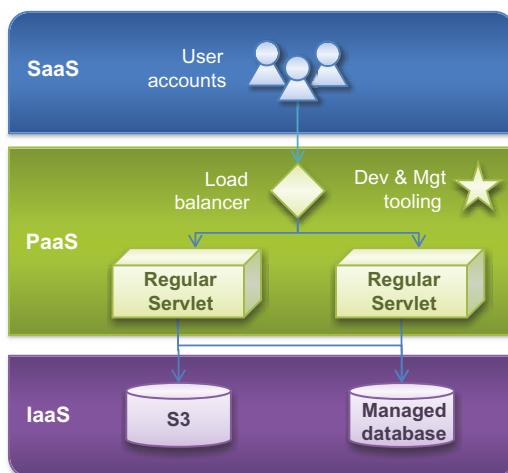


Figure 8.7 Overview of Amazon Beanstalk parts

scalability. In addition, AWS Elastic Beanstalk provides a set of tools to manage and monitor your Restlet applications:

- *AWS Management Console*—Used to upload WAR files and configure the runtime environment. Other tools, like the AWS Toolkit for Eclipse, the web service APIs, or the Command Line Tools, are also available.
- *CloudWatch monitoring*—Provides monitoring metrics such as average CPU utilization, request count, and average latency.
- *Amazon Simple Notification Service*—Allows sending notifications through email when application health changes or application servers are added or removed.

Now that you have an overview of what Amazon Elastic Beanstalk is, let's concentrate on how to deploy Restlet applications on this platform.

8.3.2 Deploying Restlet applications

The AWS Management Console, at <https://console.aws.amazon.com/elasticbeanstalk/>, is the place to manage applications and deploy them. It provides a consolidated view of each service provided by Amazon AWS. The tab dedicated to Elastic Beanstalk shows all your configured applications (figure 8.8).

Now to describe step-by-step how to add a Restlet application. The first step of the creation wizard consists of specifying the application name and the corresponding WAR file to upload. You then specify some environment details, like the name under which the application will be published (which determines its URI) and the container type to use. You can then provide configuration details like the EC2 instance type and the application health check address.

Figure 8.9 shows the screen summarizing all hints filled during the application creation wizard.

Beanstalk then configures all the necessary resources for the environment within the Amazon AWS platform—in particular, the EC2 instance(s). Because the “Launch a new environment running the application” check box is checked, the environment is automatically launched. The square to the left of the application name indicates

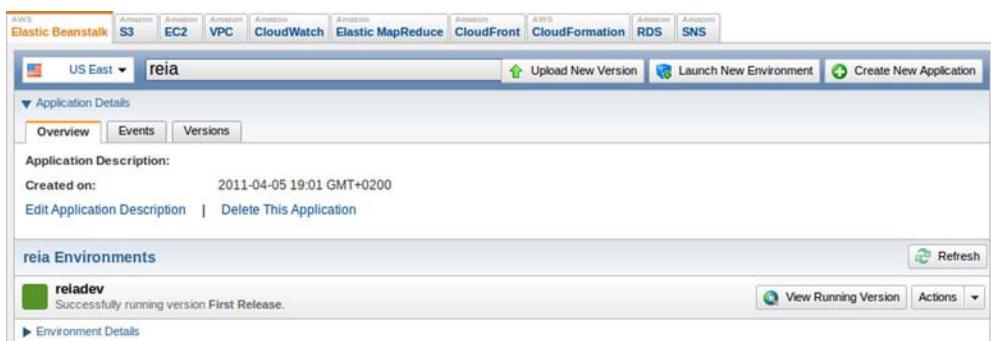


Figure 8.8 Overview of the Elastic Beanstalk tab of the AWS Console

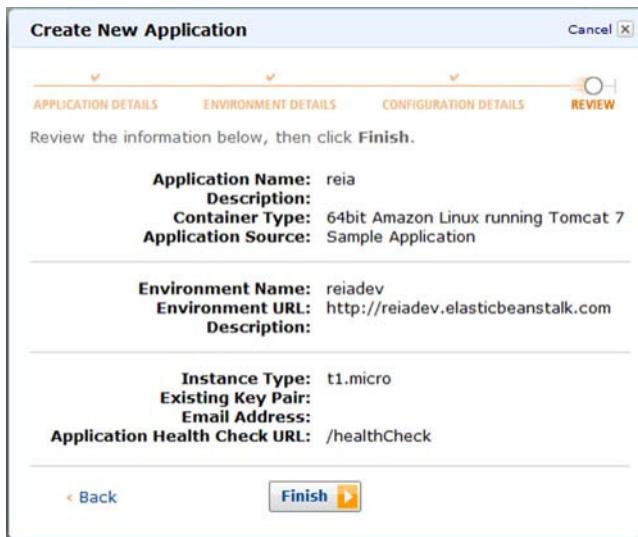


Figure 8.9 Overview of the application creation details

Eclipse support for AWS

Amazon also provides an Eclipse plug-in facilitating the development, debugging, and deployment of AWS-based applications. It provides an SDK for Java and tools to manage SimpleDB and EC2 services. This plug-in is available at <http://aws.amazon.com/eclipse/>.

whether the application is being started (spinning circle), as shown in figure 8.10. It then becomes either green or red depending on the deployment result.

In this case, the Restlet application is available through the address <http://reiadev.elasticbeanstalk.com>. To get the contact with identifier 1, you can use the address <http://reiadev.elasticbeanstalk.com/contacts/1>.



Figure 8.10 Overview of the application while being deployed to AWS Elastic Beanstalk

As you can see in the wizard, you need to specify a *health check* URI to check whether the application is up. You can't neglect this aspect or you'll have errors during launching, and the application won't be available. This special URI needs to be provided by the Restlet application. For this purpose we attach a dedicated URI within the application to an inline server resource that returns an empty representation when handling any HTTP method, shown in the following snippet:

```
public Restlet createInboundRoot() {  
    Router router = new Router(getContext());  
    (...)  
    router.attach("/healthCheck", new Restlet(getContext()) {  
        public void handle(Request request, Response response) {  
        }  
    });  
    (...)  
}
```

The last cloud platform in our list is Windows Azure, by Microsoft. Although this platform wasn't initially designed for the Java language, it regularly improves its support for Java, along with other languages.

8.4 Deployment in Windows Azure

Windows Azure provides a comprehensive and scalable cloud computing environment with computing, storage, hosting, and management capabilities. It can integrate on-premises applications with secure connectivity, messaging, and identity management.

In this section we first focus on features and characteristics of the platform and describe step-by-step how to deploy Restlet-based applications to it.

8.4.1 What is Azure?

The Windows Azure platform is a comprehensive public cloud that corresponds to a group of cloud technologies, each providing a specific set of services to application developers. The following list describes the main components of the platform:

- *Windows Azure*—Provides an IaaS solution with computing (web and worker roles), storage (blob storage, table storage), hosting (queue service, content delivery network), and management capabilities. It can integrate on-premises applications with secure connectivity, messaging, and identity management.
- *Microsoft SQL Azure*—Corresponds to a cloud-hosted relational database that's a managed version of Microsoft SQL Server.
- *Azure AppFabric*—Provides a PaaS solution with access control, caching, service bus, and integration services.
- *Marketplace*—Provides both an application marketplace comparable to Google Apps marketplace and a data marketplace based on the OData protocol.

Windows Azure has been built from the ground up with interoperability in mind. This translates differently at different levels. The platform lets the user choose between

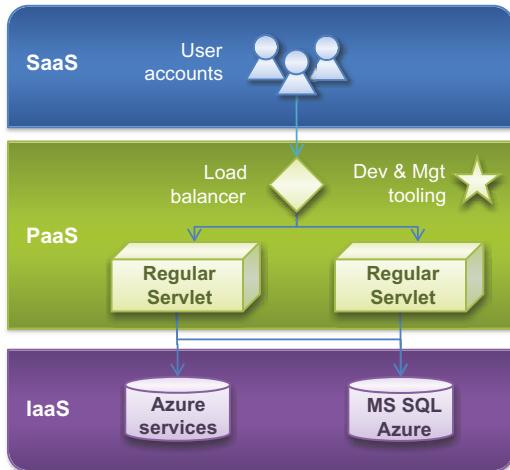


Figure 8.11 Overview of the Windows Azure platform

.NET, PHP, Ruby, Python, and Java for the application programming language. Moreover, the platform supports major standards including HTTP, XML, SOAP, and REST to offer interoperability. Figure 8.11 summarizes Azure from a Java deployment point of view.

Let's now concentrate on the deployment of a Restlet application on this platform.

8.4.2 Deploying Restlet applications

Before deploying applications in Azure, you need to install tools to build and package the applications for this platform:

- *Windows Azure SDK*—Provides developers with the APIs, basic tools, documentation, and samples needed to develop web applications that can run on Windows Azure.
- *Windows Starter Kit For Java*—Corresponds to a project template to build and package a Java-based application as an Azure-compliant one. This tool is available at <http://wastarterkit4java.codeplex.com/>. It requires using Windows 7 or Vista.
- *Apache Ant*—The famous Java build tool is available at <http://ant.apache.org>.

To install the Windows Azure SDK, Microsoft recommends using the Web Platform Installer available at www.microsoft.com/web/downloads/platform.aspx. This tool helps to download, install, and upgrade components of the Microsoft Web platform. Specify *Azure* in its search tool and add the Windows Azure SDK entry for install. When launching the process, the Web Platform Installer transparently downloads it and its required dependencies and installs all of them. This will save you a lot of time and pain.

Now that this SDK is installed, you have access to the Windows Azure SDK command prompt that you'll use to package the application using the command line. Install Apache Ant and the Windows Starter Kit for Java by downloading the distribution files

from www.windowsazure.com/en-us/develop/java/java-home/ and unzipping them. After that, everything is in place to begin work on your web application.

You need to add specific tools within your Azure applications to execute them using Java technologies. For Restlet applications, the overall package must contain the following elements:

- Java Runtime Environment (JRE) to execute Java applications within the Azure platform
- Servlet engine (Tomcat) to execute Java Web applications
- Content of the Restlet application

Let's start with the first one, the JRE. Because distributions aren't available from Oracle as zip files, you need to manually zip your JRE installation directory. You then create a JRE directory under the approot one and put your zipped file there. Do the same for Tomcat by copying the zipped distribution of the server under a tomcat directory.

Figure 8.12 shows the complete structure of the Azure project after adding the JRE and the Tomcat server as zipped files. Notice the WEB-INF and META-INF folders, containing elements of the Restlet web application.

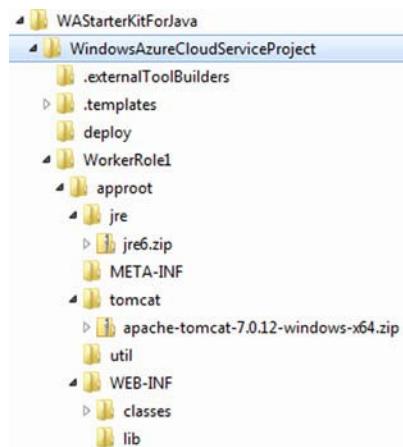


Figure 8.12 Structure of the project after adding JRE and Tomcat

Eclipse support for Azure

An Eclipse plug-in is provided to help in developing Java-based Azure applications. It provides some visual tools to create the project structure, based on a dedicated Eclipse project, and to edit role and endpoint configurations. You can obtain more information at <http://mng.bz/5nrB>.

Before going ahead in the application configuration, let's look at what's present in the template project of the Windows Azure Starter Kit for Java. Table 8.3 lists its main elements.

Table 8.3 Elements present in the template project of the Windows Azure Starter Kit for Java

Element	Description
.cspack.jar	Contains the Java implementation of the windowsazurepackage ant task.
ServiceConfiguration.cscfg	Specifies the number of instances to deploy for each role and provides values for any configuration settings declared in the service definition file.

Table 8.3 Elements present in the template project of the Windows Azure Starter Kit for Java (continued)

Element	Description
ServiceDefinition.csdef	Describes the service model. It defines the roles included with the service and their endpoints and declares configuration settings for each role.
unzip.vbs	Corresponds to the script that makes it easy to unzip archives.
download.vbs	Corresponds to the utility script for downloading files from URL locations into the approot as part of the startup.

In addition to this content, you need to add the actual content of your Restlet application. Copy the contents of the WAR file to the approot folder. In this case, it corresponds to the content present in the WEB-INF directory:

- *web.xml*—Descriptor file for the Java-based web application
- *Lib*—Directory containing all the third-party libraries needed, such as Restlet JARs
- *Classes*—Directory containing classes for the web application

The last step to configure the Azure application is to set how to access it within the ServiceDefinition.csdef file, shown in the following listing.

Listing 8.2 Content of the ServiceDefinition.csdef file

```
<?xml version="1.0" encoding="utf-8"?>
<ServiceDefinition name="WindowsAzureCloudServiceProject" (....)>
  <WorkerRole name="WorkerRole1">
    <Startup>
      <Task commandLine="util\startup.cmd"
            executionContext="elevated" taskType="simple"/>
    </Startup>
    <Endpoints>
      <InputEndpoint name="Http" protocol="tcp"
                     port="80" localPort="8080" />
    </Endpoints>
  </WorkerRole>
</ServiceDefinition>
```



The InputEndpoint XML element specifies which protocol is supported to access the application and the corresponding port translation ①. In this case, you specify to the Windows Azure load balancer that traffic on port 80 for the application needs to be forwarded to port 8080 on the machine running the Azure instance.

Another important file to update is the one located in the util directory under the approot one. It contains all the commands to execute when the hosted service starts up. In this case, it consists of unzipping JRE and Tomcat archives and copying the web application content within the webapps directory of Tomcat. You need to be very careful when implementing this file because an error in paths will cause failures at startup. The following listing describes its content.

Listing 8.3 Commands to execute at the hosted service startup

```

@REM unzip JRE
cscript "util\unzip.vbs"
    "jre\jre6.zip" "%ROLEROOT%\approot"
@REM unzip Tomcat
cscript "util\unzip.vbs"
    "tomcat\apache-tomcat-7.0.12-windows-x64.zip" "%ROLEROOT%\approot"

@REM copy project files to server
md "%ROLEROOT%\approot\apache-tomcat-7.0.12\webapps\reia"
md "%ROLEROOT%\approot\apache-tomcat-7.0.12\webapps\reia\WEB-INF"
md "%ROLEROOT%\approot\apache-tomcat-7.0.12\webapps\reia\META-INF"
xcopy /S "WEB-INF"
    "%ROLEROOT%\approot\apache-tomcat-7.0.12\webapps\reia\WEB-INF"
xcopy /S "META-INF"
    "%ROLEROOT%\approot\apache-tomcat-7.0.12\webapps\reia\META-INF"

@REM start the server
cd "%ROLEROOT%\approot\apache-tomcat-7.0.12\bin"
set "JRE_HOME=..\..\jre6"
startup.bat

```

The diagram illustrates the sequence of steps in Listing 8.3. Step 1, labeled 'Unzips JRE and Tomcat', points to the first two command blocks. Step 2, labeled 'Copies web application files', points to the third command block. Step 3, labeled 'Starts up Tomcat', points to the final command block.

The first commands unzip archives contained the Java Runtime Environment 6 and Apache Tomcat 7.0.12 ①. You copy all web application files in the webapps directory of the Tomcat distribution to a directory corresponding to the application name ②. In this case, the name is reia. The remaining commands deal with starting up the Tomcat server ③.

By default, the application is configured for use with the local Azure cloud platform. The resulting structure after packaging will be exactly the one on the target Windows Azure cloud. Because you want to directly upload the application to Azure, you need to update the content of the package.xml build file. In the packagetype attribute of the windowsazurepackage XML element, we specify the cloud value instead of the local one, as described in the following snippet:

```

<windowsazurepackage
    sdkdir="${wasdkdir}"
    packagefilename="WindowsAzurePackage.cspkg"
    packagedir = "${wapackagedir}"
    packagetype="cloud"
    projectdir="${basedir}"
    definitionfilename="ServiceDefinition.csdef"
    configurationfilename="ServiceConfiguration.cscfg"
>

```

The diagram highlights the packagetype attribute in the windowsazurepackage XML element, which is set to 'cloud'. A callout points to this attribute with the label 'Sets package type to cloud'.

Executing the Ant script (see the following snippet) will generate artifacts for a deployment using the Azure Console. The createwappackage target corresponds to the default one and is executed when no target is specified.

```
ant -buildfile package.xml
```

This generates two files in the deploy directory that must be uploaded in the console:

- *The csdef file*—Corresponds to the definition file
- *The cscfg file*—Corresponds to the configuration file

Let's connect to the Azure console from the address <https://windows.azure.com> and identify ourselves. Then from the main screen, select the Hosted Services item in the left-hand menu. This gives you access to a dedicated toolbar including commands to create and manage services. To deploy the application, you need to create a new hosted service, using the wizard described in figure 8.13.

In addition to the hints regarding the hosted service, like its name, URL prefix, region, and deployment name, you also need to reference the two files created during the packaging phase:

- *WindowsAzurePackage.cspkg file in the Package location field*—Corresponds to the content of the service in an archive file
- *ServiceConfiguration.cscfg file in the Configuration file field*—Corresponds to the configuration hints for the service

After validating data, service creation starts, and content is uploaded, as shown in figure 8.14. The structure of the service is then created by adding the worker role and instance, as shown in figure 8.15.

Because you checked the “Start after successful deployment” option in the creation wizard, the hosted service is automatically started, as shown in figure 8.16. As

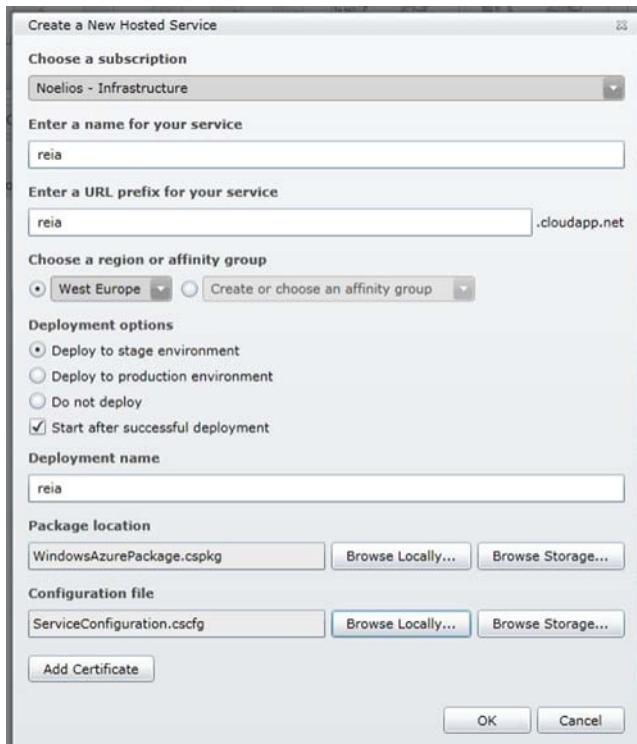


Figure 8.13 Creating the new hosted service on which the Azure application is deployed

Name	Type	Status	Environment
Noelios - Infr...	Subscription	Active	
reia	Hosted Service	Creating...	
reia	Deployment	8% uploaded	Staging

Figure 8.14 The console displays the status of the service deployment.

Name	Type	Status	Environment
Noelios - Infrastructure	Subscription	Active	
reia	Hosted Service	Created	
Certificates			
reia	Deployment	Initializing...	Staging
WorkerRole1	Role	Initializing...	Staging
WorkerRole1_IN_0	Instance	Creating host...	Staging

Figure 8.15 Creation of the worker role and instance after deploying the service

you can see in the DNS name field of the properties area, Azure dynamically associates an entry in the `cloudapp.net` domain to access the service.

In this case, the service could be reached at the following address: `http://e0e47ca9c2a54708b79d0cf96a3e1967.cloudapp.net/reia/contacts/1`.

This coverage of Restlet applications deployment to the Azure platform closes the deployment section. As you've seen, Restlet applications offer a wide range of cloud environments from Google App Engine to Amazon Web Services and Windows Azure. More platforms are also supported, such as Heroku and CloudBees. But the cloud support offered by Restlet doesn't end here, because facilities are also provided on the client side to access web APIs from the cloud and in the cloud. Let's start by describing how to access web APIs from GAE.

Choose Columns ▾				Filter hosted services	Properties
Name	Type	Status	Environment		
Noelios - Infrastructure	Subscription	Active			
reia	Hosted Service	Created			
Certificates					
reia	Deployment	Ready	Staging		
WorkerRole1	Role	Ready	Staging		
WorkerRole1_IN_0	Instance	Ready	Staging		

Figure 8.16 All elements of the hosted service correctly deployed, created, and started

8.5 Accessing web APIs from GAE

As described in section 8.2.2, Google App Engine comes with restrictions on applications due to its scalability and multitenancy requirements. One of them is that GAE applications can't make socket connections directly in order to access remote resources. They need to use services provided by the platform to be able to use higher-level protocols such as HTTP and XMPP.

In this section we describe how to call remote resources using Restlet after covering the specifics in the context of GAE.

8.5.1 GAE restrictions and URL fetch

GAE prevents you from managing low-level sockets but provides a service called URL Fetch to make possible execute requests on remote addresses. It's possible to use it directly, but this service is also provided via the classic `java.net.URLConnection` class. HTTP requests GET, POST, PUT, DELETE, and HEAD can be used and include HTTP request headers and an HTTP request body. Both HTTP and HTTPS protocols are supported. As with the rest of the GAE platform, executing requests on remote addresses comes with limitations:

- Responses can't exceed the maximum response size limit of GAE.
- Requests can only be executed asynchronously by using the low-level API of URL Fetch (but a synchronous mode is available via `HttpURLConnection` class).
- For security and optimization reasons, some HTTP headers can't be modified by the application. These headers are `Accept-Encoding`, `Content-Length`, `Host`, `Vary`, `Via`, and `X-Forwarded-For`.

The URL fetch service also supports the ability to access systems behind a company's firewall using the Secure Data Connector (SDC) technology. This aspect is described in section 8.9. Let's look at how to support the Restlet client support in this context.

8.5.2 Using Restlet to access RESTful applications

Because GAE makes it possible to use its URL Fetch service through the synchronous `HttpURLConnection` class, you'll use the `org.restlet.ext.net` extension when accessing external URLs by putting that extension's JAR file in the classpath of the GAE application. Client requests are then automatically executed through this connector.

When using a Restlet client-side API in this context, you must be very careful to not use features, such as chunked and hanging requests, that aren't supported by the URL Fetch service.

In the next section we cover Restlet support for accessing OData services.

8.6 Accessing OData services

In recent years, Microsoft has enhanced its technology named Astoria, and then called ADO.NET Data Services, by splitting it into an open specification of a data-driven REST API, called OData (for Open Data Protocol), and a proprietary server-side framework called WCF Data Services.

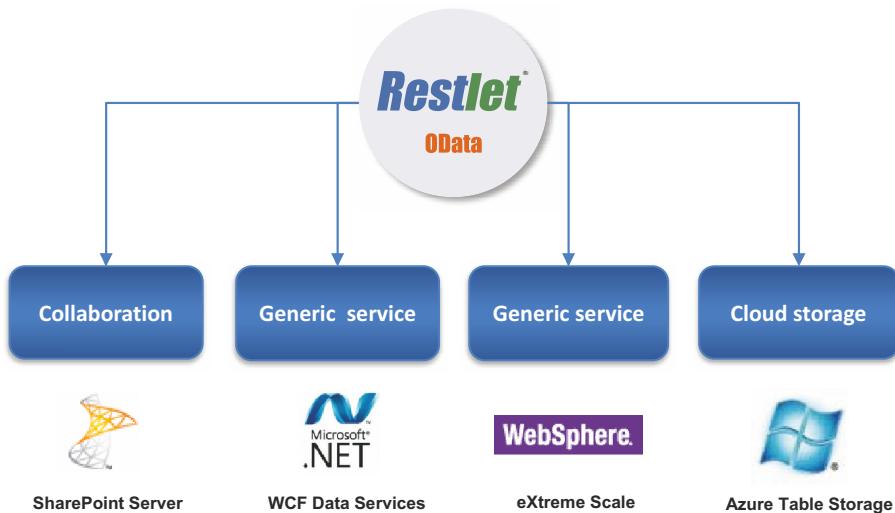


Figure 8.17 Systems accessible using the Restlet OData support

The OData protocol has also been embraced by IBM in its Java-based WebSphere eXtreme Scale product, and Microsoft has used it in several of its products, like Excel PowerPivot, SharePoint Server, Windows Azure Table Storage, and SQL Server Reporting. Other recent initiatives are the project code-named Dallas, which offers a marketplace for data services with full support for access control and billing, and the OData visualizer part of Visual Studio 2010.

OData offers a nice way to connect applications and exchange data on top of REST, HTTP, and Atom/JSON, comparable to the Google Data (GData) protocol. The OData extension for the Restlet Framework provides a convenient way to consume heterogeneous services and access data from a large set of systems typically exposed with Microsoft .NET technology. Figure 8.17 provides an overview of the systems and applications that can be accessed and queried.

As a result, the Restlet extension for OData fosters the interoperability between Java and Microsoft environments. In this section, we first provide an overview of the OData protocol and then illustrate the use of the Restlet extension.

8.6.1 What is OData?

The Open Data Protocol, OData for short, is a protocol for sharing data between applications. This protocol is open and standardized by the OData specification, which is available under the Microsoft Open Specification Promise (OSP) and recently submitted to the OASIS group for broader standardization. It allows querying and updating data over the HTTP protocol.

The OData specification supports features like pagination, ordering, and filtering of data. Several formats such as XML-based Atom and JSON are supported for data. OData is related to Atom and AtomPub—many of its parts are designed as extensions.

Atom and AtomPub

The Atom term applies to several standards. Atom Syndication Format is an XML language used to expose web feeds; Atom Publishing Protocol (AtomPub or APP) is a RESTful web API for creating and updating web feeds and similarly structured web resources.

We also discuss Atom in section 7.2.

OData provides a complete data-centric protocol accessible using HTTP in a RESTful way, thanks to these features:

- Standardization of URI patterns used to expose data resources
- Definition of the HTTP methods available to access and manipulate data
- Definition of an abstract data model (EDM format) but also concrete structure for XML-based Atom and JSON
- Use of URI structuring to query data provided by consumed services

Let's provide examples of URIs to use for querying and updating data, and what representations look like. We concentrate on the XML-based Atom format here.

First, look at an OData query. Consider a service that manages products at host `services.odata.org`. The simplest query gets all products with an HTTP GET of a URI like `/OData/OData.svc/Products` where OData could have any other service name:

```
GET /OData/OData.svc/Products HTTP/1.1
Host: services.odata.org
Accept: application/atom+xml
```

From a client point of view, it's a regular HTTP request and response, except for the additional response header indicating the OData protocol version supported by the server.

The following listing shows the corresponding response and uses the XML-based Atom format because an Accept HTTP header with this content type is specified in the request.

Listing 8.4 Content returned by a simple OData request

```
HTTP/1.1 200 OK
(...)

1 Response
status line

Content-Type: application/atom+xml; charset=utf-8
DataServiceVersion: 1.0

2 OData content
of response

<?xml version="1.0" encoding="utf-8"
      standalone="yes"?>
<feed (...)>
    <title type="text">Products</title>
    <id>http://services.odata.org/OData/OData.svc/Products</id>
    <updated>2010-02-27T20:03:28Z</updated>
    <Link rel="self" title="Products" href="Products" />
    <Entry>
```

```
(...)
</Entry>
</feed>
```

The HTTP response is standard and contains the status line and headers ①. The OData content ② is contained within the HTTP response as body and uses the Atom XML format.

The response contains a feed of products that can be huge. For this reason OData supports pagination and allows selecting a set of data using criteria. Table 8.4 provides examples of query URIs. Further details are available on www.odata.org in the URI conventions page.

Table 8.4 Examples of OData URIs to query data of services

URI	Description
/OData/OData.svc/Products(1)	Returns the first product of the collection.
/OData/OData.svc/Products(1) / Description	Returns the complete description property of the first product of the collection.
/OData/OData.svc/Products(1) / Description/\$value	Returns only the value of the description property of the first product of the collection.
/OData/OData.svc/Products?\$orderby=Rating asc	Returns the product collection in ascending order of the rating property.
/OData/OData.svc/Products?\$top=5	Returns the first five product entries where the corresponding collection is sorted using a scheme determined by the OData service.
/OData/OData.svc/Products?\$select=Price,Name	Selects price and name properties in the product collection.
/OData/OData.svc/Products?\$filter=Price%20lt%2010	Returns products where price property value is below 10.

In addition, OData can manage resources in a RESTful way with the following HTTP methods:

- *POST*—Creates an entity and returns its URI using the Location HTTP header. Entity content is provided through the HTTP request body.
- *PUT*—Updates an entity using the content provided in the HTTP request body.
- *DELETE*—Deletes an entity based on request URI.

The following listing provides an example of a request to add a category to the previous service. The `/OData/ODate.svc/Categories` URI representing all categories must be used in this case.

Listing 8.5 Adding a category with OData using an HTTP POST request

```
POST /OData/OData.svc/Categories HTTP/1.1
Host: services.odata.org
```

① **Uses HTTP POST method**

```

DataServiceVersion: 1.0
MaxDataServiceVersion: 2.0
Accept: application/atom+xml
Content-type: application/atom+xml
Content-Length: 634

```

```

<?xml version="1.0" encoding="utf-8"?>
<Entry (...)>
  <title type="text"></title>
  <updated>2010-02-27T21:36:47Z</updated>
  <author>
    <name />
  </author>
  <category term="DataServiceProviderDemo.Category"
    scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:ID>10</d:ID>
      <d:Name>Clothing</d:Name>
    </m:properties>
  </content>
</Entry>

```

Adding a category is done using an HTTP POST request on the address corresponding to categories ①. The content type for the category to add is specified using the Content-Type header ②. The category content is then specified as body for the request ③. As a side note, optional properties that are missing in the request are automatically set with their default value, as defined by the service.

Once the resource has been created on the server side, the response described in the following listing is returned. This response provides the autogenerated identifier in the Location HTTP header and contains the final state of the resource.

Listing 8.6 Response received for the add request for a category

```

HTTP/1.1 201 Created
Content-Length: 1072
Date: Sat, 27 Feb 2010 21:39:54 GMT
Location: http://services.odata.org/OData/OData.svc/Categories(10)
Content-Type: application/atom+xml; charset=utf-8
DataServiceVersion: 1.0;

```

```

<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<Entry (...)>
  <id>http://services.odata.org/OData/OData.svc/Categories(10)</id>
  <title type="text"></title> ="http://www.w3.org/2005/Atom">
  <id>http://services.odata.org/OData/OData.svc/Categories(10)</id>
  <title type="text"></title>
  ...
</Entry>

```

The response is successful with the status code 201 and message Created ①. The address of the created resource is specified using the Location header ②, and the content is provided as response body ③.

OData provides an SDK with a set of tools in different languages to consume and expose OData services. Because the protocol is based on HTTP and the REST architecture style, Restlet is a natural solution to consume OData services from Java. In the following sections we describe how to use the Restlet support for OData.

8.6.2 Generating classes for access using Restlet

From the client perspective, the Restlet OData extension provides a generation tool that will make your OData life easier. Based on the metadata exposed by any OData service, it will generate, for each declared data entity, the matching Java class with the correct properties and save a lot of time in your development. Parsing and formatting the Atom/XML is done automatically by another generated Service subclass and its internal helpers.

Figure 8.18 provides an overview of the way to use the Restlet OData extension to generate classes to consume OData services.

The OData extension lives in the `org.restlet.ext.odata` package and includes the generator tool. Generated classes support the following features:

- Querying data
- Managing entities
- Projections, similar to database view
- Transparent server-side paging
- Blobs, to expose media resources
- Row counts retrieval
- Customizable Atom feeds
- Version headers
- Operations, to expose stored procedures

The extension is also available on the Restlet edition for Android, allowing you to directly access OData services hosted, for example, on the Azure cloud computing platform, from a smartphone.

Let's reuse the product service described in the previous section and generate OData classes with Restlet support. When using XML-based Atom, you need to use the extensions described in table 8.5 in addition to core Restlet.

The `org.restlet.ext.odata.Generator` class has a main method to launch the generation. This launcher class accepts three parameters to configure the generator as described in table 8.6.

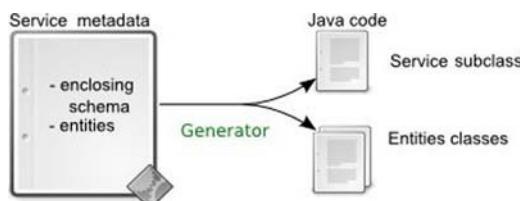


Figure 8.18 OData class generation support provided by Restlet

Table 8.5 Needed extensions for generation in context of OData with XML-based Atom

Extension	Description
org.restlet.ext.odata	Extension corresponding to the OData support and including the generator
org.restlet.ext.atom	Extension providing support for Atom
org.restlet.ext.xml	Extension providing support for XML
org.restlet.ext.freemarker	Extension corresponding to the integration of FreeMarker template engine within Restlet, used for code generation

Table 8.6 Parameters of the generator class

Parameter	Description
SERVICE_URI	Base URI of the remote service.
OUTPUT_DIRECTORY	Directory path where the classes for the service will be generated.
SERVICE_CLASS_NAME	Name for the generated service class. This parameter is optional.

To launch the generation, you can use this Java command:

```
java -cp org.restlet.jar:org.restlet.ext.xml.jar:org.restlet.ext.atom.jar
  :org.restlet.ext.freemarker.jar:org.restlet.ext.odata.jar
  :org.freemarker.jar org.restlet.ext.odata.Generator
  http://services.odata.org/OData/OData.svc out/
```

In order to simplify this generation launch, we created the Ant configuration file shown in the following listing. It configures the classpath based on a lib directory containing all dependencies and also provides a generate task to call the generator.

Listing 8.7 Ant script for launching the OData generation of Restlet

```
<project name="RestletOData" default="generate" basedir=".">
  (...)

  <target name="generate">
    <java classname="org.restlet.ext.odata.Generator">
      <arg value="http://services.odata.org/OData/OData.svc"/>
      <arg value="${basedir}/out"/>
      <classpath>
        <fileset dir="${basedir}/lib">
          <include name="**/*.jar"/>
        </fileset>
      </classpath>
    </java>
  </target>
</project>
```

Ant task for generation

The Ant task called generate uses the OData generator of Restlet to create OData classes for the specified service in the out directory.

Now that these classes have been generated, we describe how to use them to interact with the remote service.

8.6.3 Calling OData services

In the previous section you generated all the classes for the OData service based on its root URI. We'll explain how to use them to manipulate the data. Let's first talk about the generated classes. We distinguish two kinds of classes:

- *Data*—There are data classes for each entity supported by the service. They correspond to POJOs with a property for each data attribute.
- *Service*—This acts as a manager for a specific remote OData service. OData services are stateless, but service instances aren't. State on the client is maintained between interactions as a local cache in order to support features such as update management.

Now you'll find out how to use these classes to interact with the service. We begin with the query support. For each entity, a method is created, called `create<ENTITY>Query`, that returns a `Query` object from a given search URI path. The following snippet describes how to have access to all elements of the type `Product`:

```
Query<Product> queryProduct = service.createProductQuery ("/Products");
for (Product product : queryProduct) {
    String id = product.getId();
    String description = product.getDescription();
}
```

Notice here that the OData syntax for a query can be used within the URI path. For example, you can retrieve the first `Product` item with the following snippet:

```
Query<Product> queryProduct = service.createProductQuery ("/Products(1)");
Product product = queryProduct.iterator().next();
```

Some advanced features are also supported based on properties provided by the `Query` class (listed in table 8.7) that support complex use cases for implementing OData queries.

Table 8.7 Properties provided by the Query class

Method	Description
<code>expand</code>	Specifies that the query needs to return matching elements including their dependencies. The value defines which property to expand.
<code>orderby</code>	Allows ordering the result of the query based on a data property.
<code>filter</code>	Applies filtering constraints using the Language-Integrated Query (LINQ) on a set of data. Its syntax is very close to SQL's.
<code>skip</code>	Takes one value which corresponds to the number of the start entities to omit in the set of data theoretically returned by the query.

Table 8.7 Properties provided by the Query class (continued)

Method	Description
top	Takes a numeric value which represents the maximum number of results that the query will return.
select	Specifies which properties will be returned for elements matching the query. Transitivity in property names is supported here.
inlineCount	Enables the inlineCount feature if the service supports it. In this case the count of returned element is obtained from the feed document itself. This allows retrieval of both count data and entries in the same request.
getCount	Gets the number of retrieved elements for a query.

All these properties can be combined on the same query. Let's describe the features provided, beginning with filtering.

The filter method must be called on the Query object created with the createProductQuery method. The parameter of this first method corresponds to the query string. In the following snippet, we only select items whose Name property value is "Milk":

```
Query<Product> query = service.createProductQuery(
    "/Products") .filter("Name eq 'Milk'");
```

The skip and top properties allow the client to control pagination. The first one specifies the number of entities to be skipped by the server in response to a query, and the other one defines the maximum number of entities to be received. The following snippet describes how to get page two containing at most 20 products:

```
Query<Product> queryProduct = service.createProductQuery(
    "/Products") .skip(20) .top(20);
```

Another feature corresponds to projection using the select method. The latter allows specifying properties that will only be returned. At this level, transitive expressions are supported in order to browse inner elements of data. The following snippet describes how to use the select method:

```
Query<Product> queryProduct = service.createItemQuery(
    "/Products") .select("Description");
```

In addition to queries, the Restlet OData support can modify the data of the remote service: the addProduct method for adding, updateProduct for updating, and deleteProduct for deletion. The following listing describes a complete data management lifecycle based on these methods.

Listing 8.8 Implementing the CRUD methods to manage entities of OData service

```
Product product = new Product();
product.setId("3");
product.setName("My product");
```

```

product.setDescription("The description of my product");
service.addProduct(product);                                ① Adds product

Query<Product> query = service.createProductQuery ("/Products('3')");
product = query.iterator().next();

product.setDescription("Another name");
product.setDescription("Another description");
service.updateProduct(product);                            ② Updates product

service.deleteProduct(product);                          ③ Removes product

```

After creating an instance of type `Product` and setting its identifier, you call the `addProduct` method ① on the service instance to add it. You can also update this item and the changes persisted using the `updateProduct` method ②. The `deleteProduct` method ③ removes it. Unfortunately you can't test such features with the read-only online test OData service.

As you can see, Restlet provides a very convenient client façade in order to access and use OData services by hiding all complexity and technical plumbing. Restlet also takes care of the painful and time-consuming work of generating data objects for the target service.

We end our look at access to static resources in the cloud by tackling the case of S3 resources provided by Amazon AWS or a compatible provider.

8.7 Accessing Amazon S3 resources

As described briefly in section 8.3 on Amazon Elastic Beanstalk, Simple Storage Service (S3) is the online storage web service provided by Amazon Web Services allowing storing data in the cloud.

You can access and manage S3 resources through simple HTTP requests, but they rely on a custom security scheme and extension HTTP headers.

8.7.1 Configuring a bucket

A *bucket* corresponds to a place in S3 where you can store resources, such as a folder if you prefer. Before trying to access and manage elements, you first need to create a bucket. To do that, go into the S3 tab in the AWS Management Console and click the Create Bucket button. Once that's done, you can upload a file in the newly created bucket—for example, the picture corresponding to the cover of the book.

Figure 8.19 shows the AWS Management Console after the creation of a bucket called `reiabucket` and the upload of the `louvel_cover150.jpg` file.

You're now almost ready to access and manage elements in the bucket using HTTP requests through Restlet.

8.7.2 Accessing a resource with the bucket

Amazon S3 uses a dedicated security mechanism to authenticate requests. This mechanism is based on secret keys that are known on both client and server sides and used to compute signatures of requests. A request is authenticated when computed signatures

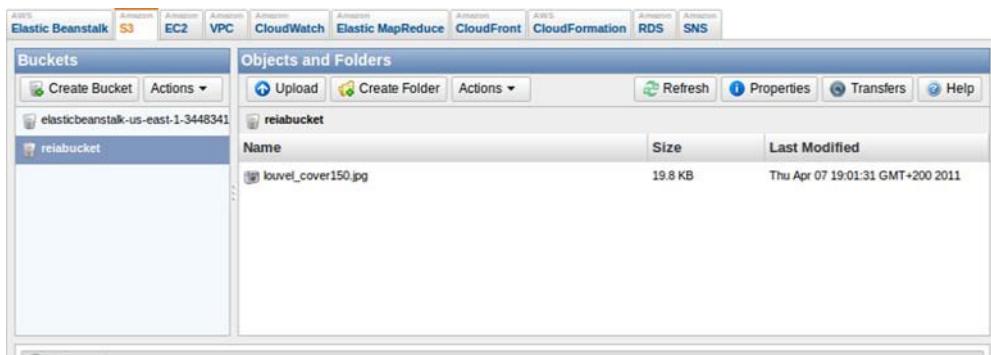


Figure 8.19 AWS Management Console describing the reiabucket element and its elements

Created	Access Key ID	Secret Access Key	Status
February 15, 2011	[REDACTED]	Show	Active (Make Inactive)

Figure 8.20 Access Keys tab provides all available access keys.

on both sides match. Authentication hints are sent within the request using the Authorization header following this syntax:

```
Authorization: AWS AWSAccessKeyId:<Signature>
```

You can access the AWS access key identifier in your AWS account in the Security Credentials section. The latter gives you access to all keys associated to your account and corresponding secret keys. Such keys are useful to sign requests based on their content. Figure 8.20 shows the Access Keys tab in the AWS account providing all available access keys.

Using Restlet, you don't have to manage the Authorization header by yourself because the AWS S3 security scheme is automatically supported. This support is logically located in the org.restlet.ext.crypto extension, as it requires encryption.

This kind of security is enabled when specifying the HTTP_AWS_S3 challenge scheme in configuration of the challenge response. The following listing gives an example that downloads the picture uploaded in the previous section and writes it to a local file.

Listing 8.9 Using AWS S3 security in HTTP requests with Restlet

```
String rootUri = "https://s3-eu-west-1.amazonaws.com/reiabucket/";
ClientResource resource = new ClientResource(
    rootUri + "louvel_cover150.jpg");
```

```

resource.setChallengeResponse(
    new ChallengeResponse(
        ChallengeScheme.HTTP_AWS_S3,
        "<AWSAccessKeyId>", "<AWSSecretKey>"));

FileOutputStream fos = new FileOutputStream(
    new File("picture.jpg"));
resource.get().write(fos);
fos.close();

```

① Sets AWS S3 security hints
② Execute request, retrieve content

You set hints for AWS S3 security using the challenge response that can be specified on the request instance using its `setChallengeResponse` method ①. The `HTTP_AWS_S3` scheme must be specified here in the `ChallengeResponse`. In our context, its identifier corresponds to the AWS access key identifier and the password to the secret key that will be used to sign the request. The request can then be executed as usual and the content present in the response extracted through its associated representation ②.

Amazon S3 service also supports other HTTP methods to remotely manage the resource. Using a PUT method will update the resource on the server (or create it if it doesn't exist), and DELETE will remove it. The following listing describes how to remotely create and delete S3 resources within a bucket using Restlet.

Listing 8.10 Remotely create and delete S3 resources using Restlet

```

String rootUri = "https://s3-eu-west-1.amazonaws.com/reiabucket/";
ClientResource resource = new ClientResource(rootUri + "resource.txt");
resource.setChallengeResponse(
    new ChallengeResponse(
        ChallengeScheme.HTTP_AWS_S3,
        "<AWSAccessKeyId>", "<AWSSecretKey>"));

resource.put("resource content", MediaType.TEXT_PLAIN);
(...)

resource.delete();

```

① Configure, execute creation request
② Configure, execute deletion request

As you can see for creation ①, you need to specify the resource content using the representation associated to the request. Deletion is like getting a resource but using the HTTP DELETE method ②. For both requests, Amazon S3 security is applied, as in listing 8.10, based on challenge response.

You can configure permissions to access and manage resources within buckets in the Permissions tab of bucket properties within the AWS Management Console, as shown in figure 8.21. For each Amazon account, you can specify which operations (list, upload, and delete) are possible.

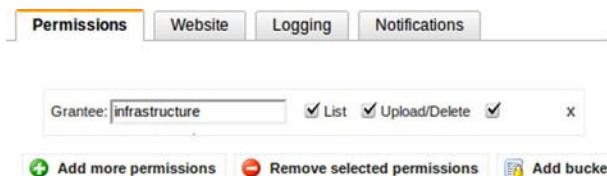


Figure 8.21 Configuration of permissions for AWS account to access and manage resources in buckets

The screenshot shows the Azure portal interface. On the left, there's a navigation menu with 'Storage' selected. In the center, a table lists storage accounts under the heading 'Filter storage accounts'. The columns are 'Name', 'Type', 'Status', and 'Last updated'. One account, 'reia', is listed as a 'Storage account' in 'Active' status, last updated on 5/13/2011 at 9:08:40 AM UTC. To the right of the table is a 'Properties' panel. It contains several sections with configuration values: 'Primary access key' (with a 'View' button), 'Secondary access key' (with a 'View' button), 'Blob URL' (reia.blob.core.windows.net), 'Table URL' (reia.table.core.windows.net), 'Queue URL' (reia.queue.core.windows.net), and 'Name' (reia).

Name	Type	Status	Last updated	Properties
Noelios - Infrastructure	Subscription	Active		
reia	Storage account	Created	5/13/2011 9:08:40 AM UTC	Primary access key <Hidden> View Secondary access key <Hidden> View Blob URL reia.blob.core.windows.net Table URL reia.table.core.windows.net Queue URL reia.queue.core.windows.net Name reia

Figure 8.22 List of storage accounts present in the Azure console for a subscription

Since version 2.1, Restlet also implements S3 server-side authentication. This makes it possible to use AWS security for your applications even outside of the Amazon S3 platform. Microsoft also provides online services for storage that you can access in a way similar to AWS S3 using Restlet. In the next section we describe how to use them.

8.8 Accessing Azure services

Windows Azure offers online services requiring a security scheme similar to AWS, based on secret keys and extension HTTP headers. Restlet also supports these schemes via its crypto extension.

Before going through how to use them, let's configure a storage account within the Windows Azure console.

8.8.1 Configuring storage accounts

Configuring a storage account is simple. After selecting Storage accounts in the left-hand menu, click the Create storage account button. After entering a name, the account is created and appears in the list, as shown in figure 8.22.

The Properties tab on the right of the screen gives access addresses for the different storage services. For the table service, in this case, the address is `reia.table.core.windows.net`. Let's find out how to use this service using Restlet.

8.8.2 Using table service

Windows Azure provides three kinds of online storage services usable from a storage account:

- *Blob service*—Stores text and binary data through three kinds of resources (the storage account, containers, and blobs).

- *Queue service*—Stores messages that may be read by any client with access to the storage account.
- *Table service*—Offers structured storage in the form of tables.

In this section we describe how to use the table service, providing three kinds of operations. The first one consists in querying the list for all available tables. Two other ones implement table management to create and delete tables.

As with S3 services, the Azure storage services require a secret key-based authentication. This key is used to sign requests. Authentication details are sent within the request using the Authorization header following the standard syntax. This kind of authentication is automatically supported when specifying the `HTTP_AZURE_SHAREDKEY` challenge scheme in the configuration of the challenge response. The next listing describes how to integrate `HTTP_AZURE_SHAREDKEY` security in HTTP requests with Restlet. The code gets the list of tables for the storage account.

Listing 8.11 Using Azure secret key security in HTTP requests with Restlet

```
String rootUri = "http://reia.table.core.windows.net/Tables";
ClientResource resource = new ClientResource(rootUri);
resource.setChallengeResponse(
    new ChallengeResponse(
        ChallengeScheme.HTTP_AZURE_SHAREDKEY,
        "<StorageAccount>", "<StorageAccountSecretKey>"));

Representation tableList = resource.get();
Feed feedTableList = new Feed(tableList);
for (Entry entry : feedTableList.getEntries()) {
    System.out.println("id = "+entry.getId());
}
```

1 Set Azure security info
2 Execute request, retrieve content
3 Display table URIs

After instantiating the Restlet client resource, you set the Azure security info using a challenge response ①, then execute the GET request ② that returns the list of tables. Azure table service uses Atom format as content type. For this reason, we use the Restlet Atom extension to get a Feed representation and iterate over its entries ③.

The REST API of the service also allows creating a table with an HTTP POST method. The table name is specified through the sent content using an Atom entry element. The following listing shows how to create a table within the table service.

Listing 8.12 Creating a table using the REST API of the table service

```
String tableName = "tableName";
Entry entry = new Entry(){
    public void write(XmlWriter writer) throws IOException {
        writer.forceNSDecl(
            "http://schemas.microsoft.com/ado/2007/08/dataservices", "d");
        writer.forceNSDecl(
            "http://schemas.microsoft.com/ado/2007/08/dataservices/metadata",
            "m");
        super.write(writer);
    }
}
```

```

};

entry.setUpdated(new Date());

Content content = new Content();
content.setInlineContent(new StringRepresentation(
    "<m:properties><d:TableName>tableName</d:TableName></m:properties>",
    MediaType.APPLICATION_XML));
content.setToEncode(false);
entry.setContent(content);
String endpoint = "http://reiatest.table.core.windows.net/Tables";
ClientResource clientResource = new ClientResource(endpoint);
resource.setChallengeResponse(
    new ChallengeResponse(
        ChallengeScheme.HTTP_AZURE_SHAREDKEY,
        "<StorageAccount>", "<StorageAccountSecretKey>"));
Representation representation
    = clientResource.post(entry);
Entry responseEntry = new Entry(representation);
String id = responseEntry.getId();

```

The delete operation is also supported through the HTTP DELETE method. In the case of a table with the name myTable, the address to use is `http://myaccount.table.core.windows.net/Tables('mytable')`.

You've seen how to access and manage tables hosted by Azure using Restlet. Let's move on to another interesting feature recently provided by the framework in order to give access to intranet resources from the cloud with Restlet's SDC extension.

8.9 Accessing intranet resources with Restlet's SDC extension

When providing applications in the cloud, companies may be reluctant to expose their data to the world because of security concerns, but sharing is sometimes necessary. Giving access to internal applications or the company's data is risky because they correspond to the core of the business and can open a door to potential attacks from the outside. On the other hand, sharing this data with users and partners through web APIs is often important strategically.

For such use cases, Google provides an interesting and secure solution called Secure Data Connector (SDC). In this section we describe how this technology works and what its main benefits are. We show how it's tied to Google's cloud platform and present the Restlet SDC extension.

8.9.1 Secure Data Connector overview

Google provides a robust solution called SDC to enable accessing data behind a corporate firewall in a secure and unobtrusive way from a public cloud platform. Its approach doesn't require giving full access to the company's backend by opening the firewall or the DMZ but implements a secure data tunnel between the service exposed in the cloud and the intranet.

The key principle of this technology is that the connection isn't initiated from outside but from the intranet. The firewall always remains active, isn't reconfigured, and

still protects the intranet. The SDC agent keeps full control over what's accessed and provides a secure tunnel to communicate with dedicated applications. This mechanism is implemented with two distinct parts:

- *Google SDC agent*—Corresponds to the software executed within the intranet receiving requests from applications exposed through the cloud.
- *Google SDC tunnel server*—Corresponds to the software that associates request domains with corresponding connections. It also forwards requests to the intranet using the matching tunnel connection.

Figure 8.23 illustrates all SDC-related parts involved in both the Google cloud platform and the user's company intranet.

Because the connection is initiated by the SDC agent, it can pass the firewall. Communications can then happen both ways (from cloud platform to intranet and vice versa). Google's Protocol Buffer provides automated and optimized formatting and parsing as well as multiplexing over a single TLS socket per active tunnel.

Now let's see how to use the SDC agent provided by Google as an open source project.

8.9.2 *Installing SDC agent*

As you can see from the overview, SDC can't work without the SDC agent running within the company intranet. In this section we focus on the open source project provided by Google at <http://code.google.com/securedataconnector/>.

The SDC agent is responsible for the secure bridge between the company intranet and the public cloud platform. It must be installed and run within the intranet and consists of a single Java application.

First, download the zip file for the chosen release. In this case we use the latest version, 1.3 RC2. After extracting this file, you have access to the JAR file of the tool and startup and shutdown scripts for Linux. Don't be afraid if you use Windows because the tool is a Java application and can be executed on any platform that supports a recent Java SE 6 runtime. Start up the agent by executing the following command line:

```
java -jar lib/sdc-agent.jar -log4jPropertiesFile config/log4j.properties -
rulesFile config/resourceRules.xml -localConfigFile config/
localConfig.xml
```

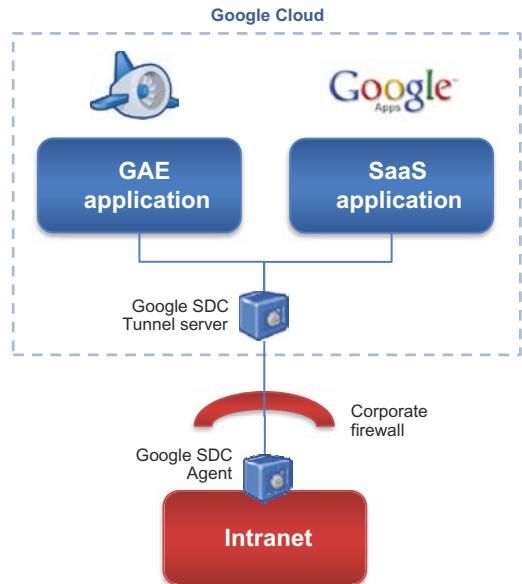


Figure 8.23 All involved parts to implement SDC technology in both Google platform and intranet

Now that you know how to start up the SDC agent process, let's dive into what parameters you can use to configure it—in particular, the access authorizations. Table 8.8 lists its main launching parameters.

Table 8.8 Main launching parameters of the SDC agent application

Parameter	Description
agentId	Specifies an identifier for the running agent process sent to the SDC tunnel server.
debug	Enables debug messages.
domain	Specifies the SDC domain name, sent to the SDC tunnel server.
localConfigFile	Specifies the file to configure the SDC agent and its connection to Google Apps.
log4jPropertiesFile	Specifies the file to configure log4j trace messages.
rulesFile	Specifies the file to configure which Google applications users and which components can access which resources within the domain associated to the agent.

As you can see in table 8.8, the SDC agent process is based on two main files for configuration. The first one, by default called localConfig.xml, aims to configure the SDC agent and its connection to the SDC server, typically in the Google cloud. The following listing describes how to specify hints regarding SDC server to access, the domain, user, and agent identifier for the agent process.

Listing 8.13 Global configuration of the SDC agent

```
<entity>
    <sdcServerHost>localhost</sdcServerHost>
    <sdcServerPort>4433</sdcServerPort>
    <allowUnverifiedCertificates>true</allowUnverifiedCertificates>
    <domain>example.com</domain>
    <user>myUser</user>
    <password>myPassword</password>
    <agentId>myAgent1</agentId>
    <socksServerPort>1080</socksServerPort>
    <healthCheckGadgetUsers></healthCheckGadgetUsers>
</entity>
```

The first properties have to do with the remote SDC tunnel server and how to access it ①. You then define the domain associated with the agent ②. Requests proxied with this domain from the client side will be routed to this agent. Finally you specify properties regarding the agent itself, such as its identifier and corresponding user and password ③. When used by the Google cloud, the sdcServerHost value must be apps-secure-data-connector.google.com and the sdcServerPort value 443.

The second configuration file, called resourceRules.xml, configures which client application users and which components can access which resources within your

domain. The following listing describes how to configure rules to give access to users for domains.

Listing 8.14 Rule configuration to access intranet applications through SDC agent

```
<resourceRules>
    <rule repeatable="true">
        <ruleNum>1</ruleNum>
        <agentId>all</agentId>
        <allowDomainViewers>true</allowDomainViewers>
        <apps repeatable="true">
            <service>AppEngine</service>
            <allowAnyAppId>true</allowAnyAppId>
        </apps>
        <url>http://www.restlet.org/</url>
        <urlMatch>HOSTPORT</urlMatch>
    </rule>
</resourceRules>
```

The configuration file contains a set of rules ① that defines how the intranet can be accessed. You can see the URL that will become accessible in the intranet ②. You can also specify authorized applications and the agents in the configuration.

You now have enough information to understand what SDC is and how to configure the intranet agent. Let's see how Restlet can help you use this technology from cloud platforms other than GAE.

8.9.3 Using the Restlet SDC connector

Google SDC is a perfect match if you exclusively use applications provided by the Google ecosystem, but the tool comes with several limitations:

- The SDC agent is available as an open source project, but that's not the case with the SDC Tunnel Server part.
- The Google App Engine SDK doesn't provide a way to test SDC locally without deploying your application in the cloud.
- This technology isn't portable because it can't be used with other cloud platforms such as Amazon EC2 and Windows Azure.
- You can't easily port a GAE application using SDC to another platform, private cloud, or public cloud and remain locked on GAE.

Because one of the Restlet Framework goals is to favor portability across various PaaS offerings, those SDC challenges were compelling in order to make figure 8.24 become a reality.

In order to make the SDC technology usable in other cloud platforms, Restlet provides an alternative open source implementation of the SDC tunnel server. It supports the same SDC protocol by using the Protocol Buffer library of Google's open source SDC agent in order to implement a multiplexing tunnel (frames going both ways without constraint) over TLS.

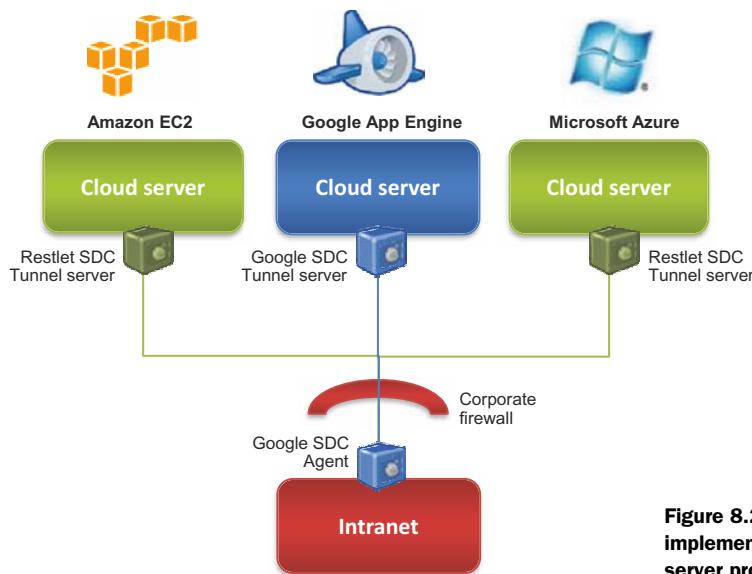


Figure 8.24 Cross cloud platform implementation of the SDC tunnel server provided by Restlet

This server is implemented within the `org.restlet.ext.sdc` extension of Restlet providing a specific client connector that embeds a SDC tunnel server. This extension has been available since version 2.1 and later in the Restlet Java SE, Java EE, and OSGi editions. Putting the corresponding JAR file in the classpath registers the SDC connector with the Restlet engine. You can then use it with its protocol identifier SDC, as in the following snippet:

```
Client sdcClient = new Client(new Context(), Protocol.SDC);
(...)
sdcClient.start();
```

The tunnel server is automatically started at connector startup when its `start` method is called. Some configuration is needed at this level regarding encryption:

```
Client sdcClient = new Client(new Context(), Protocol.SDC);
Series<Parameter> parameters = sdcClient.getContext().getParameters();
parameters.add("keystorePath", "sdc.keystore");
parameters.add("keystorePassword", "password");
parameters.add("enabledCipherSuites", "TLS_RSA_WITH_AES_128_CBC_SHA");
parameters.add("sslProtocol", "TLSv1");
sdcClient.start();
```

An important thing to remember is that this connector needs to be started before launching the SDC agent because it will immediately attempt to reach the tunnel server. Now that these two steps are done, you can use the SDC tunnel to execute an HTTP request using Restlet. The thing to understand here is the way to route requests through the SDC connector. In this context you need to explicitly set the SDC protocol and override the one deduced from the HTTP address by Restlet. This aspect can be a bit disturbing at first sight, because you usually leave Restlet to automatically find out protocol from request URI, but that isn't suitable in this case.

The other required configuration is to specify which domain the request is executing for. The tunnel server checks the correspondence between the domain and actual tunnel connection. You can do that using the Restlet proxy challenge response. Within it, you can set the SDC protocol name, the user of the domain, and a password. Apart from these aspects, the Restlet client support is usable in the same way as regular HTTP clients. The following snippet shows how to execute HTTP requests using the Request and Response objects through a SDC tunnel:

```
ClientResource cr = new ClientResource("http://www.restlet.org");
cr.setProtocol(Protocol.SDC);
cr.setProxyChallengeResponse(ChallengeScheme.SDC, "myUser@example.com",
                            "myPassword");
cr.get().write(System.out);
```

To improve code source portability, the GAE edition supports a similar syntax even though in this case there's no need for the SDC extension at all, because it's based on a native GAE feature.

8.9.4 Restlet SDC support in GAE edition

Within classic GAE applications, you normally have to specify the “use_intranet” header with the value `true` for this purpose. The following snippet shows how to set this header using the `HttpURLConnection` class:

```
URL url = new URL("http://www.restlet.org");
HttpURLConnection connection
    = (HttpURLConnection) url.openConnection();
connection.setRequestProperty("use_intranet", "true");
```



Since version 2.1 of Restlet's edition for GAE, this is now done transparently without having to set this header. Specifying the SDC protocol for the request is sufficient to use the corresponding secured tunnel. Internally Restlet adds the header when the SDC protocol is specified and ignores the proxy authorization property.

The selection of the tunnel is based in this case on the domain specified for your GAE application. Notice that the Restlet SDC provides more flexibility because it allows the use of several domains within the same application.

That closes this chapter on using Restlet in the cloud. This support by Restlet provides further perspectives for the SDC technology created by Google.

8.10 Summary

Restlet provides comprehensive support for both exposing and consuming RESTful applications in the cloud. It can be seen as a RESTful middleware to allow interactions between heterogeneous cloud client and server applications.

Based on its multiple editions and associated extensions, Restlet provides a convenient way to handle environment specifics and restrictions and improve your application portability. You learned in this chapter how to deploy Restlet applications in the most popular cloud platforms: Google App Engine, Amazon Elastic Beanstalk, and Windows Azure.

In addition, you saw how Restlet facilitates access to a wide range of RESTful web services. They can be consumed using the standard Restlet machinery, but Restlet provides additional support for OData-based and S3 services. Restlet provides support for Azure custom security schemes, code generation of data objects, and a service proxy class to simplify its use. In the case of S3, it provides support for the custom security scheme on both the client and server side.

Restlet provides SDC support that enables the use of this technology in all cloud platforms, not just Google's. SDC gives access to intranet resources located behind corporate firewalls, from a public cloud and in a highly secure way.

As a result you can increase the portability of Restlet applications across cloud infrastructures, but this doesn't stop here because other editions for Android and GWT also make using the Restlet API in web navigators and mobile devices possible and consistent. We focus on that topic in the next chapter.



Using Restlet in browsers and mobile devices

This chapter covers

- Restlet editions for GWT and Android
- Using REST within GWT applications with Restlet
- Using REST within Android-based mobile devices with Restlet

This chapter explains how to access web APIs using Restlet from clients ranging from light web browsers to mobile devices. First, it's possible to use the Restlet API within a browser, with no additional plug-ins, via the Google Web Toolkit (GWT). This chapter discusses the Restlet edition for GWT, a port of the Restlet client-side API to GWT, and how to use Restlet on the server side to address issues such as cross-domain calls and automatic object serialization. This approach allows Rich Internet Applications (RIA) to consume and modify REST resources.

We then look at the Android OS for mobile devices. It runs a Dalvik virtual machine that understands code compiled from Java source and provides a subset of Java SE APIs, plus specific APIs. We describe how the Restlet edition for Android makes it easy to access the web APIs from Android applications.

For each technology we provide basic introductions, describing concepts and how to use them. No prior knowledge is required for these technologies before

reading this chapter. Moreover we always follow an “in action” approach by developing our mail application as a case study for both technologies. In the case of the GWT technology, this consists of a mail client which calls our example mail server to retrieve and send emails. For Android, we implement a mobile mail program with both client- and server-side applications.

We begin by describing what GWT is and how REST can be used within it.

9.1 Understanding GWT

Implementing AJAX or RIA can be a painful task. Such applications require advanced skills in several different technologies such as HTML, CSS, and JavaScript. Many developers have basic skills in these areas, but things become harder with advanced concepts, like connecting client and server with AJAX.

The situation becomes even more problematic when trying to handle browser specifics. Making applications have a consistent behavior across different browsers is a tedious task. Because JavaScript is dynamically and weakly typed, debugging is more complex than when using strongly typed languages like Java. Moreover such technologies don't benefit from integrated development environments as advanced as those for the Java platform.

The GWT enhances productivity while developing high-performance web applications without having strong knowledge of technologies involved in browsers. In this section we give an overview of the technology and describe how to implement it. Finally we tackle how to make it consume web APIs.

9.1.1 GWT overview

With GWT, you can continue to develop your web applications using the Java language on both the browser side and on the server side. Implementing browser-side processing in Java without relying on an Applet plug-in is unusual. It doesn't mean that the code will be executed in the same way as with an Applet. In fact, GWT compiles this code into JavaScript code that will be executed natively by the browser without any additional plug-in. Figure 9.1 describes this mechanism.

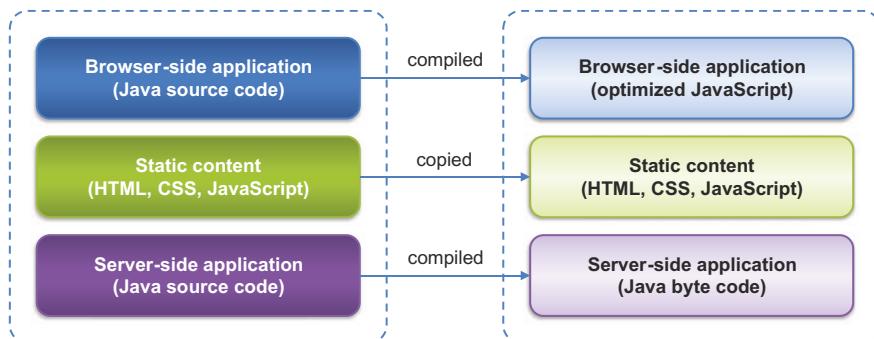


Figure 9.1 GWT application mechanisms

The main benefit of this approach is that GWT can apply optimizations for distinct browser types at compilation time. Browser-specific idiosyncrasies are taken into account while still providing the same behavior. Furthermore, this complex task is done transparently for developers, contributing to the magic of this solution.

Figure 9.1 shows a GWT application using Java on the server side. But once compiled into JavaScript, the client side of a GWT application corresponds to a set of static JavaScript files that could be served from any web server, such as PHP or Python.

Let's go deeper inside the organization of a GWT application. Such an application is composed of a set of GWT modules that follow a particular file structure. Here's how a module is composed:

- *Java classes related to the UI, data objects, and remote service interfaces*—All these classes will be compiled into JavaScript in order to be natively executed within web browsers.
- *Java classes related to remote services used within the module*—They respond to AJAX requests triggered from the client side of the GWT module, and corresponding classes will be typically executed within a Java server application.
- *Web resources*—CSS style sheets, HTML pages, and additional JavaScript code required by the module.
- *A configuration file per module*—This defines GWT elements used and the entry point class.

In order to provide these files, GWT requires the following structure:

- Root package for the module can be freely named.
- Packages corresponding to the client side must be located by default under a `client` subpackage of the module package.
- Packages corresponding to the server side must be located by default under a `server` subpackage of the module package.
- The folder for web resources must be under the `public` one right below the root package.
- A module called `MODULE-NAME` is described by a configuration file named `<MODULE-NAME>.gwt.xml` located directly under the module package.

Now we'll describe how to install and use it.

9.1.2 *Installing and using GWT*

GWT provides a dedicated Software Development Kit (SDK) which consists of a set of JAR files and command-line tools to create applications and tune them. This SDK is provided as a standalone tool that can be downloaded from the GWT website on Google Code at <http://code.google.com/webtoolkit/>.

Although you can develop GWT applications with this SDK and any Java IDE, we chose Eclipse because Google offers a convenient plug-in for it. Here are the additional features it provides:

- Wizards to create and configure the project files necessary for GWT applications
- Efficiently build GWT applications, execute them, and debug them
- Dedicated editors for GWT artifacts

To install this tool, you can use the Update Manager of Eclipse, which is available from the Help > Install new Software menu. Then you need to specify the Update Site for the Google Plugin using <http://dl.google.com/eclipse/plugin/3.6> for version 3.6 of Eclipse. (Change the number at the end of the address for another version of Eclipse.) Next you can see the installable items, as shown in figure 9.2.

As you can see, the SDK is also available through the update site, so you don't need to download and install it separately. You can select the required Google Plugin for Eclipse and even additional ones such as the visual GWT Designer; then launch the installation. After a restart of Eclipse, the Google Plugin for Eclipse with GWT support is available within your workspace.

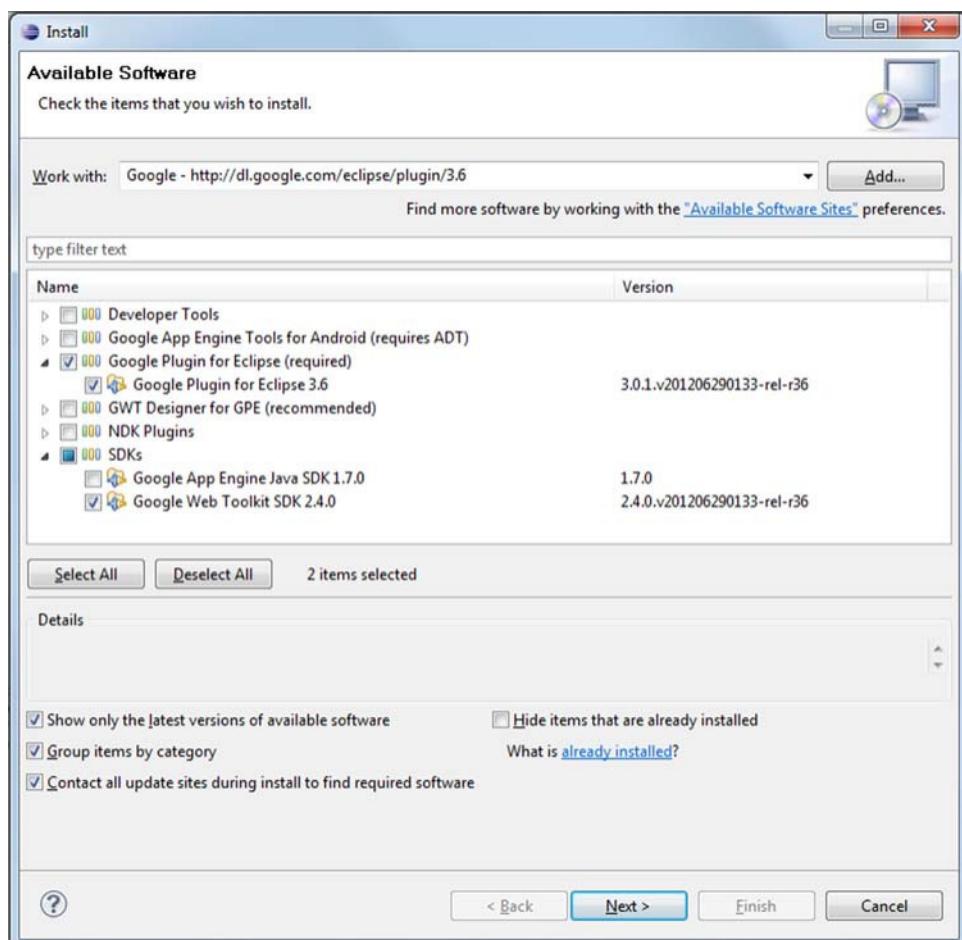


Figure 9.2 Google Plugin for Eclipse installation through Eclipse's update manager tool

9.1.3 GWT and REST

As you've seen in previous chapters, REST provides much flexibility between the client- and server-side resources in the way they interact.

GWT natively integrates mechanisms to execute remote calls using a built-in proprietary protocol called GWT-RPC. It requires the use of a custom object serialization format to exchange content. This structure is specific to the GWT technology and isn't compatible with the standard Java object serialization format. By default, GWT favors the use of its GWT-RPC technology to communicate between the browser and the back end, relying on GWT-provided Servlets on the server side to interact with remote methods. This clearly discourages you from using non-GWT-based server-side services such as RESTful web APIs and technologies other than Java. Figure 9.3 illustrates this GWT-RPC architecture.

The first concern with this solution is that it relies on the RPC paradigm, which has many drawbacks compared to REST, as explained in appendix C. In particular, when RESTfully using HTTP, you can benefit from content negotiation to select among several representation media types, reducing the coupling between clients and specific server-side technologies—the opposite of the GWT-RPC architecture. Figure 9.4 illustrates how GWT applications can access REST resources without using GWT-RPC-based calls, relying instead on the RequestBuilder class in GWT's HTTP module.

We dive deeper into these aspects in the next section. In particular we demonstrate the flexibility offered by Restlet Framework on the GWT client side in section 9.2.2.

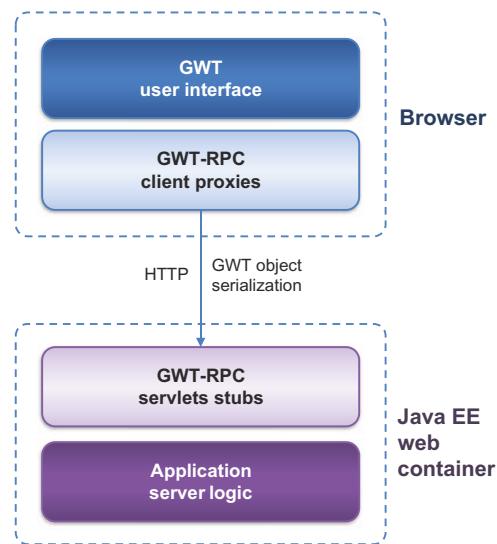


Figure 9.3 Default GWT remoting support

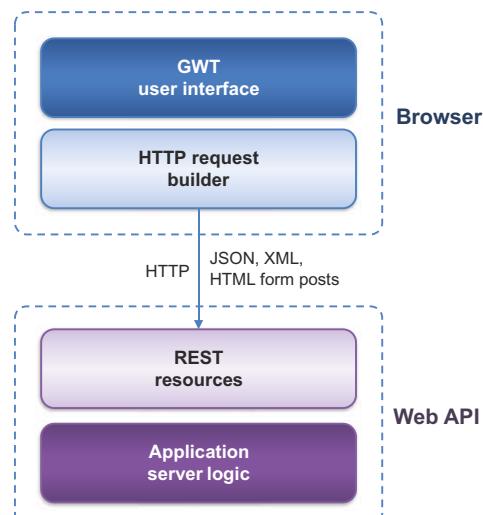


Figure 9.4 Using REST resources from GWT applications without GWT-RPC

9.2 The Restlet edition for GWT

As introduced before, Restlet provides different editions for various execution environments. At this point in the book you've seen the Java SE, Java EE, OSGi, and GAE editions, but Restlet also provides a dedicated edition for GWT. In this case only a limited part of the Restlet API is provided, including most of the client side, due to limitations inherent to browsers.

In this section we describe the Restlet support on the GWT client side. First we introduce classes provided by GWT to execute HTTP requests and then we show how to use them to call REST resources.

9.2.1 The RequestBuilder class of GWT

As described in section 9.1.3, you need to bypass the remote procedure call mechanisms of GWT in order to access RESTful origin servers and reuse an existing web API.

You need to use the low-level Java API provided by GWT to execute HTTP calls without the RPC abstraction layer. For this purpose, GWT has the `RequestBuilder` class supporting all HTTP methods and providing two ways of handling requests. The first way uses the *command* pattern. After instantiating the `RequestBuilder`, you set request parameters and the response callback. Then execute the request with the `send` method. In the second approach, you provide a callback method when executing the request using the `sendRequest` method, which also accepts data to send as parameter. (In either case you need to specify headers before calling the method.)

Listing 9.1 describes how to use this second approach to call a remote web resource with the GET method.

Listing 9.1 Using RequestBuilder class

```
RequestBuilder requestBuilder
    = new RequestBuilder(RequestBuilder.GET, "/accounts/10");
requestBuilder.setHeader("Accept", "application/json");
requestBuilder.sendRequest(null, new RequestCallback() {
    public void onResponseReceived(
        com.google.gwt.http.client.Request request,
        com.google.gwt.http.client.Response response) {
        (...)}
    }
    public void onError(
        com.google.gwt.http.client.Request request,
        Throwable exception) {
        (...)}
});
```

The diagram illustrates the execution flow of the code. It starts with step 1, 'Initialize, execute HTTP request', which points to the final line of the code where the `sendRequest` method is called. From there, it branches into two parallel paths: step 2, 'Handle successful response', which follows the execution of the `onResponseReceived` callback, and step 3, 'Handle error', which follows the execution of the `onError` callback.

First you instantiate the `RequestBuilder` class and provide it the HTTP method to use and the URL of the resource to address. After specifying HTTP headers, send the request using the `sendRequest` method ①, passing a callback object including a

method invoked in case of success ② of the HTTP request and another in case of error ③.

The `RequestBuilder` class is close to the HTTP protocol. It allows you to completely define request content and to extract content from the corresponding response. Using it can be painful because it requires knowledge of all the subtleties of the HTTP protocol such as how to format and parse HTTP headers. In addition, as strange as it sounds, `RequestBuilder` doesn't natively support HTTP Basic authentication, which requires a Base 64 algorithm implementation to encode the header value.

As you can see, using this class in the context of REST would be tedious when trying to take advantage of all the capabilities such as authentication, content negotiation, partial requests, conditional requests, or precise cache control. This is where the Restlet API brings its value to GWT applications with a dedicated edition built on top of `RequestBuilder`.

9.2.2 Restlet port to GWT

Although GWT offers support for low-level HTTP calls, it doesn't help much when you want to use HTTP features such as content negotiation, ranged questions, or cache management. In addition, it lacks an easy way to automate representation serialization and deserialization at the Java class level, leading many developers to use GWT-RPC instead, requiring a GWT-specific backend, when a RESTful web API can talk with any client, not only GWT ones.

There is a need for a higher-level API and framework to solve those issues. As you've guessed, this is exactly what the Restlet Framework edition for GWT provides! In this section you'll see the ideas behind the Restlet port for GWT and how to call REST resources in action.

CONCEPTS

As described in previous sections, GWT lets you develop using the Java language, but because it runs inside the web browser with no plug-in, it has to be compiled into JavaScript before being executed, limiting the number of Java SE APIs supported.

Restlet port to GWT is an adaptation of Restlet's client-side API to work on top of GWT's low-level HTTP client API, as shown in figure 9.5. This edition doesn't include the server-side parts of Restlet and comes with a limited set of extensions. In addition, on the server side the full Restlet API from other editions can be used and can even benefit from a server-side GWT extension that we cover later on.

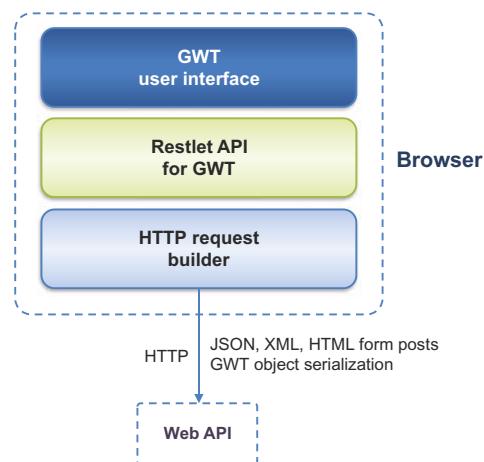


Figure 9.5 How the Restlet edition for GWT fits in the overall GWT architecture

This approach provides a flexible way to use Restlet on the GWT client side and to let RIA communicate with web APIs. As previously mentioned, Java isn't required on the server side. Let's see how to concretely use Restlet Framework edition for GWT with such architecture.

ARCHITECTURE FLEXIBILITY

GWT comes with a default RPC-style mechanism to interact with the server-side part of applications. As previously mentioned in section 9.1.3, GWT integrates a proprietary GWT-RPC protocol which ties the server side to GWT and the Servlet API. Although this approach can work for applications using GWT technology exclusively, it isn't flexible in the context of REST. A RESTful web API should be accessible from any kind of HTTP client, and not be tied only to GWT.

The Restlet edition for GWT brings this flexibility within your GWT-based applications. It allows using an open range of formats for exchanging content, such as JSON and XML (and any other content type that you can parse and format from a string). Moreover you aren't tied to a particular technology on the server side (not even Restlet). Figure 9.6 summarizes this.

Restlet isn't required on the server side, but it offers comprehensive server-side support, including a convenient solution to cross-domain calls in an AJAX environment with the `Redirector` class. We discuss this feature in section 9.3.3. Moreover Restlet support doesn't prevent you from using GWT-RPC if necessary (or more likely, if already in place), as both mechanisms can work in parallel. Figure 9.7 shows the full flexibility of using GWT with Restlet.

Now that we've set the scene for the GWT client-side support of Restlet, it's time to see how to use it in practice when calling a REST resource from a GWT client.

9.2.3 Communicating with a REST API

The central feature of the GWT edition of Restlet consists of communicating with REST resources from the client side of GWT applications. As introduced in the previous section, Restlet for GWT provides an API that's close to other Restlet editions. It also takes advantage of GWT features like deferred binding to make Restlet easier to use and remove the need for representation formatting and parsing.

In this section we implement a web mail client application based on the example code available in the GWT distribution at <http://gwt.google.com/samples/Mail/> that

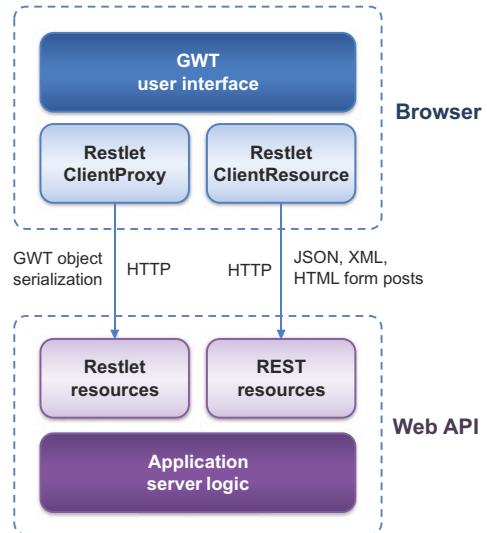


Figure 9.6 All the aspects shown here display the flexibility of the Restlet edition of GWT, which is one of its key features.

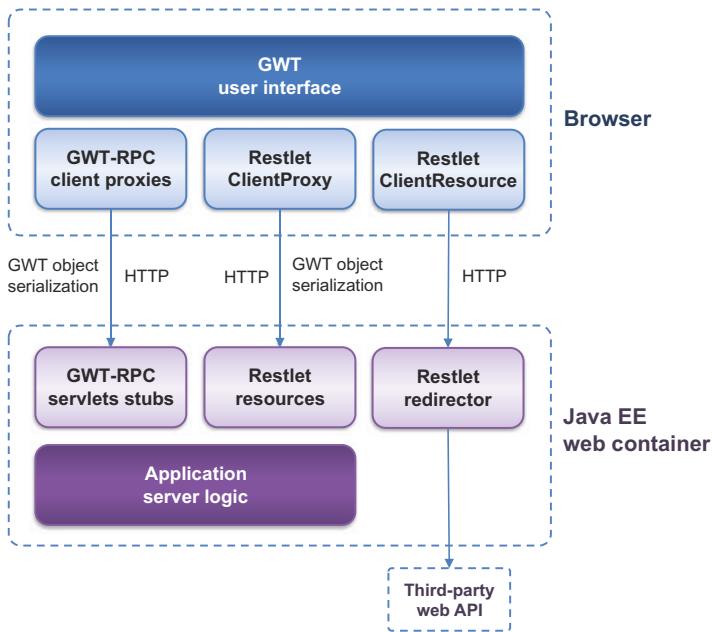


Figure 9.7 Full flexibility using GWT-RPC and Restlet for GWT

provides a rich mail client UI, but without true persistence. You'll connect this GUI to the mail application developed in previous chapters using the Restlet Framework edition for GWT.

USING CLIENT-SIDE RESTLET API

The nature of the underlying AJAX mechanism limits the Restlet API available in the GWT edition. Some standard classes, like `java.net.URI`, aren't available. The use of asynchronous calls is required and there is no support for reflection. Calling REST resources located in other domains also requires some workarounds.

Base package of the GWT edition for Restlet

In the Restlet edition for GWT, the base package is `org.restlet.client` instead of the usual `org.restlet`. The reason is that a GWT application contains both client and server sides in the same web application, and having different packages for each side greatly facilitates their coexistence.

This edition corresponds to a GWT module and needs to conform to GWT naming conventions, as described in section 9.1.1. In this context, client-side content is located in a client subpackage by default.

The Restlet edition for GWT makes it possible to use extensions within GWT applications. The list of available extensions is much more restricted than in other editions due to the underlying browser capacities. Table 9.1 lists all compatible Restlet modules available.

Table 9.1 Modules usable with GWT

Module	Description
org.restlet.Restlet	Port of client-side of the Restlet API
org.restlet.JSON	Port of JSON representation support
org.restlet.XML	Port of XML representation support

As with every GWT module, you need to specify the required dependencies within the module descriptor file. The GWT edition of Restlet integrates such a descriptor in each JAR file. For this example, you use the core module as well as the JSON and XML extensions. In the GWT mail client application, the descriptor called `Mail.gwt.xml` and located in the `org.restlet.example.book.restlet.ch09` package contains these lines:

```
<module>
  ...
  <inherits name="org.restlet.Restlet"/>
  <inherits name="org.restlet.JSON"/>
  <inherits name="org.restlet.XML"/>
  ...
</module>
```

Unsurprisingly the central class of Restlet client support for GWT is `ClientResource`. It has a similar contract to what we've seen, except that some methods have been removed due to GWT limitations for client side, as summarized in table 9.2.

Table 9.2 ClientResource methods removed for the GWT edition

Method kind	Description
Static create methods	Take a <code>Class</code> parameter and are built on the <code>wrap</code> method, which uses Java reflection.
<code>ClientResource</code> constructor using <code>URI</code> class	The <code>URI</code> class isn't available in client-side GWT support.
<code>handle</code> methods accepting <code>Class</code> parameter	Indirectly use Java reflection.
Methods to execute HTTP methods and accepting <code>Class</code> parameter	These methods (<code>get</code> , <code>post</code> , <code>put</code> , <code>delete</code> , and <code>options</code>) accept a <code>Class</code> parameter and are built on the previous <code>handle</code> methods.
Methods <code>getChild</code> , <code>getParent</code> and <code>wrap</code>	Some signatures of these methods accept a <code>Class</code> parameter and use Java reflection.

The major change when using the GWT edition of `ClientResource` is that only asynchronous response handling is supported. For this reason, responses must be retrieved using callbacks, and calling methods always return immediately. Executing a call without specifying a callback throws an exception.

Let's now enhance the GWT-based webmail application to add data persistence using Restlet. To remain concise, you'll focus only on the contacts list, corresponding to code in the `Contacts` class initially in the `com.google.gwt.sample.mail.client` package and moved into `org.restlet.example.book.restlet.ch09.client`.

We first remove the inner class `Contact` from the original sample code and the `contacts` property. Instead, we use the `ContactRepresentation` class provided by the Restlet mail application in the `common` package as defined in the previous chapter for GAE.

Listing 9.2 describes how to use the `ClientResource` class to make HTTP calls from client-side GWT code and retrieve the list of contacts for an artificially fixed Homer user. The way to initialize the instance and execute the request is different from the previous use you've seen, as you need to make asynchronous calls.

Listing 9.2 Using the `ClientResource` class within GWT client side

```
ClientResource clientResource =
    new ClientResource("/accounts/chunkylover53/contacts/");
clientResource.setOnResponse(new Uniform() {
    public void handle(Request request, Response response) { ←
        try {
            JsonRepresentation representation
                = new JsonRepresentation(response.getEntity());
            JSONArray jsonContacts = (JSONArray) representation.getJsonObject();
            (...)

        } catch (Exception ex) {
            GWT.log("Unable to parse JSON", ex);
        }
    }
});
clientResource.get(MediaType.APPLICATION_JSON); ←
```

After creating the `ClientResource` and specifying the target resource URI, set the callback to handle the response using the `setOnResponse` method and the `Uniform` callback interface. The latter has a `handle` method ① that is only called when the response has been received from the server. Now that the callback is set, you can execute the request using the `get` method ②, which immediately returns (`Void` result parameter). Specify the desired representation format (JSON in the listing) with the first method parameter. Now let's concentrate on using GWT support for JSON and XML with the corresponding Restlet extensions.

USING JSON AND XML REPRESENTATIONS

Using JSON and XML technologies within GWT is a bit specific because they correspond to the port of JavaScript ones into Java. They're provided through modules `com.google.gwt.json.JSON` and `com.google.gwt.xml.XML`.

For JSON, the central class is `JSONParser`, which parses text and returns a set of JSON objects. But the use of the JSON parser is hidden within the `JsonRepresentation` class of Restlet. The root type for JSON element corresponds to `JsonValue`, which has

subclasses for specific types available in the com.google.gwt.json.client package, as listed in table 9.3.

Table 9.3 Classes corresponding to supported JSON types

JSON type	Description
JSONArray	Array of JSON values
JSONObject	Boolean value
JSONNull	Null value
JSONNumber	Numeric value
JSONObject	JSON object with string-keyed property values
JSONString	String value

The next listing describes how to use these classes to browse the contacts array present in the received JSON content. According to the expected type of element, you can cast it to the corresponding one listed in table 9.3.

Listing 9.3 Parsing JSON content received through the representation

```
JsonRepresentation representation
    = new JsonRepresentation(response.getEntity());
JSONArray jsonContacts
    = (JSONArray) representation.getValue();
for (int i = 0; i < jsonContacts.size(); i++) {
    JSONObject jsonContact = (JSONObject) jsonContacts.get(i);
    ContactRepresentation contact = new ContactRepresentation();
    contact.setFirstName(((JSONString) jsonContact.get("firstName"))
.stringValue());
    contact.setLastName(((JSONString) jsonContact.get("lastName"))
.stringValue());
    contact.setEmail(((JSONString) jsonContact.get("email"))
.stringValue());
    contact.setLogin(((JSONString) jsonContact.get("login"))
.stringValue());
    contact.setSenderName(((JSONString) jsonContact.get("senderName"))
.stringValue());
}
```

The diagram illustrates the process of parsing JSON content. It starts with a call to `representation.getValue()`, which returns a `JSONArray`. This is annotated with a callout 1: "Get JSON array of contacts". The next step is to iterate over the array using a `for` loop, annotated with a callout 2: "Iterate over JSON array". Finally, for each contact in the array, a `ContactRepresentation` object is created and its properties are set from the corresponding JSON object, annotated with a callout 3: "Build contact object from JSON".

You first get the JSON contact array directly from the representation using its `getValue` method ①, noting that since version 2.1 there is now a `getJSONArray()` method. Then iterate the array and get a `JSONObject` instance representing a contact for each contained element ②. You're now able to build a `Contact` instance from the properties of this `JSONObject` ③. You can then use the `Contact` instance to update the contact part of the UI (not shown).

Next we'll implement the same processing in XML. In this case the representation gives access to a `Document` instance representing the XML document. From the latter

you have access to the root document and its children. Using the `getFirstChild` and `getElementsByTagName` methods, you can browse the whole XML tree of information, making it possible to build a list of contacts that will be used to update the UI. The following listing describes how to use the GWT XML API to handle the received XML content.

Listing 9.4 Parsing XML content received through the representation

```

DomRepresentation representation
    = new DomRepresentation(response.getEntity());
Document document = representation.getDocument();
Element listElement = (Element)document.getFirstChild();
NodeList nodes = listElement.getElementsByTagName(
    "org.restlet.example.book.restlet.ch09.common.ContactRepresentation");
for (int i = 0; i < nodes.getLength(); i++) {
    Element contactElement
        = (com.google.gwt.xml.client.Element)nodes.item(i);
    ContactRepresentation contact = new ContactRepresentation();
    Element contactFirstNameElement
        = (Element)contactElement.getElementsByTagName("firstName").item(0);
    contact.setFirstName(
        contactFirstNameElement.getFirstChild().getNodeValue());
    Element contactLastNameElement
        = (Element)contactElement.getElementsByTagName("lastName").item(0);
    contact.setLastName(
        contactLastNameElement.getFirstChild().getNodeValue());
    Element contactEmailElement
        = (Element)contactElement.getElementsByTagName("email").item(0);
    contact.setEmail(contactEmailElement.getFirstChild().getNodeValue());
    Element contactLoginElement
        = (Element)contactElement.getElementsByTagName("login").item(0);
    contact.setLogin(contactLoginElement.getFirstChild().getNodeValue());
    Element contactSenderNameElement
        = (Element)contactElement.getElementsByTagName("senderName").item(0);
    contact.setSenderName(
        contactSenderNameElement.getFirstChild().getNodeValue());
}

```



Build contact object from XML

① Get XML document

② Browse XML document

③ Build contact object from XML

You first get the XML document directly from the representation using its `getDocument` method ①. You then browse the XML tree and get an `Element` instance representing a contact for each contained element ②. You can now instantiate a `ContactRepresentation` instance and fill it with the previous element's attributes ③.

As you can see, directly using JSON and XML formats within GWT applications can be painful and results in a lot of code. In the next section we describe an automatic serialization mechanism that version 2.0 of Restlet Framework introduced for GWT.

AUTOMATIC OBJECT SERIALIZATION SUPPORT

As you saw in the previous section, writing the formatting and parsing code for exchanged data is time-consuming. The GWT edition of Restlet provides a convenient way to automatically generate this serialization code.

This edition goes further than the simple port of the `ClientResource` class and relies on a powerful feature of GWT called *deferred binding*. Using this feature, extra code can be automatically generated during the compilation phase. Restlet uses it to generate automatic serialization code.

With this approach, you don't need to manually provide implementations of the Restlet annotated interface; you can directly create instances with the `GWT.create` static method. The deferred binding is activated during the compilation phase if you extend the `org.restlet.client.resource.ClientProxy` interface in your annotated Restlet interfaces:

```
<module>
  ...
  <generate-with class="org.restlet.rebind.ClientProxyGenerator">
    <when-type-assignable class="org.restlet.client.resource.ClientProxy"/>
  </generate-with>
</module>
```

Nothing else has to be done in the module file of your GWT application except specifying the `org.restlet.Restlet` module as in the previous section. The deferred binding configuration is automatically inherited.

Before we describe how to call the REST resource with this feature, you need to declare a `ClientProxy` extension dedicated to this resource. This entity will act as a proxy in front of the resource and handle all the plumbing required to communicate with it through REST. The power of this approach is that you can use common Restlet annotations to associate HTTP methods with Java methods. When a method of this interface is invoked, a call to the related HTTP method is transparently executed.

GWT and Restlet versions

Restlet Framework version 2.0 supports version 2.0 and 2.1 of Google Web Toolkit, but not later, due to API breaking changes introduced in GWT 2.2. Restlet Framework version 2.1 supports GWT 2.3 and above, but not previous versions.

In addition, you have to add a parameter of type `Result<?>` to each method corresponding to a remote call in order to asynchronously handle the response. The following snippet describes how to declare a dedicated `ClientProxy` extension for the contacts resource in order to remotely manage it through REST:

```
public interface ContactsResourceProxy extends ClientProxy {
  @Get
  public void retrieve(Result<ArrayList<ContactRepresentation>> result);
}
```

1 Extend ClientProxy

2 HTTP GET method

```

    @Post
    public void add(ContactRepresentation contact, Result<Void> result);
}

```

HTTP POST method ③



ContactsResourceProxy extends the ClientProxy interface ①, indicating that it acts as a client proxy for the remote Contacts resource. It adds a method to retrieve contacts and associates it with the corresponding HTTP method ② ③ to use when executing them. The same Restlet annotations are used as in other editions, for example @Get for the GET HTTP method, except that they come from the org.restlet.client.resource package.

In the end, a call to retrieve turns into an HTTP GET call. The method has one parameter of type Result to handle the response. This type is parameterized by the expected type for response. The magic here is that Restlet will internally manage the conversion to and from representations, relying on GWT-specific object serialization format.

Comparison between GWT and regular Java object serializations

In regular Java SE, there are two ways to automatically serialize Java objects. The first uses a special binary format via the `java.io.ObjectInputStream` and `ObjectOutputStream` classes, and the second uses an XML format via the `java.beans.XMLEncoder` and `XMLDecoder` classes.

Restlet edition for Java SE supports those mechanisms by default when handling media types corresponding to the `MediaType.APPLICATION_JAVA_OBJECT` and `APPLICATION_JAVA_OBJECT_XML` constants.

GWT object serialization is another compact format optimized for size but comparable to native Java serialization. It's specific to the GWT technology and was introduced for the GWT-RPC mechanism. Restlet only reuses the object format and not the rest of the GWT-RPC plumbing, associating it to the `APPLICATION_JAVA_OBJECT_GWT` constant.

Note that each class that needs to be serialized by GWT needs to implement the `Serializable` interface and not use collections interfaces by concrete classes such as `ArrayList` instead of `List`.

The GWT AsyncCallback interface is also supported as the last parameter of each Restlet-annotated method corresponding to a remote call. This interface from GWT-RPC defines a similar contract to Restlet's Result class and can be useful to reduce code changes when converting GWT-RPC calls into RESTful ones. If you care more about portability of the client side, then you should use the `Result` class, which is also supported in non-GWT editions to make asynchronous calls.

Although GWT-RPC requires you to write special server-side code, you don't need to change anything on the server side for Restlet resources besides adding the `org.restlet.ext.gwt` extension. It's now time to illustrate this approach by calling REST resources from GWT client code. The code shown in listing 9.5 retrieves the list of contacts.

Supported data format for Restlet deferred binding support

At the time of writing, the deferred binding support of Restlet only supports the special GWT object serialization format. In future versions, XML and JSON-based formats could be added by leveraging the Piriti library. This library provides JSON and XML mappers for GWT based on annotations and deferred binding. Further details are available on Google Code at <http://code.google.com/p/piriti/>.

Listing 9.5 Using deferred binding support of Restlet to execute REST call

```
ContactsResourceProxy contactsResource =           ← ① Create client proxy
    GWT.create(ContactsResourceProxy.class);
contactsResource.getClientResource().setReference(
    "/accounts/chunkylover53/contacts/");
contactResource.retrieve(new Result<ArrayList<ContactRepresentation>>() {           ←
    public void onSuccess (ArrayList<ContactRepresentation> result) {                   ←
        for (ContactRepresentation contact : result) {
            addContact(contact);
        }
    }

    public void onFailure(Throwable caught) {           ←
        GWT.log("Unable to retrieve the contacts list");
    }
});
```

The diagram illustrates the execution flow of the code in Listing 9.5. It shows five numbered steps: 1. Create client proxy (GWT.create), 2. Set call properties (setReference), 3. Execute call (retrieve), 4. Handle successful response (onSuccess), and 5. Handle error response (onFailure). Arrows indicate the flow from step 1 to 2, 2 to 3, 3 to 4, and 3 to 5. A callout 'Handle successful response' points to step 4, and another callout 'Handle error response' points to step 5.

Handle
successful
response

The first step consists of creating the client proxy for the contacts resource with `GWT.create` ①. Because the `ContactsResourceProxy` extends the `ClientProxy` interface, you have access to the underlying `ClientResource` instance through the `getClientResource` method to set properties of the call, like the target address, supported media types, or authentication info ②. The call to the `retrieve` method ③ executes the HTTP request. Handling the corresponding response ④ ⑤ is done asynchronously using the `Result` callback interface in its `onSuccess` method (if everything goes well).

Now that every part of the execution chain is implemented, you can test it to get contacts from a RESTful application and display them in the UI. Let's start the application within Eclipse using `Run As > Web Application`. This starts hosted mode; access it in a browser using the address `http://127.0.0.1:8888/Mail.html?gwt.codesvr=127.0.0.1:9997`.

When the Contacts tab in the menu on the left is displayed, you should see all the contacts returned by the RESTful mail server, as illustrated in figure 9.8.

Before describing the server-side GWT extension, let's see how to handle cross-domain requests on HTTP resources from the client side.

9.2.4 Handling cross-domain requests on the client side

A common issue with web browsers is that they block resources located in domains different from the one that served the resource making the request. This security policy is known as the Same Origin Policy (SOP) and comes from restrictions on using the

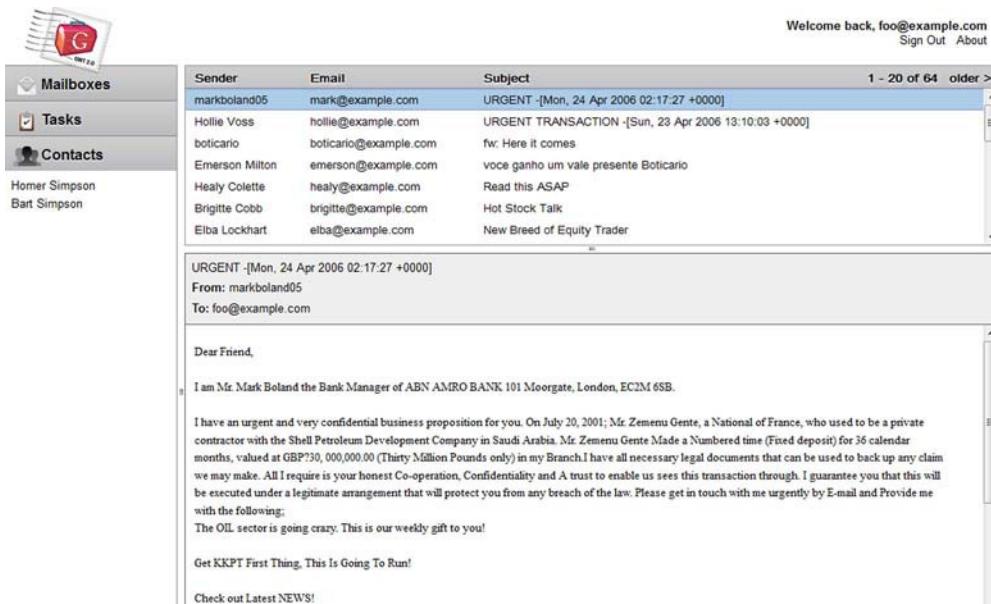


Figure 9.8 GWT mail client accessing the RESTful web API

XMLHttpRequest object in JavaScript. Restlet provides an elegant solution on the server side, as we describe in section 9.3.3.

Past solutions to cross-domain limitations used workarounds, such as dynamic script element generation, which aren't convenient because they don't provide a consistent solution across browsers and don't support all representation media types for data. The common JSONP approach, natively supported by GWT with its JsonpRequestBuilder class, is shown in the following listing.

Listing 9.6 Using jsonp within GWT

```
JsonpRequestBuilder jsonp = new JsonpRequestBuilder();
jsonp.requestObject("http://otherdomain.com/mypath", new
    AsyncCallback<MyData>() {
    public void onFailure(Throwable throwable) {
        ....
    }
    public void onSuccess (MyData data) {
        ....
    }
});
```

The problem here is that the returned content contains not only data but also the name of the JavaScript callback, as described in following snippet. For this reason, this approach is more of a hack than an ideal solution because it doesn't truly follow the REST principles:

```
<callback>(<json data>)
```

In addition to these workarounds, the W3C has proposed the new Cross-Origin Resource Sharing (CORS) working draft, which provides a way for web servers to support cross-site access controls based on response headers. Such an approach is built on the Access-Control-Allow-Origin header, which is not yet supported by all browsers.

The browser uses the content of the header to determine whether the resource can be accessed by any domain in a cross-site manner. Specifying * defines that all domains can access the resource whereas a specific one or a list restricts this access. Listing 9.7 describes how to use the Access-Control-Allow-Origin header in the response to an HTTP resource call using Restlet. Even though it requires changes on the server-side to set this header, the calls will effectively be made directly by the client.

Listing 9.7 Using cross-domain header in a Restlet response

```
@Get  
public Representation getContent() {  
    (...)  
    Series<Header> responseHeaders = (Series<Header>)  
        getResponseAttributes().get(HeaderConstants.ATTRIBUTE_HEADERS);  
  
    if (responseHeaders == null) {  
        responseHeaders = new Series(Header.class);  
        getResponseAttributes().put(HeaderConstants.ATTRIBUTE_HEADERS,  
            responseHeaders);  
    }  
  
    responseHeaders.add(new Header("Access-Control-Allow-Origin", "*")); ←  
    (...) Set cross-domain header  
}
```

You've now completed your tour of the client-side edition of Restlet for GWT. It provides a way to access REST resources from GWT applications that mimics the way Restlet works in Java SE and EE applications. This edition also uses GWT's deferred binding to allow transparent calls to REST resources with a minimum amount of manual code. So far we've only discussed the client side of GWT applications, but Restlet also supports GWT on the server side.

9.3 Server-side GWT extension

In addition to the edition for GWT on the client side, the Restlet Framework provides a dedicated server-side extension called `org.restlet.ext.gwt`, available in the following editions: Java SE, Java EE, Google App Engine, and OSGi. This extension is light because it only converts between GWT binary representations and Java objects, based on a dedicated media type, without implementing additional GWT-RPC plumbing. The ConverterService uses this extension to convert Java objects in GWT object serialization format for request with the corresponding media type. (We detail the conversion mechanism in section 4.5.4.)

These entities are directly integrated with the Restlet engine when the extension is present in the classpath; no explicit configuration is necessary.

Two issues regarding the server side need to be detailed here: How to make GWT-RPC work with the Restlet support for GWT, and how to enable cross-domain requests from the server side.

9.3.1 Working along with GWT-RPC

As described in section 9.2.2, Restlet edition for GWT retains the ability to use GWT-RPC in parallel to a RESTful web API.

Imagine that you want to have a remote contact service on a server that you can call using default remoting mechanisms of GWT. By default, implementing such services requires you to implement the remote interface by extending the GWT RemoteServiceServlet Servlet with service methods. These GWT remote services are typically Servlets, as described in following snippet:

```
public class ContactsServiceImpl
    extends RemoteServiceServlet
    implements ContactsService {
    public List<Contact> getContacts() {
        (...)
```

The ContactsServiceImpl class is then configured within the web.xml file as a Servlet. Now that this first part is done, let's configure the REST resources.

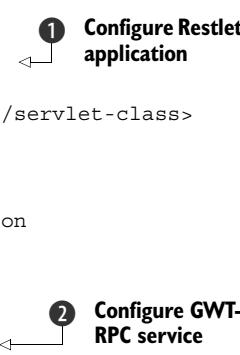
There are several ways to configure such resources on the server side with Restlet. For this demonstration, you'll configure the Restlet application using the Servlet extension and its ServerServlet class. The following listing describes how to configure both approaches in the web.xml file of a web application, the Restlet application, and the GWT-RPC remote service.

Listing 9.8 Configure both GWT-RPC and Restlet support

```
<web-app>
  (...)

  <servlet>
    <servlet-name>Restlet Contact resource</servlet-name>
    <servlet-class>org.restlet.ext.servlet.ServerServlet</servlet-class>
    <init-param>
      <param-name>org.restlet.application</param-name>
      <param-value>
        org.restlet.example.book.restlet.ContactApplication
      </param-value>
    </init-param>
  </servlet>

  <servlet>
    <servlet-name>RPC Contact service</servlet-name>
    <servlet-class>
      org.restlet.example.book.restlet.ContactsServiceImpl
```



```

</servlet-class>
</servlet>

( ... )
</web-app>

```

As you can see, you configured two Servlets in the web.xml file, corresponding to the two approaches. First the ServerServlet Servlet ① of Restlet configures the entry point to the Restlet application called org.restlet.example.book.restlet.ch09.resource. ContactApplication using the org.restlet.application initialization parameter. The default GWT-RPC approach is then configured ② using an implementation of RemoteServiceServlet called ContactsServiceImpl.

Configuring the REST approach along with RPC-based approach is based on Servlet configurations. Before ending our discussion of server-side support for GWT, we'll explain a convenient and elegant way to issue cross-domain calls without the previous limitations.

9.3.2 Handling cross-domain requests on the server side

Although there are some solutions on the client side to make cross-domain calls on HTTP resources, they aren't completely standard or they depend on proper browser support. You can implement an alternative approach on the server side. In this case you need to have an additional component acting as a reverse proxy for the target HTTP resources. This approach can add latency because two requests are now executed in sequence.

Restlet provides the org.restlet.routing.Redirector class to implement a reverse proxy, as shown in figure 9.9. This class is much more generic than this GWT use case and therefore isn't part of the GWT extension for this reason. We describe it here, though, because it provides a convenient solution to this problem. For further details, refer to section 8.3.

The redirector is typically configured in the createInboundRoot method of the application, as described in the following snippet:

```

public Restlet createInboundRoot() {
    Router router = new Router(getContext());
    String target = "http://www.restlet.org{rr}";
    Redirector redirector = new Redirector(
        getContext(), target,

```

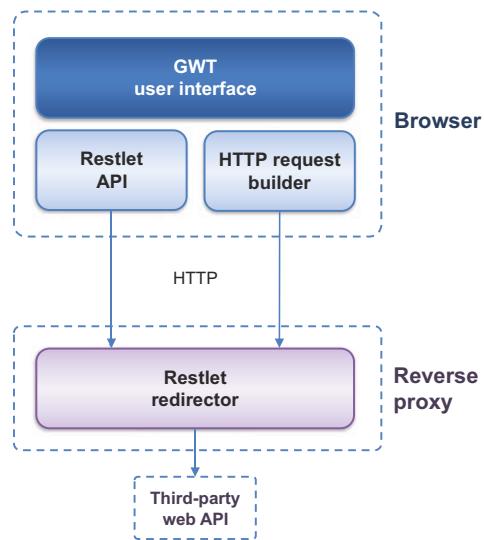


Figure 9.9 Implementing reverse proxy using the Redirector class

```

        Redirector.MODE_SERVER_OUTBOUND) ;
    router.attach("/search", redirector);
}

```

In the preceding snippet the first step consists of instantiating the `Redirector` class. The use of this class is described in section 7.3.

Notice the `{rr}` pattern when attaching the `redirector` on the `router`. This allows appending to the target URI all elements present at the end of the original URI. Without it, all requests would be redirected to only one. In the end, requests to `www.myexample.com/search/foo/bar` will be served as if they were made to `www.restlet.org/foo/bar`, but without the cross-domain limitations.

This section ends the coverage of the GWT edition of the framework. Restlet integrates all internal issues and also takes advantage of GWT's deferred binding to simplify use further. Based on the design experience of this edition, asynchronous call support was added to the Restlet API in other editions. Its use is almost identical, involving both `ClientProxy` and `Result` interfaces. We now turn to another thin client for RESTful applications: Android. We'll describe how to use Restlet on Android-based mobile phones with the Restlet edition for Android.

9.4 Understanding Android

With the commodification of smartphones started by the Palm Treo and pushed further by the Apple iPhone, more and more mobile users have a usable access to the web from their phones. So far developers have been stuck with proprietary platforms that lacked the productivity and portability common in the Java world. Then came Android!

Android is an open source mobile OS initiated and driven by Google and the Open Handset Alliance, including prestigious manufacturers. Needless to say, Android has gained a lot of traction recently.

Technically speaking, Android is built on a customized Linux kernel, libraries such as WebKit for the web browser, and an extensible Application Framework developed in the Java language but running on a special Dalvik virtual machine (see a complete overview in figure 9.10). All the built-in applications are written in Java and can be accessed or customized via the Android API.

Before describing how to use REST with Android, we'll give a few more details about the Android technology itself. We'll then describe how Restlet can help and how you can use it within Android applications to access RESTful web APIs.

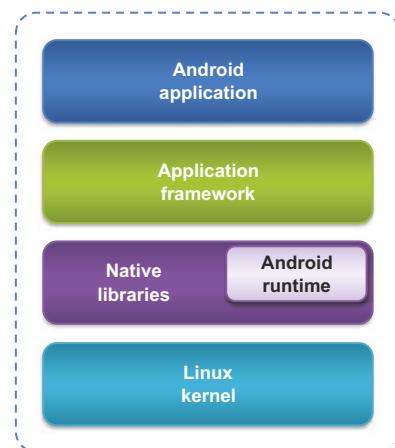


Figure 9.10 Layers of the Android platform

9.4.1 *Android overview*

Android is a complete software stack targeting mobile devices that includes an OS, middleware, and key applications. It provides a complete foundation to build native mobile applications. For this purpose, the Android SDK includes tools and APIs written for the Java programming language. Figure 9.10 shows a high-level description of the Android platform.

Table 9.4 lists more details on each part of this platform.

Table 9.4 Different parts of the Android platform

Part name	Description
Applications	Android comes with a set of core applications for common use cases. These applications correspond to email and SMS clients, calendar and contact managers, maps, browsers, and so on. All these are built on the application framework and are written using the Java programming language.
Application framework	This framework is powerful and truly open because it gives access to the same Java APIs as core applications and is used in a component-based approach. Reusing components is simple, and any application can publish its features and consume other ones. The framework also makes it possible to override or replace components.
Libraries	Android includes a set of C/C++ libraries used by components of the Android platform, such as System C library, media, and 2D/3D libraries. You don't need to know more about these libraries right now—only that their features are exposed to developers through the previously described application framework.
Android runtime	Android internally uses a special virtual machine (VM) called Dalvik , which has been optimized to run on mobile devices and to run multiple VMs efficiently. This virtual machine is built on the Linux kernel for underlying functionality such as threading and low-level memory management. This tool includes a set of core libraries that provides most of the functionality available in the core libraries of the Java programming language.
Linux Kernel	Android uses Linux version 2.6 as the OS for core system services, memory and process management, network stack, and driver model. The kernel can be seen as an abstraction layer between the hardware and the rest of the software stack above it.

Because a full description of Android is beyond the scope of this book, we won't provide details here; we recommend *Android in Action*, 3rd Edition, by W. Frank Ableson, Robi Sen, Chris King, and C. Enrique Ortiz (Manning, 2011), for more information.

9.4.2 *Installing Android and Eclipse plug-ins*

Developing Android applications requires you to install the Android SDK, available at <http://developer.android.com/sdk/>. An emulator called the Android Virtual Device (AVD) comes with the SDK and simulates real Android-based mobile phones. After downloading the SDK, you should run the Android SDK Manager and install the suggested packages, including the Android SDK Platform tools, as shown in figure 9.11.

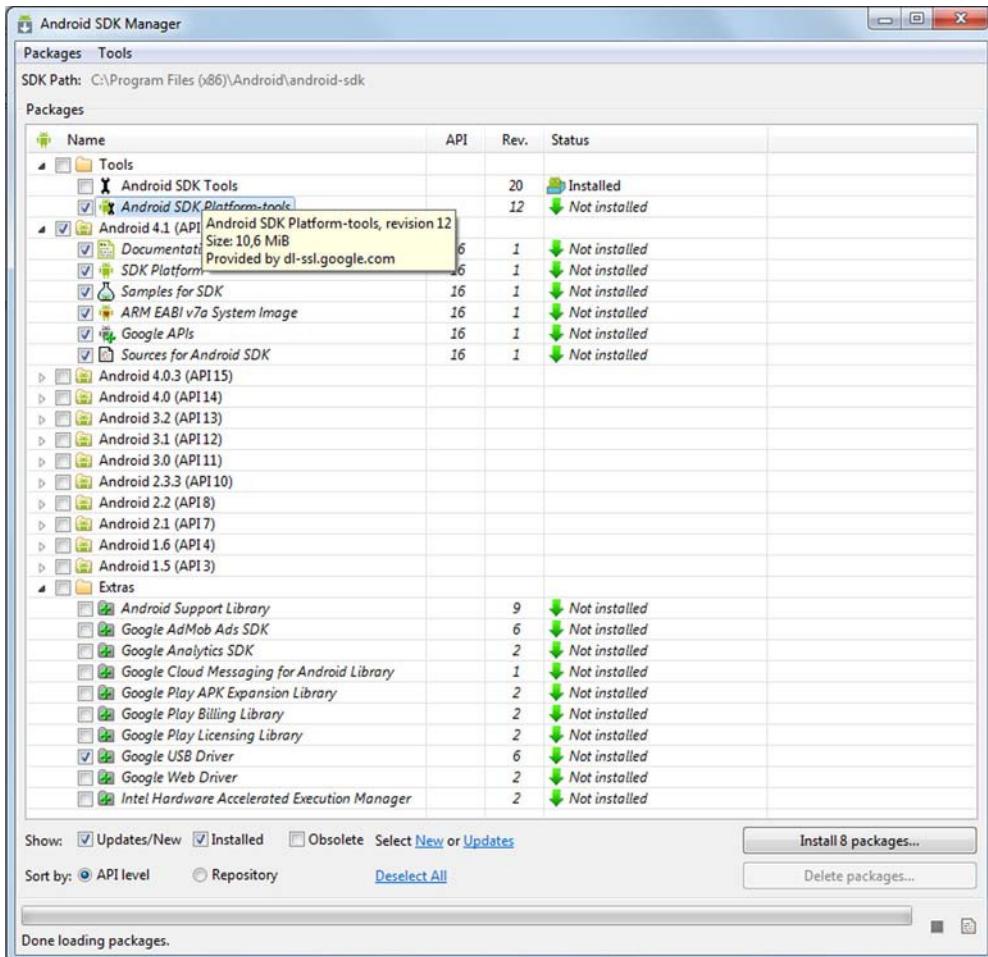


Figure 9.11 Android SDK Manager

Increasing memory allocated to Eclipse IDE

As you install plug-ins for GWT, GAE, and Android, the memory allocated by default to your Eclipse environment might become too limited and produce strange errors when you try to launch an Android application in the device emulator.

To raise this limit, edit your `eclipse.ini` file and set a value of `512m` or higher for the `launcher.XXMaxPermSize` property, and `512m` or higher for the `-Xmx` argument.

Next, you'll use the AVD Manager that manages the AVD, on which you'll execute Android applications developed with Eclipse. Click the New button, provide a name for the AVD, and leave default values in properties. When you're done, click the Create AVD button, and the new AVD will appear in the list, as shown in figure 9.12.

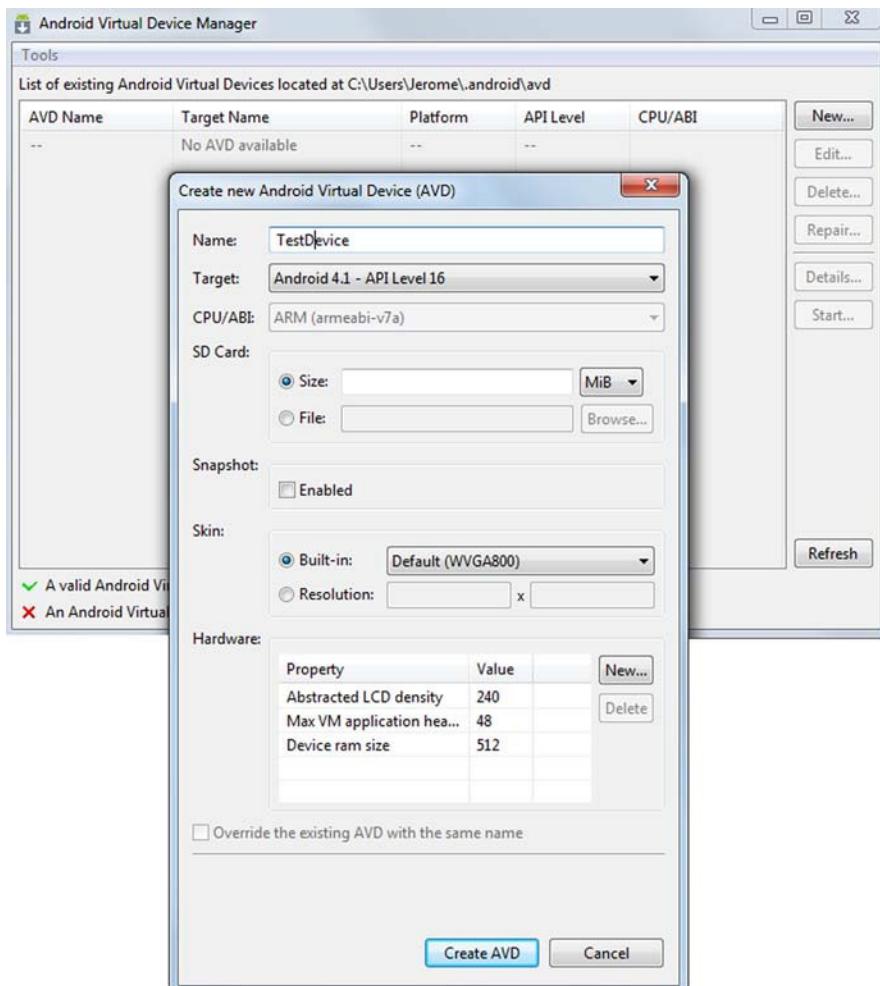


Figure 9.12 Android Virtual Device Manager

In addition to the basic SDK, a special integration with Eclipse, called the Android Development Tools (ADT) plug-in, provides the following features:

- Integration of external Android development tools within Eclipse
- Wizards to create and configure artifacts necessary for Android applications
- Facilities to efficiently build Android applications and generate signed APKs (Android packages similar to JAR files), which can be shipped to end users
- Dedicated editors for Android application files like layout files

To install this tool, use the Update Manager of Eclipse, available from the Help > Install New Software menu. You need to specify the Update Site for ADT within the corresponding window. The address of this site is <https://dl-ssl.google.com/android/eclipse/>. After

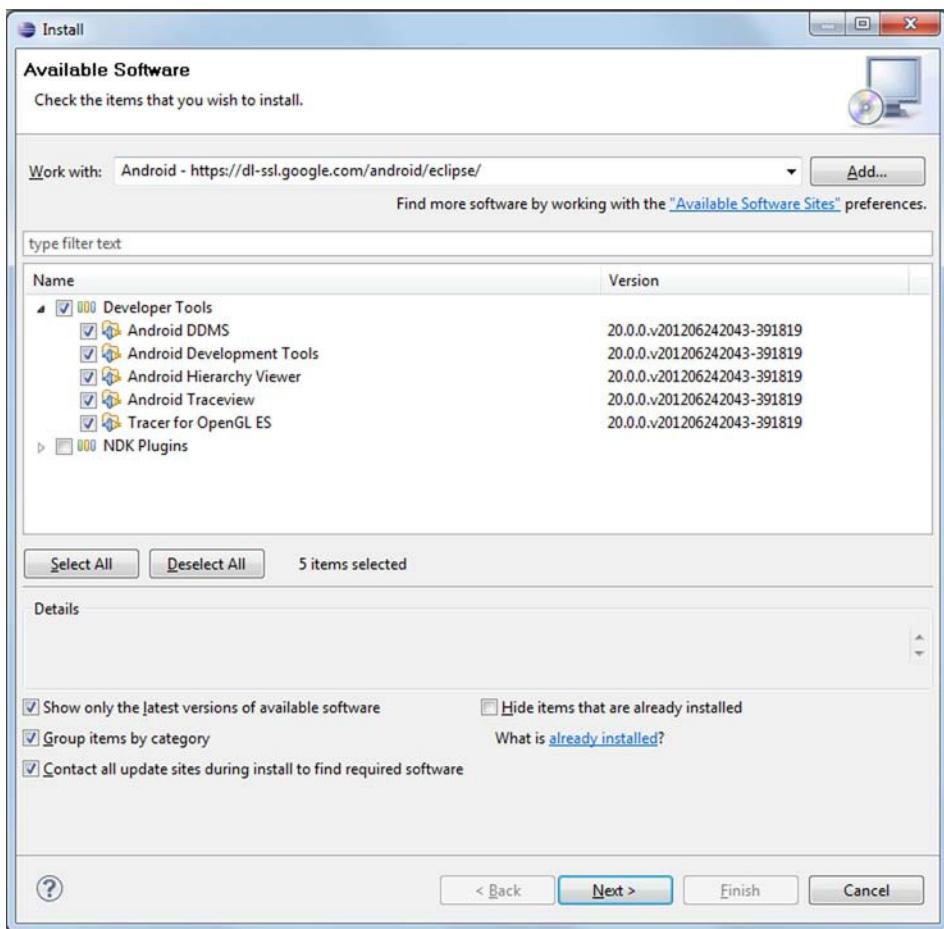


Figure 9.13 ADT modules to install through Eclipse's update manager tool

specifying it, you have access to the modules described in figure 9.13. In addition to Android development tools, select the following modules:

- *Dalvik Debug Monitor Server tool (DDMS)*—Provides a debugging tool for Android application development
- *Android Hierarchy view*—To debug and optimize your user interfaces
- *Android TraceView*—Provides a graphical viewer for execution logs saved by your application

Let's select all of them and start their installation. After restarting Eclipse, ADT is available within your Eclipse instance.

Now that you have everything properly installed for developing Android applications with Eclipse, let's come back to our subject and tackle how to use REST within Android applications. Because Android targets mobile devices, corresponding applications are

commonly connected to the internet and can access its resources. As you're about to see, the Restlet Framework can provide a useful help to access remote web APIs from Android applications.

9.5 The Restlet edition for Android

As with GWT, Android provides a particular execution environment based on the Java language but with differences in terms of available runtime APIs. Differences between the Dalvik VM and the regular Java VM correspond to issues such as garbage collection, memory footprint, and CPU use. The other differences are related to the runtime APIs available, with some packages added and others removed. For this reason Restlet provides a dedicated edition that's compatible with Dalvik and the APIs provided by Android.

You'll see throughout this section that all facilities provided by Restlet to consume web APIs are usable within Android-based devices. After describing the features of the edition for Android, we'll illustrate Restlet's use in Android native applications for both client-side and server-side support. The major benefit here is the ability to use the high-level API of Restlet instead of the low-level HTTP clients provided natively by Android.

9.5.1 Restlet port to Android

The Restlet edition for Android contains some adjustments to take into account, such as classes not supported by the Android runtime environment, but otherwise is similar to the Java SE edition.

With this edition, you can use Restlet on Android with both client-side and server-side HTTP connectors. Restlet extensions are being progressively ported; at the time of writing, all extensions listed in table 9.5 have been tested and work on the Android platform.

Table 9.5 Restlet extensions that can be used with Android

Extension	Description
org.restlet.ext.atom	Support for the Atom and AtomPub standards
org.restlet.ext.crypto	Support for cryptography including AWS and Azure
org.restlet.ext.html	Support for multipart HTML forms sending
org.restlet.ext.httpClient	Integration with Apache HTTP Client 4.1
org.restlet.ext.jaas	Support for JAAS authentication and authorization framework
org.restlet.ext.jackson	Integration with Jackson library for automatic JSON serialization
org.restlet.ext.json	Support for JSON representations
org.restlet.ext.net	Integration with Java URLConnection class

Table 9.5 Restlet extensions that can be used with Android (continued)

Extension	Description
org.restlet.ext.odata	Support for the OData web protocol (client side)
org.restlet.ext.rdf	Support for RDF parsing and formatting
org.restlet.ext.sip	Support for Session Initiation Protocol (SIP)
org.restlet.ext.xml	Support for XML and XSLT representations

Let's look at how to develop Android applications using Restlet support for client and server sides.

9.5.2 Client-side support

The client-side support in Android edition remains the same as the Java SE edition and is based on the `ClientResource` class. Three HTTP client connectors are currently available: the internal Restlet one, the Apache HTTP client already available on Android, and the `HttpURLConnection` class.

As the time of writing, the best choice for Android 2.2 (Froyo) and above is to use the `HttpURLConnection` class, which has the lightest footprint and is recommended by the Android team moving forward [11]. When the footprint isn't a concern, and internet connectivity is reliable, you can also consider the two other connectors that bring other benefits.

In all cases, you need to remember to add the `android.permission.INTERNET` Uses Permission in your `AndroidManifest.xml` file to ensure that your Android application is allowed by the end user to access the internet via the HTTP protocol.

Specific issues when using Restlet on Android

The internal HTTP client has been rewritten using the `java.nio` package. On some android devices, this may lead to encountering this kind of exception: `java.net.SocketException: Bad address family`. In this case, you can turn off the IPv6 preference as follows:

```
System.setProperty("java.net.preferIPv6Addresses", "false");
```

Also, contrary to other editions, the Android edition can't make use of Restlet's auto-discovery mechanism for connectors and converters provided as Restlet extensions. This is due to a limitation in the way Android repackages JAR files into APK files, leaving out the descriptor files in the `META-INF/services` folder used by the Restlet Framework for autodiscovery. The internal connector and converters are still automatically configured and can be used without additional configuration.

The workaround for extensions consists in manually registering those additional connectors and converters with the Restlet Engine. You can do this by adding to the

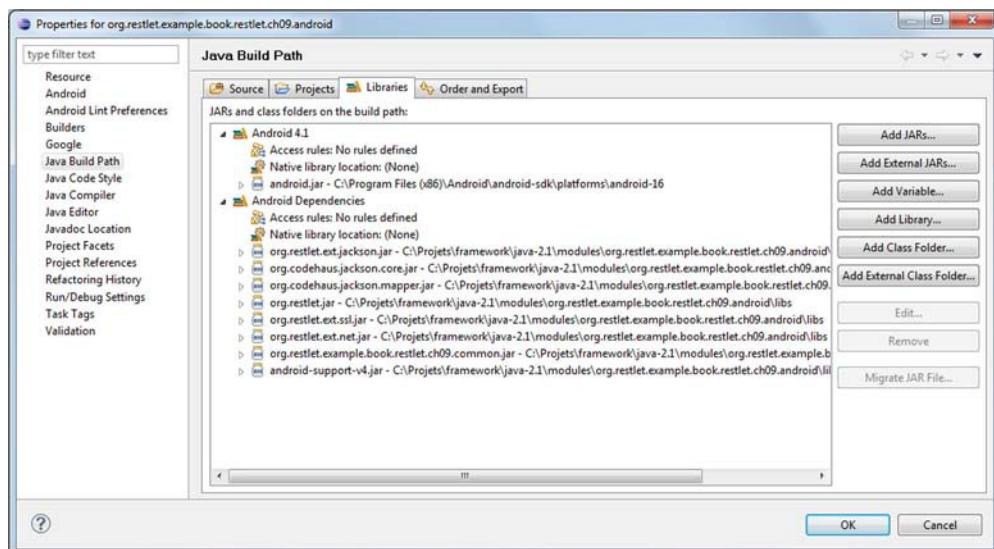


Figure 9.14 Configuring Restlet JARs for the Android project

list returned by the `getRegisteredConverters` and `getRegisteredClients` methods of the `org.restlet.engine.Engine` instance, as illustrated in listing 9.9.

Now that we're done with the Android-specific set-ups, we can focus on the application. We'll implement a simple Android application to display emails based on the GAE mail server that you used for the GWT client earlier.

You first create an Android Project with the corresponding New wizard of Eclipse ADT, check the activity creation, and specify `MobileMailClientMainActivity` for its name. Because you want to use the Restlet edition for Android, you must add the corresponding library JARs to the special `libs` folder, which adds them to the Java Build Path, as shown in figure 9.14.

Everything is now configured within the project. No time to waste—let's complete the autogenerated `MobileMailClientMainActivity` class to retrieve the list of emails from the web application in the back end at startup. For that, you complete the existing `onCreate` method. This class extends `ListActivity` because you want to display a list of emails, as shown in the next listing.

Listing 9.9 Implementation of the `SimpleWebMailActivity` class

```
public class MobileMailClientMainActivity extends ListActivity {

    @Override
    public void onCreate (Bundle savedInstanceState) {
        super.onCreate (savedInstanceState);
        setContentView(R.layout.activity_mobile_mail_client_main);

        ListView emailList = getListView();
        emailList.setTextFilterEnabled(true);
```

```

Engine.getInstance().getRegisteredClients().clear();
Engine.getInstance().getRegisteredClients()
    .add(new HttpClientHelper(new Client(Protocol.HTTP)));
Engine.getInstance().getRegisteredConverters()
    .add(new JacksonConverter());
AsyncTask<Void, Void, Void> task =
    new AsyncTask<Void, Void, Void>() {
        @Override
        protected Void doInBackground(Void... params) {
            ClientResource clientResource = new ClientResource(
                "http://reia-ch09.appspot.com/accounts/chunkylover53-mails/");
            MailsResource mailsResource = clientResource
                .wrap(MailsResource.class);
            MailsRepresentation emails = mailsResource.retrieve();
            final String[] subjects =
                new String[emails.getEmails().size()];
            for (int i = 0; i < emails.getEmails().size(); i++) {
                System.out.println(emails.getEmails().get(i));
                subjects[i] = emails.getEmails().get(i).getSubject();
            }
            runOnUiThread(new Runnable() { ← Display emails using UI thread
                public void run() {
                    setListAdapter(new ArrayAdapter<String>(
                        MobileMailClientMainActivity.this,
                        R.layout.activity_mobile_mail_client_main,
                        R.id.list_item, subjects));
                }
            });
            return null;
        }
    };
task.execute(null, null, null); ← Launch asynchronous task
}

@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater()
        .inflate(R.menu.activity_mobile_mail_client_main, menu);
    return true;
}
}

```

The code demonstrates the use of the Restlet Client API within an Android application. It starts by setting up the Restlet Engine with a specific HttpClientHelper. An AsyncTask is used to perform the HTTP request to a Google App Engine endpoint, which retrieves a list of emails. The retrieved emails are then processed to extract their subjects. Finally, the UI thread is updated using the runOnUiThread method to display the list of subjects.

The use of the `ClientResource` class within an Android activity is identical to regular Java applications, including the ability to use annotated interfaces. The main differences are in the way Android strongly suggests you run your HTTP calls using `AsyncTask` and only updating the UI using the `runOnUiThread` method.

Now, if you launch the Android client via the emulator, it should properly connect to the GAE back end accessible publicly and display the UI in figure 9.15.

This section discussed how to access RESTful applications using the client-side support in the Restlet edition for Android. There were additional steps to take in order to

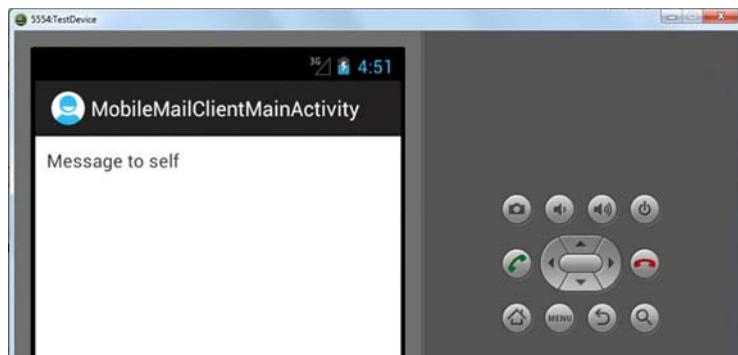


Figure 9.15 Simple mobile Android mail client

Using 127.0.0.1 and localhost for application testing

Be careful when using 127.0.0.1 or localhost within a URI. They correspond to the phone itself and not the machine when using Android emulator within Eclipse, for example. If you want to access RESTful applications running outside Android applications, you must necessarily use host names or IP addresses.

make Restlet work in this context, but it still provided higher-level support compared to the native Android platform, especially when you already use Restlet for your backend. Let's now describe Restlet's server-side support for Android.

9.5.3 Server-side support

In contrast to GWT, which compiles Java into JavaScript, Android applications are, like regular Java applications, compiled into byte code and executed on a Linux OS instead of a web browser. This aspect makes it possible for these applications to act as HTTP servers, not just clients.

Keep in mind that Android targets mobile devices that are less capable in terms of memory and CPU. Although Dalvik is similar to a JVM, it doesn't provide all the APIs present in standard JVMs. Because of these restrictions, it's not so simple to use classic server-side tools such as Java EE web containers. But Restlet edition for Android comes to the rescue.

As on the client side, you can use server-side support for Restlet in Android applications. This makes it possible to implement server applications, including HTTP servers. For that purpose, you can use all of Restlet's capabilities described throughout the book to implement RESTful applications on your mobile phone. The next listing describes how to start a standalone Restlet server within the `onCreate` method of an Android activity. This method is called when the activity is first created.

Listing 9.10 Starting a Restlet server within an Android activity

```
public void onCreate (Bundle savedInstanceState) {  
    super.onCreate (savedInstanceState);
```

```

Component component = new Component();
component.getServers().add(Protocol.HTTP, 8182);
component.getDefaultHost().attachDefault(new MailApplication());
try {
    component.start();
} catch (Exception e) { (...) }
}

```

Create,
initialize Restlet
component

Start
component

An Android port of the well-known web container Jetty is also provided by the i-Jetty project. Although Restlet provides its own internal HTTP server and all you need to implement server-side RESTful applications, i-jetty might be useful if you need Servlet support.

Using Restlet server support on smartphones is a marginal use case, because it's impossible to reach the IP of smartphones through mobile networks due to telco security restrictions. But this issue will become more useful because Android is starting to target other mobile devices, such as tablets and notebooks, using more open wifi networks.

In addition, the Android Cloud to Device Messaging framework (C2DM) helps developers send data from servers to their applications running on Android devices.

9.6

Summary

The edition-based approach of Restlet allows you to use the same Java API (or at least a subset because of specific restrictions) when implementing or accessing RESTful applications across various execution environments. Some of these editions target cloud environments as seen in chapter 8, and others target RIAs and mobile devices, as covered in this chapter.

GWT is a technology that allows the development of rich browser-based applications using the Java language. The client part is compiled to JavaScript and can be executed within a regular web browser. The server side is a full Java-based server application executed by an application server. You can use the client Restlet API within client-side GWT applications to asynchronously call RESTful applications. Although the standard Restlet API is usable, this edition goes further and uses the deferred binding feature of GWT to provide an even simpler way to use Restlet with automatic representation serialization.

Android is a popular technology focused on mobile devices such as smartphones and providing a complete execution environment based on a Linux kernel and a special Dalvik VM. Using Restlet in this environment is possible thanks to its Android edition taking into account restrictions on the client side and server side. As a result, your Android applications can both consume and expose RESTful web APIs.

In this chapter you saw how to deploy Restlet applications to different execution environments using the same API and features. This makes the Restlet Framework a good choice to increase the portability of your client- or server-side code across those various environments, especially if you rely on Google technologies such as Android, GWT, and GAE.

In the next chapter we'll connect the dots between the REST world, its hypermedia roots, and the quickly-growing subject of Linked Data, which is part of the larger Semantic Web. Chapter 10 will give you another great example of the benefits of basing your application architecture on a REST foundation using the Restlet Framework.

Embracing hypermedia and the Semantic Web



This chapter covers

- Hypermedia and why it's important for RESTful web APIs
- Hypertext and hyperdata support in Restlet to drive applications
- The relationship between REST and the Semantic Web
- How Restlet can expose and consume linked data in RDF

As you've seen in the previous chapters, the Restlet Framework was designed on top of REST, the architectural style of the web. One of the benefits of that is that Restlet can be used to build all kinds of web applications, classic websites (Web 1.0), RIAs and web APIs (Web 2.0), and now even Semantic Web Services (Web 3.0).

In this chapter we cover in detail one of the core REST principles: *hypermedia as the engine of application state* (HATEOAS). This principle is important for designing RESTful web APIs but difficult to understand and implement correctly and pervasively. We'll explain ways to support this principle and rely on our RESTful mail application to illustrate design options.

We'll explain how Restlet can support the new hyperdata trend, describing its support for the RDF media type via a dedicated extension, on both the client and server sides. REST and the Semantic Web are a perfect match, and a concrete example of this is Restlet's ability to expose and consume linked data in RDF. As an illustration you'll "semantify" the mail application by adding support for semantic contacts using the FOAF standard format.

Let's get started with a short introduction to hypermedia and its relation to the web and to REST.

10.1 Hypermedia as the engine of RESTful web APIs

The most challenging principle of the REST architecture style to understand and apply correctly is that your application should be driven by hypermedia. This may at first seem cryptic and less important than other REST principles, such as the identification of resources using URIs or the interaction with resources via a uniform interface, but it's the last link that closes the REST design loop.

In this section we'll first describe the HATEOAS principle before defining hypermedia and two of its common specializations, hypertext and hyperdata (and explaining along the way how the Restlet Framework can help you deal with them).

10.1.1 The HATEOAS principle

REST was designed to reduce coupling issues between clients and servers and to allow their independent evolution. This remarkable capability, exploited daily by web browsers that don't need to be recompiled for each website you navigate, relies on the power of hypermedia to progressively discover the next state and available actions of a web application, relying on hyperlinks, embedded scripts, and web forms. This principle is often called HATEOAS, based on chapter 5 of the dissertation that defined REST [12].

HATEOAS means that when accessing a RESTful web API, programmatic clients should be able to dynamically navigate its resources, just as a web browser does with websites, progressively discovering the supported methods, related resources, and so on, thanks to the use of hypermedia. URIs should have no special meaning to clients, even though servers will likely organize them in a specific structure to facilitate implementation of resources, and even though they should be stable to facilitate bookmarking and replaying actions just as with websites.

In general, web API designers prefer to build this knowledge inside API client kits and developer documentation, using custom XML or JSON media types and versioned URIs, as you saw in chapter 6 when we talked about documenting and versioning Restlet applications. One of the reasons is that even though HATEOAS is a core REST principle, designing hypermedia media types isn't a simple task. Let's look closer at this issue, reminding ourselves first what *hypermedia* and *hypertext* mean.

10.1.2 What are hypermedia and hypertext?

Before becoming such a widespread concept, the idea of hypermedia was first envisioned in 1945 by Vannevar Bush in his famous article “As We May Think” [13], describing Memex, a system that would help humans to artificially reproduce their mental associative thinking process.

In 1963 Ted Nelson invented the terms *hypermedia* and *hypertext* and tried to illustrate how they could work in an ambitious project named Xanadu, described in an extensive article in *Wired* [14]. Here’s his original definition from his book *Literary Machines*: “By ‘hypertext’ I mean nonsequential writing—text that branches and allows choice to the reader, best read at an interactive screen.”

Even though Xanadu never became complete or usable, it was a great source of inspiration for products such as Hypercard, first published by Apple in 1987, and later the World Wide Web, with HTML as a simplified variant.

Combined with HTTP, HTML found a sweet spot inside the emerging internet and became a central aspect of the REST architecture style. The original WWW document from Tim Berners-Lee and Robert Cailliau in 1990 was titled “WorldWideWeb: Proposal for a HyperText Project,” underlying how critical hypertext and hypermedia were to the web. Here’s the first sentence of their founding email: “HyperText is a way to link and access information of various kinds as a web of nodes in which the user can browse at will. It provides a single user-interface to large classes of information (reports, notes, data-bases, computer documentation, and on-line help).” Over the years, several other formats added hypermedia capabilities, such as:

- PDF, Word, PowerPoint, Excel, and similar office productivity tools that allow inclusion of hyperlinks to web documents, potentially in the same format
- SVG vector image, which embed XLinks to other web documents
- Atom feeds, including links to other blog posts or resources

All those popular formats required significant effort and time to be designed and correctly implemented, but none is as popular as HTML. HTML is lightweight, human-readable, extremely versatile, and interoperable. In addition, it has a built-in forms feature that complements hyperlinks as a way to animate the application, discover next available transitions that a user can follow to navigate existing resources, change their state, create new ones, and so on.

10.1.3 Hypertext support in Restlet

As you’ve seen in previous chapters, there are several ways to produce HTML representations using the Restlet Framework. Here are the main options to remember:

- Use a template representation from the FreeMarker or the Velocity extensions, which can be combined with a data model to produce an HTML page to be rendered by a browser. This is similar to the JSP approach.

- Use an XML representation, built using either SAX or DOM APIs or retrieved via another means, such as a third-party web API, and wrap it with an XSLT representation as explained in section 4.2, in order to produce HTML output.
- Build the HTML document programmatically by appending elements, attributes, and content to an AppendableRepresentation and sending it back to a browser. This works well for short HTML documents that can be held in memory.
- Build the HTML document programmatically by extending the `write(Writer)` method of a WriterRepresentation as illustrated in section 8.4.1 and sending it back to a browser. This is a little bit harder to program than the previous option, but can produce an HTML document of any length without requiring large amounts of memory, because the content is progressively streamed to the browser. This is perfect when you convert large amounts of data retrieved from a database on the fly to HTML.
- Serve static HTML documents from disk using a FileRepresentation, or even serve static directories like a regular web server, using the Directory class as explained in section 8.1.3.

In addition, Restlet can add forms to HTML documents using the previous options and process posted form data sets using either the built-in `org.restlet.data.Form` class for simple URL encoded forms, as explained in section 8.1.1, or the Apache FileUpload extension, illustrated in section 8.1.5 for multipart postings, typically including a mix of regular fields and uploaded files.

Turning to the client side, the typical HTML client is a web browser, but sometimes you need to do a form posting programmatically. For simple forms, the `Form` class will work fine. But to send multipart forms you'll need to use the new `org.restlet.ext.html` extension introduced in version 2.1 of Restlet, including a `FormDataSet` class.

In our context, the problem with HTML is that it was primarily intended for hypertext representations displayed in web browsers, with a human interacting with the application. Most web APIs are used by programmatic clients that need to exchange what are typically stable data structures. This is clearly the main difficulty faced by web APIs that want to be RESTful. As noted by Roy T. Fielding in his blog [15], if you don't respect this principle, you shouldn't mark your web API as RESTful.

As a compromise, you could say it's *inspired by REST*, or *REST-like*, but there's clearly a need to mix structured data exchanges with hypertext and hypermedia in a way that's usable by programmatic clients and not only humans. This new trend is often called *hyperdata*, illustrated in figure 10.1. It's the topic of our next section.

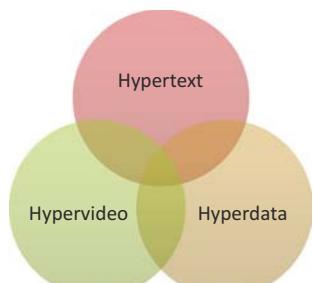


Figure 10.1 Hyperdata and other types of hypermedia

10.1.4 The new hyperdata trend

When you imagine your web API, you probably first visualize it from the server point of view, in terms of resource classes, hierarchies of URIs, or XML- or JSON-based data structures exchanged with clients. For a more guided approach, the ROA/D methodology described in appendix D proposes precise steps to develop your web API.

To make it truly RESTful you also need to put yourself in your clients' shoes and make sure that their coupling with your server is minimal, ideally only consisting of some base URIs, knowledge of the uniform interface used (typically HTTP standard methods), and finally knowledge of the media type. Let's focus here on the last point and see more concretely how you can make the XML- or JSON-based media types suitable for REST.

In the next subsections, you'll see three main approaches you can follow, not necessarily in an exclusive way.

USING STANDARD REPRESENTATIONS

The easiest approach is to rely on a standard media type such as Atom and its sister AtomPub standards to define how to exchange structured content such as blog entries, eventually extending it, as GData from Google and OData from Microsoft are doing.

Even though Atom and AtomPub are based on XML, retrieving the content in JSON form is generally possible as an alternative. The main benefit from this approach is that you increase the interoperability of your web API, because more clients will be able to access and understand its language. The main drawback is that you need to adapt or wrap your domain data inside those standard structures, which leads to additional complexity. Unless you're building a blog or a notification application, you can end up using the Atom structure as an unnecessary envelope for your own data.

MIXING MICROFORMATS AND HTML REPRESENTATIONS

The second possibility is to use HTML and insert structured data directly inside it, such as business cards, calendar events, social connections, and so on. This approach has been pioneered by the Microformats community [16] and is interesting because search engines like Google and Bing will detect the additional data and use it to enrich their search results.

But for programmatic clients the use bar is high, because parsing HTML documents is already a complex task, and parsing additional embedded data is even harder. Although it's a great idea to expose your API resources in HTML with embedded data, you should consider this as one option among other media types by supporting HTTP content negotiation, as explained in section 5.5.

CREATING CUSTOM REPRESENTATIONS

The remaining common possibility is to create your own XML- or JSON-based media types from scratch and add hypermedia capabilities to them. The classic web of HTML pages gains a lot of its power from the inclusion of hyperlinks to other pages. This gives modularity, reusability, and navigability of content. In your own custom XML or JSON media types, you can apply the same idea and use the ability to embed links to

other resources in the data you exchange between client and servers, just as you're used to doing with HTML documents.

In the email system you expose resources such as mails, accounts, and so forth. In addition to HTML representations aimed at HTML clients, these resources will provide XML representations for programmatic clients that are interested only in raw data. For example, Mr. Homer Simpson, one of the users of the system, has an account at www.rmep.org/accounts/chunkylover53/. The representation of this resource (the data clients and server exchange when describing this user account) could be some XML containing the name of the user, its login, the name used when sending a message using this account, and so on:

```
<account>
  <firstName>Homer</firstName>
  <lastName>Simpson</lastName>
  <login>chunkylover53</login>
  <nickname>Personal mailbox of Homer</nickname>
  <senderName>Homer</senderName>
  ...
</account>
```

Likewise, the representation for a resource exposing a mail item could be some XML including its subject line, its content, its status (draft, sent, received), and so on. Each mail item is also stored in a particular account. In the representation of the mail resource you can refer to this account by using the URI as a hyperlink, as shown in the following snippet, where the account in question is none other than Mr. Simpson's:

```
<mail>
  <status>received</status>
  <subject>Message to self</subject>
  <content>Doh!</content>
  <accountRef>
    http://www.rmep.org/accounts/chunkylover53/
  </accountRef>
  ...
</mail>
```

A programmatic client that is handed the representation of a given mail item can then, if needed, interact with the account in which the mail is stored using the URI reference. The client application can get the representation of the account using the GET method and modify it with other methods such as PUT, POST, and DELETE, if allowed.

Using URIs like this to refer to other resources is easy and straightforward and will make your web API closer to being RESTful, with reasonable design and implementation effort.

But a limitation of this custom media type approach is that it restricts the clients that can use your web API. If you don't have the ability to develop and distribute client kits or libraries for popular platforms such as Java, .NET, Python, PHP, Android, or iOS, your clients will first need to develop parsers and formatters for your media types, which is a barrier for use and interoperability.

Whichever option you choose for your representations, you should expect to spend as much time on their design as you do for other web API aspects, including URI namespace structure.

As you've seen, interoperability between RESTful clients and servers based on standard formats is important and is a key factor in the success of the classic web-based on the HTML, HTTP, and URI trio. But it's possible to go beyond this when developing hyperdata formats. The idea is to add a level of interoperability among the data itself, using the Semantic Web and its RDF standard format in a lightweight and pragmatic way. This approach is often called *Linked Data* and is the topic of the next section.

10.2 The Semantic Web with Linked Data

As you've seen, hypermedia isn't only about hypertext media types, but is also about hyperdata and, more importantly, interoperability of data. This goal is shared by the Semantic Web, which was also initiated by Tim Berners-Lee.

In this section we first give a brief overview of the Semantic Web and its relationship to REST via the new Linked Data initiative, which is a great illustration of hyperdata. Then we introduce RDF, the core standard for semantic representations, and explain how to expose, consume, and browse linked data with Restlet. This is also a good time to introduce FOAF, an RDF vocabulary to describe social relationships.

10.2.1 REST and the Semantic Web

The Semantic Web is an ambitious initiative that was publicly launched in 2001 by a now-famous *Scientific American* article [17] that resulted in great expectations. The work on its foundations involved many researchers across the globe and took a significant amount of time. During this time, developers and companies were left wondering how they could best take advantage of the specifications that had started to come out of the W3C, such as RDF, RDF Schema, OWL, and SPARQL. An impression of excessive complexity started to emerge along with the perceived lack of real-world use cases where the Semantic Web could shine.

That's when Tim Berners-Lee introduced his Linked Data idea, as an application of the Semantic Web that would allow browsing of semantically linked resources. Instead of storing semantic data in large specialized databases and requiring a special language to interact with them, the idea was to use the web and its HTTP protocol as a way to interact directly with the graph of semantic data, using hyperlinks to jump from one hyperdata document to another, just as we do with hypertext documents.

Linked Data finally offers a pragmatic and operational approach to the Semantic Web that's also perfectly in line with REST principles, including HATEOAS. This lighter approach was the foundation of Restlet support for the Semantic Web, available in the `org.restlet.ext.rdf` extension.

Concretely, Linked Data relies on URIs (which are HTTP URLs) to identify important data, in the same way that they identify documents on the regular web. It also relies on HTTP to retrieve, create, update, or delete the data, as with other RESTful web APIs.

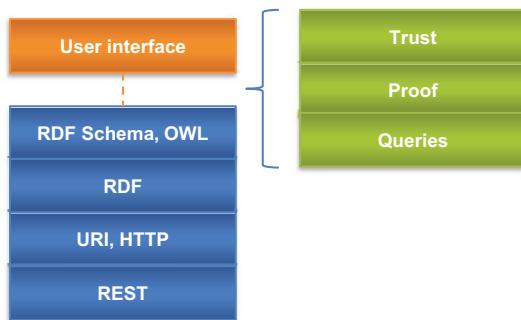


Figure 10.2 The Linked Data technological stack

Restlet and its semantic roots

When the Restlet Framework launched in 2005, it was the result of extracting a generic piece of code from a website project called Semalink, aimed at facilitating the adoption of the Semantic Web by closing the gap with the regular web of documents. Later on, the success of the Linked Data initiative and its support in Restlet via the RDF extension were in a way a return to the project roots.

As illustrated in figure 10.2, Linked Data relies on the RDF language (on top of the REST, URI, and HTTP lower layers) to represent those data resources and their relationships with other resources and their attributes. Finally we have RDF Schema and its richer cousin, Ontology Web Language (OWL), which are languages (also expressed in RDF) used to define valid RDF graph structures called *ontologies*, or *meta-models* if you're more familiar with model-driven engineering.

In the long term you can expect other layers of the Semantic Web vision to find their way inside this stack to solve issues such as distributed queries, proof, and trust. There are already proposals to address these needs, such as SPARQL to query RDF databases. Some of these were defined before the Linked Data initiative and will probably have to be rethought to fit better with the regular web in order to reach a broader use level.

Regarding trust, there's a WebID protocol [3] in the W3C Incubator which aims at using HTTPS and client SSL certificates to build a web of trust in a pragmatic way that can nicely complement Linked Data. We've mentioned RDF several times, so now it's time to have a closer look at it and see how to use it for resource representations.

10.2.2 Using RDF in representations

RDF is the acronym for Resource Description Framework, where a resource has the same meaning as in the REST architecture style—something of interest that can be addressed by a URI. As its name implies, RDF provides a way to describe and represent web resources in a precise and interoperable manner. To help you understand how RDF works, we first present the RDF data model, explaining the main concepts involved and how they relate to REST, and discuss the serialization media types available for this data model.

To make this discussion more concrete, we'll use examples from the FOAF language, which lets you express social links between people, a much simpler but open and semantic variant of Facebook or LinkedIn data sets.

RDF DATA MODEL

In RDF all the data is defined as a graph, where nodes are either resources identified by a URI or literals (like a string or an integer) and where links connect either one resource to another or a resource and a literal defined as an attribute value. Those links are also frequently called statements, triples, or properties.

Figure 10.3 partially describes the Homer Simpson resource as a graph with a central resource node, three literal nodes on the left, and three related resource nodes on the right.

As you can see, the links have labels defining how Homer relates to the Marge, Bart, and Lisa resources and the meaning of the "Homer," "Simpson," and "homer@simpson.org" literals. Obviously, Homer knows his wife and children, and we could have added more links to express the exact familial relationships, such as father and husband. But in this case we decided to follow the FOAF vocabulary [18], which is less interested in family relationships than in generic links between people.

With the previous example, you almost have a valid RDF graph. The piece that's missing is unambiguous information telling you that the links are related to FOAF and not to another vocabulary. For this purpose, RDF relies again on URIs to precisely and uniquely define the meaning of those links.

This is the most important difference with hyperlinks found in HTML documents: better interoperability and the ability to have several links between the same pair of resources. In this case, the exact value of *knows* is <http://xmlns.com/foaf/0.1/knows>, the value of *mbox* is <http://xmlns.com/foaf/0.1/mbox>, and so on.

Let's create this example RDF graph using the RDF extension of the Restlet Framework available in the org.restlet.ext.rdf.jar file. As illustrated in the following listing, the translation is straightforward, because you're able to reuse the Reference class from the org.restlet.data package to define URI references.

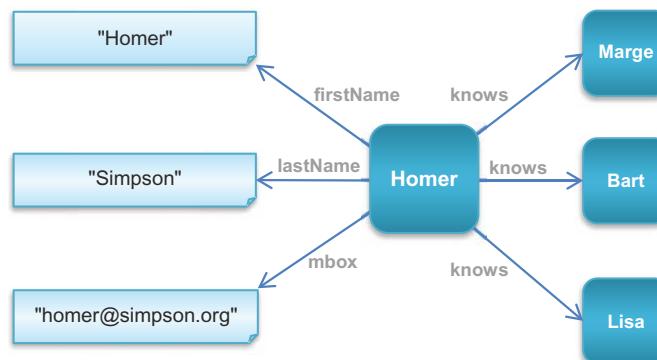


Figure 10.3 Example RDF graph partially describing Homer Simpson

Listing 10.1 Creating an RDF graph with Restlet RDF extension

```

import org.restlet.data.Reference;
import org.restlet.ext.rdf.Graph;
import org.restlet.ext.rdf.Literal;

public class FoafExample {

    public static void main(String[] args) throws IOException {
        String FOAF_BASE = "http://xmlns.com/foaf/0.1/";
        Reference firstName = new Reference(FOAF_BASE + "firstName");
        Reference lastName = new Reference(FOAF_BASE + "lastName");
        Reference mbox = new Reference(FOAF_BASE + "mbox");

        Reference homerRef = new Reference(
            "http://www.rmep.org/accounts/chunkylover53/");
        Reference margeRef = new Reference(
            "http://www.rmep.org/accounts/bretzels34/");
        Reference bartRef = new Reference(
            "http://www.rmep.org/accounts/jojo10/");
        Reference lisaRef = new Reference(
            "http://www.rmep.org/accounts/lisa1984");

        Graph example = new Graph();
        example.add(homerRef, firstName, new Literal("Homer"));
        example.add(homerRef, lastName, new Literal("Simpson"));
        example.add(homerRef, mbox, new Literal(
            "mailto:homer@simpson.org"));
        example.add(homerRef, knows, margeRef);
        example.add(homerRef, knows, bartRef);
        example.add(homerRef, knows, lisaRef);
    }
}

```

FOAF ontology constants

Linked Simpson resources

Example RDF graph

As introduced in figure 10.2, it's also possible to define the structure of valid RDF graphs as ontologies using the RDF Schema and OWL languages. *Ontology* might sound like a daunting term, but in this setting it refers to a set of object classes where the relations between classes and attributes are typed using a URI rather than a potentially ambiguous and imprecise label.

Figure 10.4 illustrates how to visualize a part of the FOAF ontology as a regular UML class diagram. If you read appendix D on the ROA/D methodology, you should be able to see how remarkably this diagram can complement figure D.15 and the Account resource class. An RDF class defined as Linked Data can be exactly the same as a REST resource class defined in a web API (see section D.4.3, “Identifying and classifying the resources,” for details), but adding further information about its attributes and relationships with other resource classes, such as other persons, images, and documents in the FOAF case.

Let's move beyond the abstract RDF data model and see how to serialize RDF graphs and use them as representations.

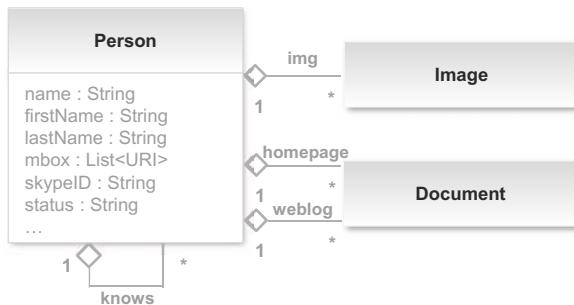


Figure 10.4 Person class in the FOAF vocabulary

RDF REPRESENTATION VARIANTS

Contrary to XML vocabularies such as Atom or XHTML, RDF doesn't force you to use a single serialization format. Even though the primary format is XML-based, there are others available:

- RDF/XML, a comprehensive XML serialization format for RDF
- Notation 3 (or n3), a compact alternative to RDF/XML also able to express rules
- Turtle, a subset of n3, simple and human-readable
- N-Triples, an even simpler subset of Turtle, useful for storing and exchanging RDF

To make those formats more concrete, you'll now serialize the example FOAF graph built with the Restlet extension. Let's add the following lines of code to the code in listing 10.2:

```

System.out.println("\nRDF/XML format:\n");
example.getRdfXmlRepresentation().write(System.out);

System.out.println("\nRDF/n3 format:\n");
example.getRdfN3Representation().write(System.out);

System.out.println("\nRDF/Turtle format:\n");
example.getRdfTurtleRepresentation().write(System.out);

System.out.println("\nRDF/NTriples format:\n");
example.getRdfNTriplesRepresentation().write(System.out);
  
```

The `getRdf*Representation()` methods create an instance of the `RdfRepresentation` class, passing it the `Graph` instance and the proper media type constant. Now, if you run this code, you'll first serialize the graph into RDF/XML. The key XML element is `rdf:Description`, which contains all the properties related to the Homer resource identified by the XML attribute `rdf:about`. Note also how a prefix `_NS1` was declared for the FOAF ontology URI:

```

<?xml version="1.0" standalone='yes'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:_NS1="http://xmlns.com/foaf/0.1/">
  <rdf:Description rdf:about="http://www.rmep.org/accounts/chunkylover53/">
    <_NS1:firstName>Homer</_NS1:firstName>
    <_NS1:lastName>Simpson</_NS1:lastName>
    <_NS1:mbox>mailto:homer@simpson.org</_NS1:mbox>
  
```

```

<__NS1:knows>
  <rdf:Description rdf:about="http://www.rmep.org/accounts/bretzels34"/>
/></__NS1:knows>
<__NS1:knows>
  <rdf:Description rdf:about="http://www.rmep.org/accounts/jojo10"/>
></__NS1:knows>
<__NS1:knows>
  <rdf:Description rdf:about="http://www.rmep.org/accounts/lisa1984"/>
></__NS1:knows>
</rdf:Description>
</rdf:RDF>

```

The second and third serializations are for n3 and Turtle formats and produce the same result. Note in the following snippet that the graph is all written in one line starting with `<http://www.rmep.org/accounts/chunkylover53/>`. In addition you can use namespace as illustrated in the first lines:

```

@prefix #: <:>.
@prefix rdfs: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix type: <http://www.w3.org/2001/XMLSchema#>.
@prefix rdf: <http://www.w3.org/2000/01/rdf-schema#>.
@keywords a, is, of, has.
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/firstName> "Homer";
<http://xmlns.com/foaf/0.1/lastName> "Simpson";
<http://xmlns.com/foaf/0.1/mbox> "mailto:homer@simpson.org";
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/bretzels34/>,
<http://www.rmep.org/accounts/jojo10/>,
<http://www.rmep.org/accounts/lisa1984/>.

```

Let's see how the fourth and simplest format, N-Triples, serializes the graph. For each link there's a new line, and no prefixes or factorizations are used. The result is more verbose but also straightforward to write, read, and understand:

```

<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/firstName> "Homer".
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/lastName> "Simpson".
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/mbox> "mailto:homer@simpson.org".
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/bretzels34/>.
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/jojo10/>.
<http://www.rmep.org/accounts/chunkylover53/>
<http://xmlns.com/foaf/0.1/knows>
<http://www.rmep.org/accounts/lisa1984/>.

```

In addition to those pure RDF serializations, it's also possible to embed RDF inside other documents such as XHTML pages to enrich them with semantic data. The W3C-supported way to do this is RDFa, but similar efforts have been proposed, such as

Microformats and HTML 5 Microdata [19]. In the following snippet you can see one way to express the example graph as an XHTML page with RDFa special attributes:

```
<div xmlns:foaf="http://xmlns.com/foaf/0.1/"
    about="http://www.rmep.org/accounts/chunkylover53/">
    <span property="foaf:firstName">Homer</span>
    <span property="foaf:lastName">Simpson</span>
    <a rel="foaf:mbox" href="mailto:homer@simpson.org">
        homer@simpson.org</a>
        <a rel="foaf:knows" href="http://www.rmep.org/accounts/bretzels34/">
        Marge </a>
        <a rel="foaf:knows" href="http://www.rmep.org/accounts/jojo10/">
        Bart</a>
        <a rel="foaf:knows" href="http://www.rmep.org/accounts/lisa1984/">
        Lisa</a>
    </div>
```

The mixed approach illustrated by RDFa makes it easy to embed semantic data in web pages but also makes it harder to extract the data back into proper RDF. To solve these issues W3C proposed GRDDL as a standard way to extract the RDF data by applying XSLT stylesheets to the XHTML documents.

The Schema.org initiative

In June 2011, Google, Bing, and Yahoo! launched the Schema.org initiative to facilitate semantic annotation of data in regular web pages. They provide a way to express semantic data using HTML Microdata, including both common ontologies for things such as Persons, Places, Organizations, and so on, and support for their extraction in the most popular web search engines. A mapping to RDFa is also specified, making it a great option to consider for mixing RDF and web pages.

At this point, you should understand how powerful the RDF data model is—capable of modeling anything in a precise way—and how flexible it can be used in representation formats depending on the use context. In the next section you use RDF and its support in the Restlet Framework to expose the mail accounts as Linked Data.

10.3 Exposing and consuming Linked Data with Restlet

In this section you restore the example mail application and expose two variants of the account resources, one in XML and one in RDF, using the FOAF vocabulary. Finally we look at the client side and see how to consume Linked Data and navigate from one resource to another by following RDF links.

10.3.1 Exposing RDF resources

As you saw in listing 10.1, the RDF extension of Restlet comes with a DOM-like API composed of the following classes:

- Graph contains links and can produce an RDF representation.

- Link is equivalent to an RDF statement or a triple composed of a source resource URI reference, a link type URI reference, and a target value, either literal or resource.
- Literal provides target values along with a datatype URI reference and language.

Once you've built a Graph instance, you can return it directly via a Restlet resource with annotated methods. To put this into practice in the example application, you first need to complete the account resource to expose a richer domain model than in previous chapters, where you only used a simple string.

Figure 10.5 shows enriched Account and Contact classes with new properties and relationships. Note how it resembles figure D.8 in appendix D (covering the ROA/D methodology, applied to our mail example), presenting only a part of the whole domain model but with more detail on the available properties.

Beyond the obvious properties in Account, the profileRef property in Contact is supposed to contain a URI reference to the FOAF profile of the contact. If the contact is also managed by the RESTful mail application, the URI will refer to the related Account resource (which will have a FOAF variant exposed).

The following listing initializes the domain model with four user accounts—for Homer Simpson and the three other members of his family: Marge, Bart, and Lisa.

Listing 10.2 Setting up the domain model with user accounts

```
public MailServerApplication() {
    setName("RESTful Mail API application");
    setDescription("Example API for 'Restlet in Action' book");
    setOwner("Restlet SAS");
    setAuthor("The Restlet Team");

    Account homer = new Account();
    homer.setFirstName("Homer");
    homer.setLastName("Simpson");
    homer.setLogin("chunkylover53");
    homer.setNickName("Personal mailbox of Homer");
    homer.setSenderName("Homer");
    homer.setEmailAddress("homer@simpson.org");
    homer.getContacts().add(new Contact("/accounts/bretzels34/"));
    homer.getContacts().add(new Contact("/accounts/jojo10/"));
    homer.getContacts().add(new Contact("/accounts/lisa1984/"));
    getAccounts().put("chunkylover53", homer);

    Account marge = new Account();
    marge.setFirstName("Marjorie");
    marge.setLastName("Simpson");
    marge.setLogin("bretzels34");
    marge.setNickName("Personal mailbox of Marge");
```

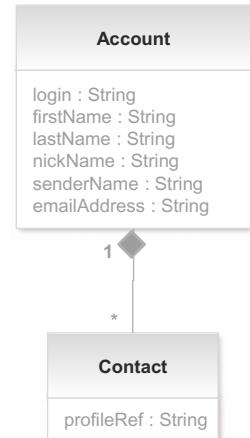


Figure 10.5 RESTful mail example domain object model

Homer user account

Marge user account

```

marge.setSenderName("Marge");
marge.setEmailAddress("homer@simpson.org");
marge.getContacts().add(new Contact("/accounts/chunkylover53/"));
marge.getContacts().add(new Contact("/accounts/jojo10/"));
marge.getContacts().add(new Contact("/accounts/lisa1984/"));
getAccounts().put("bretzels34", marge);

Account bart = new Account();                                ← Bart user account
bart.setFirstName("Bartholomew");
bart.setLastName("Simpson");
bart.setLogin("jojo10");
bart.setNickName("Personal mailbox of Bart");
bart.setSenderName("Bart");
bart.setEmailAddress("bart@simpson.org");
bart.getContacts().add(new Contact("/accounts/chunkylover53/"));
bart.getContacts().add(new Contact("/accounts/bretzels34/"));
bart.getContacts().add(new Contact("/accounts/lisa1984/"));
getAccounts().put("jojo10", bart);

Account lisa = new Account();                                ← Lisa user account
lisa.setFirstName("Lisa");
lisa.setLastName("Simpson");
lisa.setLogin("lisa1984");
lisa.setNickName("Personal mailbox of Lisa");
lisa.setSenderName("Lisa");
lisa.setEmailAddress("lisa@simpson.org");
lisa.getContacts().add(new Contact("/accounts/chunkylover53/"));
lisa.getContacts().add(new Contact("/accounts/bretzels34/"));
lisa.getContacts().add(new Contact("/accounts/jojo10/"));
getAccounts().put("lisa1984", lisa);
}

```

Let's now enhance the `AccountResource` annotated interface to expose the FOAF variant representation. As illustrated in listing 10.3, our first choice would have been to use a generic `AccountRepresentation` bean, to offer automatic conversion with XML, JSON, and similar formats using the XStream, Jackson, and similar Restlet extensions, as detailed in chapter 4. But for RDF representations we don't yet have an integrated and automated solution to annotate beans to produce proper RDF, even though existing open source projects [20] could be integrated to Restlet to achieve this. As a work-around you declare another `getFoafProfile()` method annotated with `@Get` but this time specifying with the "`rdf`" annotation value that this only returns RDF variants, either in RDF/XML, RDF/n3, Turtle, or N-Triples format.

Listing 10.3 Enhanced Account annotated resource interface

```

public interface AccountResource {
    @Get
    public AccountRepresentation represent();

    @Get("rdf")
    public Graph getFoafProfile();
}

```

Let's continue by providing the implementation of this enhanced resource via the `AccountServerResource` class, shown in listing 10.4. The implementation of both annotated methods is similar in the sense that they both populate a representation, using simple Java properties in the first case and URI properties in the second case, based on the FOAF ontology. Both approaches have benefits and drawbacks in terms of simplicity and interoperability.

The first approach is simpler because regular POJOs are used, but the interoperability is limited because the clients must have knowledge of those properties and their meanings to interpret and make good use of them.

The second approach is more complex because you need to type each property using a URI based on a parent ontology, which is less natural for Java developers. The benefit is that interoperability is improved because clients will be able to interpret the data at a higher level than raw XML. If they understand the ontology (such as the one defined by FOAF), they can precisely interpret the resulting semantic representations.

One of the advantages of using Restlet is that you can use both approaches at the same time, without additional development cost, as illustrated in the following listing.

Listing 10.4 Enhanced Account server resource

```
public class AccountServerResource extends ServerResource implements
    AccountResource {
    private Account account;                                     ← | The associated
    public Map<String, Account> getAccounts() {                  |
        return ((MailServerApplication) getApplication()).getAccounts(); |
    }
    @Override
    protected void doInit() throws ResourceException {
        String accountId = getAttribute("accountId");
        this.account = getAccounts().get(accountId);
    }
    public void remove() {
        getAccounts().remove(this.accountId);
    }
    public AccountRepresentation represent() {                   ← | Build representation
        AccountRepresentation result = null;
        if (account != null) {
            result = new AccountRepresentation();
            result.setEmailAddress(account.getEmailAddress());
            result.setFirstName(account.getFirstName());
            result.setLastName(account.getLastName());
            result.setLogin(account.getLogin());
            result.setNickName(account.getNickName());
            result.setSenderName(account.getSenderName());
            for (Contact contact : account.getContacts()) {
                result.getContactRefs().add(contact.getProfileRef());
            }
        }
    }
}
```

```

        return result;
    }

    public Graph getFoafProfile() {
        Graph result = null;

        if (account != null) {
            result = new Graph();
            result.add(getReference(), FoafConstants.MBOX,
                new Literal("mailto:" + account.getEmailAddress()));
            result.add(getReference(), FoafConstants.FIRST_NAME,
                new Literal(account.getFirstName()));
            result.add(getReference(), FoafConstants.LAST_NAME,
                new Literal(account.getLastName()));
            result.add(getReference(), FoafConstants.NICK,
                new Literal(account.getNickName()));
            result.add(getReference(), FoafConstants.NAME,
                new Literal(account.getSenderName()));

            for (Contact contact : account.getContacts()) {
                result.add(getReference(), FoafConstants.KNOWS,
                    new Reference(getReference(),
                        contact.getProfileRef().getTargetRef()));
            }
        }

        return result;
    }
}

```

Build RDF graph

When you return the `Graph` instance from the `getFoafProfile()` method, the `ConverterService` takes over the process. If you've correctly added the `org.restlet.ext.rdf.jar` in your classpath, the `RdfConverter` will automatically wrap the `Graph` instance into an `RdfRepresentation` using the correct RDF media type based on client preferences.

At this point we encourage you to launch the server component and retrieve both representations from a web browser using the following URIs:

```

http://localhost:8111/accounts/chunkylover53/?media=xml
http://localhost:8111/accounts/chunkylover53/?media=rdf

```

As a result, you should obtain XML and FOAF documents similar to the one presented in sections 10.1.4 and 10.2.2. The mail example application is now not only RESTful but also part of the Linked Data!

SAX-LIKE RDF PROCESSING

In addition, the RDF extension is also capable of handling large RDF representations. The issue with the DOM-like `Graph` approach is that all links are stored in memory before being serialized or after being parsed.

For this purpose, the extension comes with a `GraphHandler` abstract class that's similar in spirit to the SAX `ContentHandler` interface. You can use this class for both RDF parsing and writing purposes, and it contains a callback method for RDF processing events such as `startGraph()`, `endGraph()`, and `link(...)`.

We won't cover this SAX-like RDF feature in detail here, but if you want to experiment with it, you should look at the `GraphBuilder` class provided, which is a subclass of `GraphHandler` creating a `Graph` instance when parsing an RDF representation. There's also `RdfRepresentation` and its `parse(GraphHandler)` and `write(GraphHandler)` methods, which you can override to provide custom handling.

TEMPLATE RDF GENERATION

Remember that it's always possible to produce RDF representations using template engines like FreeMarker and Velocity, if that fits your use case. There's nothing wrong with this approach; it can easily give you precise control over RDF formatting.

Let's move to the client side and see how you can consume—and, more important, browse—your Linked Data mail application.

10.3.2 Consuming linked data with Restlet

In this section we'll explain how Restlet can consume RDF representations and we'll address the two main challenges you face when consuming linked data.

The first challenge is the need to support several RDF formats, such as RDF/XML and RDF/n3. For this purpose, the Restlet extension for RDF gives you an abstraction layer with the `RdfRepresentation` class, which is capable of using the correct parser based on the media type of the RDF representation returned by an origin server.

The second challenge is to have the ability to easily navigate among hyperlinked resources based on your use case. Again the RDF extension has a handy solution thanks to its `RdfClientResource`, which adds class methods to `ClientResource` for retrieving linked RDF resources and literal-valued properties.

Another challenge is to add knowledge to your clients of the RDF vocabularies (also known as ontologies) commonly used in Linked Data, and sometimes to map from one ontology to another. Here the best practice is to use existing ontologies such as RDF Schema, OWL, Dublin Core or FOAF as much as possible.

Let's get back to coding and try to consume the account resources using RDF. You could use the annotated `AccountResource` interface, but let's assume that the backend isn't necessarily written in Restlet. After all, a web API should be accessible from any kind of HTTP client. Listing 10.5 only provides the URI of Homer's account, and still you're able to display the literal value of each of its properties, as well as the properties of each of its contacts, by navigating to the linked FOAF profile based on the `FoafConstants.KNOWS` URI (<http://xmlns.com/foaf/0.1/knows>).

Listing 10.5 Generic FOAF browser

```
public class FoafBrowser {  
  
    public static void main(String[] args) {  
        displayFoafProfile("http://localhost:8111/accounts/chunkylover53/");  
    }  
  
    public static void displayFoafProfile(String uri) {  
        displayFoafProfile(new RdfClientResource(uri), 1);  
    }  
}
```

Launch
FOAF
browsing

```

public static void displayFoafProfile(RdfClientResource foafProfile, ←
    int maxDepth) {
    Set<Couple<Reference, Literal>> literals = foafProfile.getLiterals();

    if (literals != null) {
        for (Couple<Reference, Literal> literal : literals) {
            System.out.println(literal.getFirst().getLastSegment() + ":" +
                + literal.getSecond());
        }
    }

    System.out.println("-----");
    if (maxDepth > 0) {
        Set<RdfClientResource> knows = foafProfile
            .getLinked(FoafConstants.KNOWS);

        if (knows != null) {
            for (RdfClientResource know : knows) {
                displayFoafProfile(know, maxDepth - 1);
            }
        }
    }
}
}

```

Recursive FOAF display

If you launch this FoafBrowser after starting the server, you should see the following output in the console, plus some log messages related to the HTTP connector launch:

```

lastName: Simpson
firstName: Homer
nick: Personal mailbox of Homer
name: Homer
mbox: mailto:homer@simpson.org
-----
lastName: Simpson
mbox: mailto:lisa@simpson.org
name: Lisa
firstName: Lisa
nick: Personal mailbox of Lisa
-----
lastName: Simpson
firstName: Bartholomew
mbox: mailto:bart@simpson.org
nick: Personal mailbox of Bart
name: Bart
-----
firstName: Marjorie
nick: Personal mailbox of Marge
name: Marge
mbox: mailto:homer@simpson.org
lastName: Simpson
-----
```

Although that approach is convenient and expressive, it requires you to load all of the DF representations in memory, in a DOM-like way. If you need to load larger representations or have a finer-grained control of your RDF client, you can also use the

RdfRepresentation class directly, in a SAX-like way. For this purpose you can use the constructor that takes a Representation parameter and then invoke the parse (GraphHandler) method to consume its content as a series of Link instances, one link at a time.

As you've seen in this section, consuming linked resources using RDF representations is as easy as exposing them with the Restlet extension for RDF. It provides a solution for most of the challenges you'll face with only a few additional classes, mainly RdfClientResource and RdfRepresentation.

10.4 **Summary**

This chapter covered a lot of ground, including advanced topics that many web API developers aren't initially aware of. Awareness of these topics is increasingly important for the future of RESTful web APIs.

Hypermedia, Linked Data, and above all the Semantic Web are topics that are much too large to cover in this chapter (or even in this book). We tried to present the important parts of the RESTful web in a pragmatic and Restlet-centric way to give you a sense of both the importance and the power of concepts such as HATEOAS and hyperdata, while giving concrete examples of implementation using Restlet on both client and server sides.

We presented the RDF extension of Restlet, which is capable of handling most common RDF serialization formats, such as RDF/XML, RDF/n3, Turtle, and N-Triples, and provides a convenient abstraction layer for both reading and writing, using either a DOM-like or SAX-like approach (to draw a parallel with common XML-processing techniques).

You're now approaching the end of this book, with a final chapter that steps back a little to see what Restlet has to offer beyond what's been covered so far, such as extra extensions available, resources provided by the community that extend or make use of the Restlet Framework, and the roadmap for the next Restlet version.

The future of Restlet

This chapter covers

- State of the HTTP protocol and alternatives
- The rise of SPDY and its impact on REST, HTTP, and Restlet
- Planned enhancements to Restlet API, extensions, and editions
- Planned Restlet Studio tool for Eclipse and Restlet Cloud service

As you near the end of this book, we'll step back and look beyond the detailed coverage of Restlet in action. This final chapter summarizes the state of HTTP and REST and introduces alternatives, such as WebSockets, Server-Sent Events or SPDY (the "SPeeDY" protocol initiated by Google), and puts them into perspective with the evolution of HTTP—in particular, the well-advanced HTTP/1.1 bis initiative.

This is a topic with a major impact on Restlet development because the Restlet API directly maps REST and HTTP concepts, as explained throughout the book and detailed in appendix E. With an HTTP/2.0 version now being planned by IETF, will the Restlet API stay relevant?

We'll review the Restlet roadmap, including evolution of the Restlet API, packaging of the Restlet Forge for multiedition development, as well as the introduction

of the broader Restlet Platform plan, including an open source Restlet Apps suite, a productivity-oriented Restlet Studio based on Eclipse IDE, and a Restlet Cloud hosting facility.

We'll wrap up by introducing the APISpark online web API platform developed by Restlet's creators and explain how it relates to the Restlet Platform, illustrating what you can achieve by putting Restlet in action in an original way! We'll conclude by presenting external Restlet community projects and by explaining how you can contribute back to this open source project.

Let's get started with the evolution of HTTP and see whether the rise of the alternative SPDY protocol supported by Google and Amazon is a threat or an opportunity.

11.1 Evolution of HTTP and the rise of SPDY

REST dates to 2000, but HTTP was created nearly a decade before and is continuing to evolve, though at a slower pace. In this section we review the history of HTTP and discuss alternative protocols, in particular SPDY, and how they could influence HTTP moving forward.

11.1.1 HTTP history so far

The HTTP protocol started as notes written [10] by Tim Berners-Lee in 1991. After an IETF standardization effort, the first standard HTTP/1.0 version (RFC 1945 [21]) was published by Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk in 1996.

With the exponential growth of the web, it quickly became urgent to improve this protocol and address scalability, performance, and interoperability issues observed in the field. This led to the definition of HTTP/1.1 (RFC 2616 [22]) and the formalization of REST [12], the architecture style of the web, with major contributions made by Roy T. Fielding.

HTTP 1.0 is now used by the vast majority of websites, web APIs, and web clients today, but there are still questions regarding the interpretation of its now aging and monolithic specification. In 2007, an IETF working group was launched to revise HTTP/1.1 and address those issues, but without changing the protocol.

Figure 11.1 summarizes the evolution of HTTP over time, up to the formalization of REST in 2000 and added perspectives for future versions, including revision HTTP/1.1 bis.

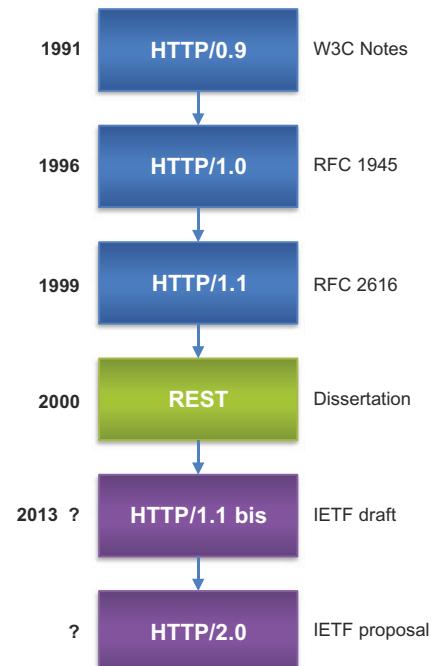


Figure 11.1 Timeline of the HTTP protocol

11.1.2 Refactoring with HTTP/1.1 bis

The HTTP/1.1 bis initiative is led by Mark Nottingham and involves Julian Resche, Roy T. Fielding, Yves Lafon, and a few others. It's now well advanced, and we can hope for this work to be complete this year or soon after.

This rewrite splits the specification into seven parts as illustrated in figure 11.2. At the bottom, we find the Messaging part which deals with connection management, raw message syntax, and HTTP(S) URI schemes. The layers above Messaging describe the semantics of HTTP and the major features offered, such as caching, authentication, conditional requests, and ranged requests. Those layers are exactly what the Restlet API exposes using Java.

In addition, a registration process for standard methods and authentication schemes is proposed via IANA, which already manages media types, HTTP status codes, and encodings.

The timing of this effort is perfect because of the growing pressure to bypass the limitations of HTTP/1.1. Let's look at those alternatives.

11.1.3 The rise of alternatives

Despite being a huge success, several HTTP/1.1 limitations were identified:

- Hard to provide near-real-time client updates
- Excessive use of concurrent sockets by modern browsers
- Excessive bandwidth consumption due to verbose messages

To work around those issues, techniques such as Comet have emerged, pushing HTTP to its limits. Alternative protocols such as WebSocket, Server-Sent Events, and SPDY are also being promoted, introducing both innovations and a risk of fragmentation.

WEB SOCKET

To overcome the real-time communications limits of HTTP between web browsers and servers, the HTML 5 Working Group has initiated a new WebSocket protocol that allows bidirectional exchanges between a browser and a server using HTTP for the initial handshake and upgrading to a full duplex connection.

Using WebSocket, a single TCP connection can be used to send and receive events asynchronously. Figure 11.3 illustrates how WebSocket is positioned to the side of HTTP/1.1 and REST rather than on top of them.



Figure 11.2 Parts of the HTTP 1.1 bis specifications

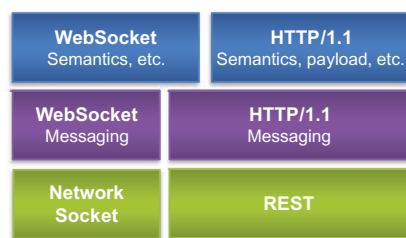


Figure 11.3 WebSocket and REST

This protocol has now been taken over by the IETF in a “BiDirectional or Server-Initiated HTTP (hybi)” working group [23], but it’s unlikely at this point that this protocol will attempt to respect REST principles as discussed in this blog post [24].

Even though the rise of WebSocket has been slowed by security issues in first draft versions, leading some browser vendors to disable early implementations, this new protocol poses a fragmentation risk by bypassing HTTP/1.1 for long-running, event-driven communications. Indeed, by building on top of a TCP connection with loosely enforced semantics, there is a risk that developers will start to overuse it by reinventing unRESTful communication strategies on top of it.

SERVER-SENT EVENTS

For certain applications, HTTP appears to need enhancement. Let’s now look at a simpler yet RESTful alternative also proposed by the HTML 5 Working Group and led by the W3C: Server-Sent Events (SSE). This specification [25] uses HTTP and JavaScript cleanly and proposes a special media type that facilitates implementation of event-driven, near-real-time applications.

Here’s a simple example of SSE representation, identified by the `text/event-stream` media type, which is a stream of lines of text:

```
event: add  
data: 73857293  
  
event: remove  
data: 2153  
  
event: add  
data: 113411
```

The problem with mult tab browsers, Comet, SSE, WebSocket, and other new techniques and protocols is that they require your browser to open and maintain several TCP connections in parallel to the same remote server, posing scalability issues at both ends and increasing network congestion. Even when nonblocking IO can help deal with these issues (as supported by the Restlet Framework), things are more complex than they should be, at least from a network TCP/IP point of view.

SPDY

This is where the SPDY protocol [26] offers an innovative solution. It was introduced by Google as part of the Chromium project [27] (the open source project at the basis of Google Chrome browser). SPDY aims to reduce web page latency and positions itself as an experimental protocol for a faster web.

Besides its experimental support in Google Chrome, SPDY won a strong supporter at the end of 2011 when Amazon announced its Kindle Fire tablet. This tablet came with a new way of browsing the web using limited mobile devices, using the power of the cloud in terms of network connectivity, processing power, and storage capabilities.

The most exciting part of Kindle Fire isn’t its hardware but its Silk browser (<http://amazonsilk.wordpress.com/>) and the cloud browsing infrastructure it uses to achieve what Amazon calls *split browsing*. This radical change in the way we might browse

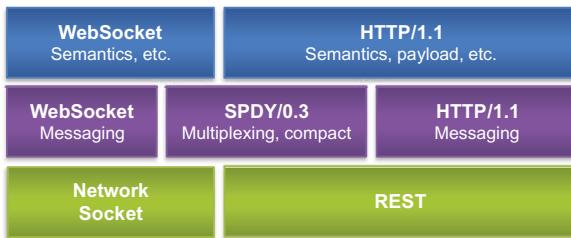


Figure 11.4 SPDY positioning

the mobile web tomorrow is the result of combining traditional caching web proxies used in large organizations to save bandwidth with the capabilities of the Amazon cloud infrastructure.

The goal is to provide a faster browsing experience by adjusting web content (for example, scaling down images) to the capabilities of the device and offering better network latency thanks to the Amazon data centers, which provide the same connectivity and caching capabilities that AWS S3 and CloudFront enjoy. The final and most radical change is the Silk browser's use of the SPDY protocol.

Technically speaking, SPDY multiplexes several HTTP streams over a single TCP connection. Those streams can go in both directions, can be initiated by either the client or server, and are always secured using TLS. SPDY also offers an alternative messaging layer to HTTP/1.1 but intends to stay compatible with all other HTTP layers above, including HTTP/1.1 bis.

As illustrated in figure 11.4, it's also possible to emulate the WebSocket's JavaScript API on top of SPDY using a special option in the Chrome browser.

Is this the beginning of the end of HTTP? Instead of being a threat, SPDY's design qualities and its growing support and usage by key players such as Google and Amazon is more of an opportunity for HTTP to see a version 2.0 defined (see figure 11.5).

This new version would replace the existing messaging part of HTTP/1.1 with a fully multiplexed and compact alternative based on SPDY and inspired by other alternatives such as Roy T. Fielding's own Waka [28] or HTTP-MPLEX.

Looking at HTTP's history, changes to such a fundamental protocol won't happen overnight, but Mark Nottingham wrote [29] about SPDY (ex-FLIP) back in 2009, already talking about a potential HTTP/2.0. In 2012, the HTTP 1.1 bis Working Group has even formally started to work on a HTTP/2.0 proposal, taking into account presentations from the SPDY team among others.

We've discussed the future of SPDY with respect to REST and HTTP. Let's see how it fits into the Restlet roadmap.

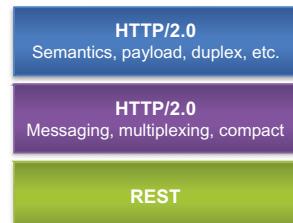


Figure 11.5 Potential HTTP/2.0 development

11.2 The Restlet roadmap

This book covers the 2.1 version of the Restlet Framework, the latest version available at the time of writing. This last chapter is a great place to summarize the roadmap for the next versions and beyond. Let's start with the support for SPDY.

11.2.1 Connectors for SPDY, HTTP and SIP

We've explained how SPDY could become the basis of a future HTTP/2.0 version, retaining compatibility with HTTP/1.1 semantics, so it's natural for Restlet to provide both client and server connectors for this promising protocol.

Those connectors will be built on top of the nonblocking NIO foundation powering the internal HTTP and SIP connectors in version 2.1. The goal is to add connectors compatible with the Google Chrome and Amazon Silk web browsers. For Restlet application developers this feature will be both easy and transparent to take advantage of, only requiring the addition of connectors to the parent Restlet component.

In addition at Restlet we'll continue to improve the internal HTTP and SIP connectors, based on this common NIO core, to bring them to the same level of performance as the extension connectors based on the Jetty and Simple libraries.

The next section lists the main areas of enhancements to the Restlet API itself before looking at the broader Restlet open source project.

11.2.2 Enhancements to the Restlet API

Although the Restlet API is now mature, there are still plans for new features and simplifications. Here we'll review the most important ones considered.

JAVA 6 SUPPORT

The first major change in version 2.2 will be the requirement of Java 6 as the minimum version instead of Java 5, which has been supported since version 1.0. Java 5 has entered the end-of-life process at Oracle, and only paying customers are now receiving security fixes.

In doing this migration, new features available in the Java SE and Java EE editions will be exploited, like the built-in support for the META-INF/services discovery pattern.

NEW CACHESERVICE

In addition, the long-awaited CacheService will be added, including a pluggable mechanism to support cache engines such as file-based caches, in-memory caches, and so on. Currently, the Restlet API has a comprehensive support for HTTP caching metadata and directives but no built-in way to support client-side and server-side caching of content of calls and representations. The goal is to speed up the response of Restlet calls, in particular those generating large and dynamic but rarely changing representations, both on client and server side.

BETTER CONVERTERSERVICE

Although the ConverterService is one of the most powerful Restlet API features, useful enhancements can still be made. Currently this service supports the serialization of

objects into representations or the deserialization of representations into objects. This works fine and uses a plug-in mechanism for extensions such as XStream, Jackson, JAXB, JiBX, and a few more, but it would be great to go beyond this and support other use cases, such as:

- From representation to representation
- From object to representation using a formatting document (such as FreeMarker or Velocity templates)
- From representation to object using a parsing document (such as FreeMarker or Velocity templates)
- From object to object using a transformation document (such as XSLT stylesheets or Jackson's ObjectMapper class)

In addition the Restlet API will provide a way to declare the converters to use for each application as well as their respective order. The goal is to provide a more deterministic way to use multiple converters at the same time, without relying on the classpath order or on the OSGi bundle start level for this (which isn't even possible with WAR files in the Java EE and GAE editions).

UNIFIED BEAN CONVERTER

Currently we rely on extensions to support conversion between representation beans and common formats such as JSON (JSON.org, XStream+Jettison, Jackson), XML (XStream, JAXB, JiBX), RDF or GWT, and so on. But those extensions and the related libraries tend to be redundant and lack integration with REST representation aspects such as metadata and hyperlinks.

As Roy T. Fielding reminds us in his blog post [15], REST APIs must be hypermedia-driven, particularly on the client side, if you want to loosely couple web clients with web servers. Therefore, the goal for next Restlet versions is to help you easily support content negotiation over various media types, with a single, consistent set of representation annotations and supporting common media types such as XML, JSON, HTML/Form post, RDF, or CSV, while using a flexible internal converter.

Because this will be an important development effort, the current extension-based approach will continue to be supported. This strategy is similar to the one underway regarding the internal NIO-based connector.

BETTER CONNEGSERVICE

In addition, there is a need to facilitate content negotiation based on other dimensions than typical representation metadata. For example, the account.html.ftl.ie file could be mapped to the HTML media type, decoded using the FreeMarker template engine, and returned only for Internet Explorer browsers. The vary HTTP header (Response#dimensions property) should also be updated properly. Here is a list of new dimensions that could be taken into account:

- User agent type (example: Firefox)
- User agent category (example: mobile, desktop)

- User agent version
- User preferences
- IP address and domain name (example: UI branding, white-label)
- Authentication state (example: anonymous version)
- Authorization state (example: user role)

EDITIONS SIZE OPTIMIZATION

In addition to previous Restlet API enhancements, a new module profile mechanism will be developed in order to distribute light Restlet modules, which is essential for Android and GWT editions.

On Android devices, instead of the Restlet internal connector, most Restlet developers prefer to use the Apache HTTP Client connector because it's already shipped on the devices. The problem is that currently the org.restlet.jar module is always distributed with the internal connector even if it isn't used.

With the profile mechanism, you'll be able to choose among several versions of the same JAR file such as org.restlet.jar for the regular file and org.restlet-lightclient.jar for a version without the server-side part of the API and the internal HTTP connector.

While we're discussing Restlet editions, let's discuss the next two editions that are considered for the next Restlet versions.

11.2.3 Editions for JavaScript and Dart

Due to the emergence of HTML 5 on all sorts of browser platforms, including mobile ones, the improvement of JavaScript engines such as Chrome V8, Firefox SpiderMonkey, and the emergence of Node.js on the server side, it seems increasingly important to provide the richness of the Restlet API to JavaScript on both client side (browser and Node.js) and server side (Node.js).

There's already the GWT edition of Restlet, which provides a nice solution by compiling Java to JavaScript, but there is room for a lighter and more hypermedia-driven/document-oriented solution, combining AJAX and UI libraries such as jQuery, Ext JS, and YUI, to name a few.

In Restlet Incubator, there's already a prototype of a native JavaScript port working in browsers and Node.js on the client side. The first goal is to complete this port and start distributing it as an additional edition.

Next we plan to develop another edition for the new and promising Dart language. Dart was launched by Google in 2011, positioned at the crossroads of JavaScript and Java, and providing features of both statically and dynamically typed languages with a syntax similar to Java. Thanks to its syntax and its proper OO typing system, it looks like a semiautomated port from the main Java code base is possible, as with the GWT edition. This offers a good transition to the next roadmap item: the planned enhancements to the Restlet Forge.

11.2.4 Restlet Forge

Since version 1.1, the Restlet Framework has been distributed in several editions, leading to the development of an innovative porting mechanism to automate as much as possible the maintenance of the six editions now available in version 2.1.

For this purpose we use code annotations to provide text transformation rules in order to generate a new version of a given class for a specific edition. The following transformations are supported:

- Removing a specific block of code, single instruction, method, or member
- Adding a specific block of code, single instruction, method, or member
- Excluding an entire class

The implementation is *not* based on Java annotations, because they aren't flexible enough for this purpose. The idea is closer to the concept of conditional compilation used in other languages like C. It consists only of simple Java comments such as the following:

- `// [ifdef <list of editions>] <optional keywords>`
- `// [ifndef <list of editions>] <optional keywords>`
- `// [enddef]`

The advantage of using comments is that they can be put everywhere in the source code, and they're preserved even if the source code is automatically formatted by the development environment. (Unfortunately, Eclipse doesn't allow such comments in the import section.)

USING CODE ANNOTATIONS

The `ifdef` and `ifndef` annotations mark the beginning of a block and target a list of editions specified with a simple comma-separated list of edition IDs. Depending on the optional keywords used, they can require an `enddef` marker.

Here is how to indicate that a block of code only applies to the Java EE and Java SE editions:

```
// [ifdef jee,jse]
instructions1;
instructions2;
// [enddef]
```

On the contrary, the following code will be removed from the GWT edition, but will be kept in other editions:

```
// [ifndef gwt]
instructions1;
instructions2;
// [enddef]
```

Now, imagine that the GWT edition requires a set of instructions that must not be compiled in the main source code. This code must be commented in the main source code but uncommented for the GWT edition, using the `uncomment` keyword:

```
// [ifdef gwt] uncomment
// instructions1;
// instructions2;
// [enddef]
```

The `uncomment` keyword applies for both `ifdef` and `ifndef` block markers and handles only single-line comments (comments starting with `//`).

Other available keywords are provided to reduce the amount of markup by getting rid of the `enddef` marker: `member`, `method`, and `instruction`.

These keywords can be used in conjunction with the mandatory `ifdef/ifndef` block markers and the optional `uncomment` keyword. Imagine that you want to add an attribute (including its comment) for a specific edition. Let's use `member`:

```
// [ifdef gwt] member uncomment
/** This is a specific attribute for the GWT edition. */
// private String attribute;
```

And you can remove a whole method only for the GAE and GWT editions:

```
// [ifndef gae,gwt] method
/**
 * Handles a call.
 *
 * @param request
 *          The request to handle.
 * @return The returned response.
 */
public final Response handle(Request request) {
    final Response response = new Response(request);
    handle(request, response);
    return response;
}
```

The `instruction` keyword handles a single instruction line inside a method. The following code shows how to remove a single instruction for any edition except GWT and replace it by another one for GWT:

```
// [ifndef gwt] instruction
this.context = context;
// [ifdef gwt] instruction uncomment
// this.context = (context != null) ? context : new Context();
```

As you can see, this customization mechanism is simple yet powerful and, more important, isn't specific to the Restlet Framework. One goal moving forward is to let you make use of the Restlet Forge for your own multiedition applications. Besides the traditional portability needs not addressed by the JVM, there is also a growing need to support portability across cloud platforms.

DISTRIBUTION WORKFLOW

In addition, the Restlet Forge comes with a built-in mechanism to package the compiled Restlet artifacts and distribute them in various forms such as a zip file, a Windows installer, a Maven repository, and even an Eclipse/p2 update site as illustrated in figure 11.6.

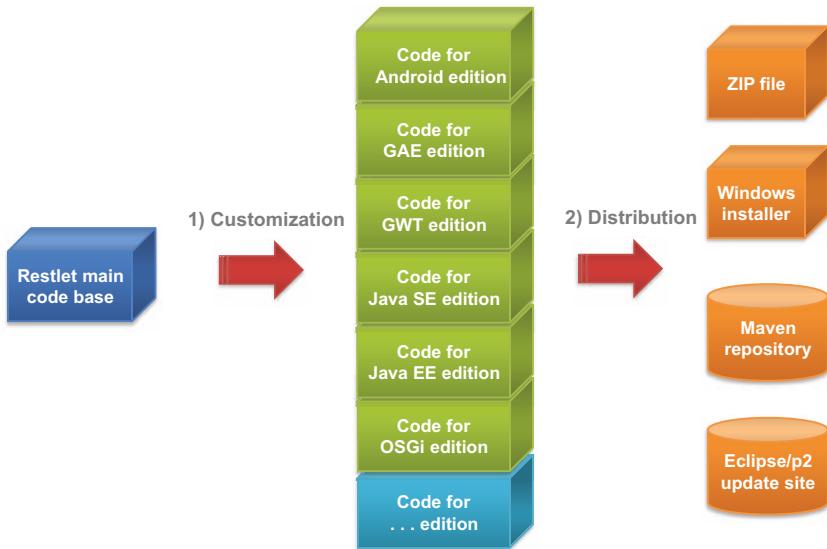


Figure 11.6 Restlet Forge workflow

Internally, the Restlet Forge relies on Ant, FreeMarker, and additional open source libraries and uses XML files to describe editions, modules, and libraries of the project to build and distribute. The remaining work consists in packaging and documenting this piece of code and ensuring that there is no aspect specific to the Restlet Framework remaining in it.

The next section introduces a plan to expand this open source project beyond its Restlet Framework roots into a broader Restlet Platform.

11.2.5 Restlet Platform

After covering the Restlet Framework in depth in this book, including the Restlet API, Engine, and Extensions, we introduced the Restlet Forge and talked about how it will gain importance moving forward as part of a larger Restlet open source project. As illustrated in figure 11.7, there are more pieces planned as part of this Restlet Platform.

Let's start this overview with the Restlet Apps planned right on top of the Restlet Framework.

RESTLET APPS

One of the core design choices of the Restlet Framework is to make it as generic as possible regarding the domain- and business-specific aspects of applications developed with it. In addition, no persistence or presentation technology is preferred over another at the Restlet API level, but optional extensions are provided to facilitate the use of some of them.

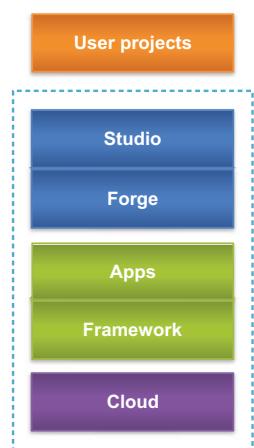


Figure 11.7 Restlet Platform overview

Over the years it appeared that there was space to provide generic and highly reusable Restlet applications for common needs such as searching, administration consoles, reverse proxies, static websites, and so on.

Moving forward, a set of open source Restlet Apps will be developed and distributed as additional `org.restlet.app.<name>` modules, including ready-to-run features, easily customizable and embeddable inside larger Restlet components.

To get started, an `org.restlet.app.search` application will be added to speed up the development of your Restlet projects requiring some search functionality. This will allow you to focus more on your domain-specific features.

This Restlet Search application will use Apache Lucene and the existing Restlet extension for Lucene, supporting the semantic indexing and retrieval of resources based on their multimedia representations. This project will be based on an existing development effort completed by Restlet creators as part of a collaborative R&D effort named HD3D², initiated by several postproduction studios, including Mikros Image.

Those optional Restlet Apps will be developed in the same open source manner as the existing Restlet Framework, using the same licensing options.

RESTLET STUDIO

In addition, in order to facilitate the use of the Restlet Framework within Eclipse IDE, tooling support will be proposed via a Restlet Studio product (see figure 11.8) including a set of wizards to facilitate the creation of Restlet applications, filters, and resources, as well as model-driven tooling to easily generate Restlet applications. It could be installed either as a standalone Eclipse-based application or as a set of plug-ins extending an existing Eclipse installation.

This tool is still under development by the Restlet creators. It's at an advanced prototype stage and will be freely available to Restlet developers in the future.

A special focus is being placed on productivity in order to develop RESTful web APIs, generating the base Restlet server code, client kits for various platforms such as Java, PHP, C#, or iOS, as well as up-to-date web API documentation. Those automated, extensible generation capabilities will rely on a comprehensive web API model expressed using the EMF ECore technology already introduced in this book.

In addition, the core development team is migrating the main Restlet code base to Java 6 and GitHub for version 2.2, relying on Eclipse Juno (4.2) as the main IDE development tool, ensuring that no errors and warnings appear.

RESTLET CLOUD

To facilitate the hosting of Restlet applications, a comprehensive Restlet Cloud hosting solution is being developed. It uses the Restlet Framework for remote deployment and administration of hosted applications and relies on OSGi as a foundation technology to provide both hot deployment and update.

Restlet Cloud will be usable both privately on your own machines and via an online service available at <http://restlet.net>. Even though this service is only part of the Restlet roadmap at this stage, it's already used by Restlet creators in the recently launched APISpark online service.



Figure 11.8 Restlet Studio welcome page

11.2.6 APISpark, the online platform for web APIs

Following the success of REST, a quickly growing number of web APIs is being developed, as illustrated by the statistics from the ProgrammableWeb.com website, a famous web API directory. The growth curve is exponential, and there is no reason why it will stop during the next decade. Just as it's common today to have a website and a blog for an organization (or even an individual), tomorrow it will be as common to have web APIs in order to foster an innovative community, to provide data and service across multiple channels with a consistent user experience, and to communicate across connected devices.

In the face of this massive need for web APIs, we need to lower the barrier of entry to their development and operation, using the power of the Restlet Framework in a simpler way. This is why Restlet creators are developing APISpark, an all-in-one online platform for web APIs, covering the needs of all actors of the web API chain as illustrated in figure 11.9.

APISpark has put the Restlet Platform in action to simplify the creation, hosting, management, and use of web APIs. It powers both the website itself and the generated



Figure 11.9 Main actors in the web API chain

users' APIs. In addition the upcoming Restlet Cloud hosting solution is already being used. Generated web APIs rely only on the open source Restlet Framework and Restlet Apps, creating no lock-in with closed-source software, as illustrated in figure 11.10.

APISpark can be used from a simple web browser by any web developer because it doesn't require any knowledge of Java or of the Restlet Framework. Thanks to its API template approach, web APIs can be created in minutes instead of weeks or months for a traditional IT development project using the Restlet Framework. Figure 11.11 illustrates the home page of a web API created on APISpark.

Once configured and deployed, the web APIs and their underlying data stores (containing both structured entities and flat file folders) are immediately accessible by API users on an apispark.net subdomain or custom domain. They can then be monitored by an API manager person, using features such as API analytics reports illustrated in figure 11.12.

As you can see, APISpark is offering a concrete and easy way to see Restlet in action beyond this book. You can learn more about the service and try it at <http://apispark.com>.

Let's conclude by explaining the place of the Restlet community, how it works, and how you can be part of it.

11.3 Restlet community

Developing the Restlet Framework as an open source project since 2005 has been a great experience for the Restlet team, technically and personally challenging but also rewarding thanks to the numerous exchanges with the Restlet community. This community includes users seeking help in forums, suggesting new features, or reporting bugs, as well as developers contributing patches to fix bugs, enhance an existing feature, or even add and maintain a complete new feature such as a Restlet extension.

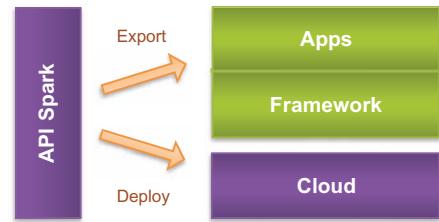


Figure 11.10 APISpark relationships with the Restlet Platform

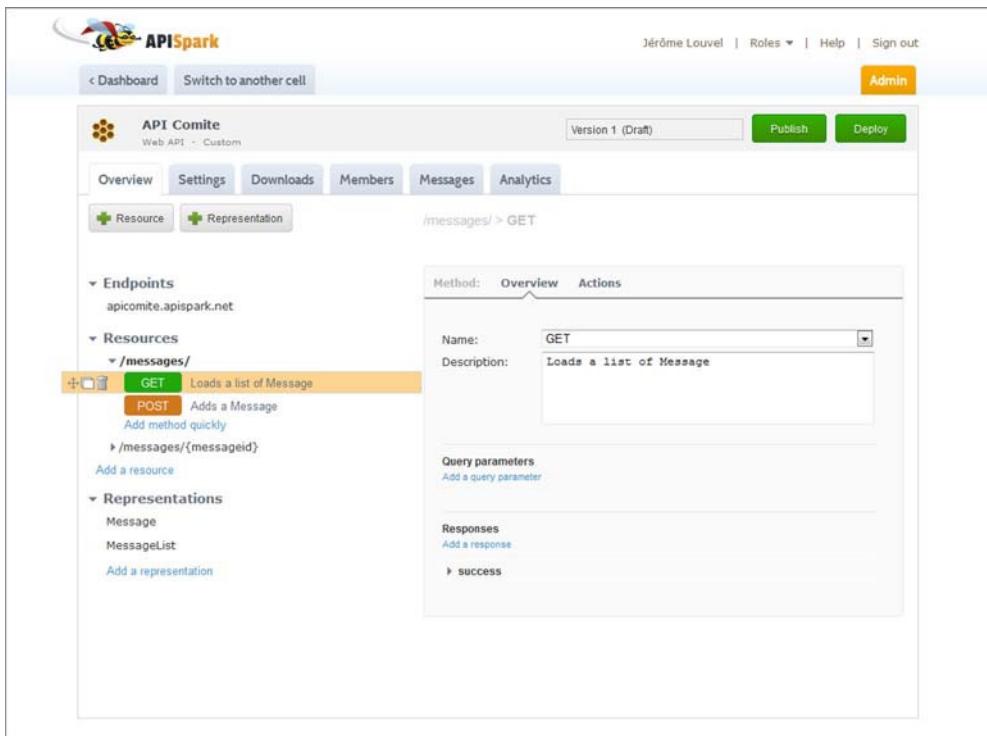


Figure 11.11 Web API overview on APISpark

Over the years we've also seen a growing number of other open source projects developed using the Restlet Framework. The next section reviews some interesting and active ones that we're aware of.

11.3.1 *Third-party projects*

There are several categories of third-party projects, depending on their level of Restlet use. Some are direct extensions of the Framework, whereas others provide higher-level features on top of it.

RESTLET INTEGRATIONS

First, there are two similar integration efforts by popular open source Enterprise Service Buses (ESB)—Apache Camel and Mule ESB—allowing use of the Restlet Framework as part of their event-driven processing chain. MuleSource has developed a special transport for Restlet that lets you use the Restlet API within Mule ESB and provides a Restlet client connector. Camel developers have integrated a pluggable Restlet component to facilitate both consuming and exposing RESTful web APIs.

The Piriti project (<http://code.google.com/p/piriti/>) provides JSON and XML mappers for GWT. It comes with an extension for the Restlet edition for GWT, offering a great way to interact with JSON and XML representations exposed by web APIs not

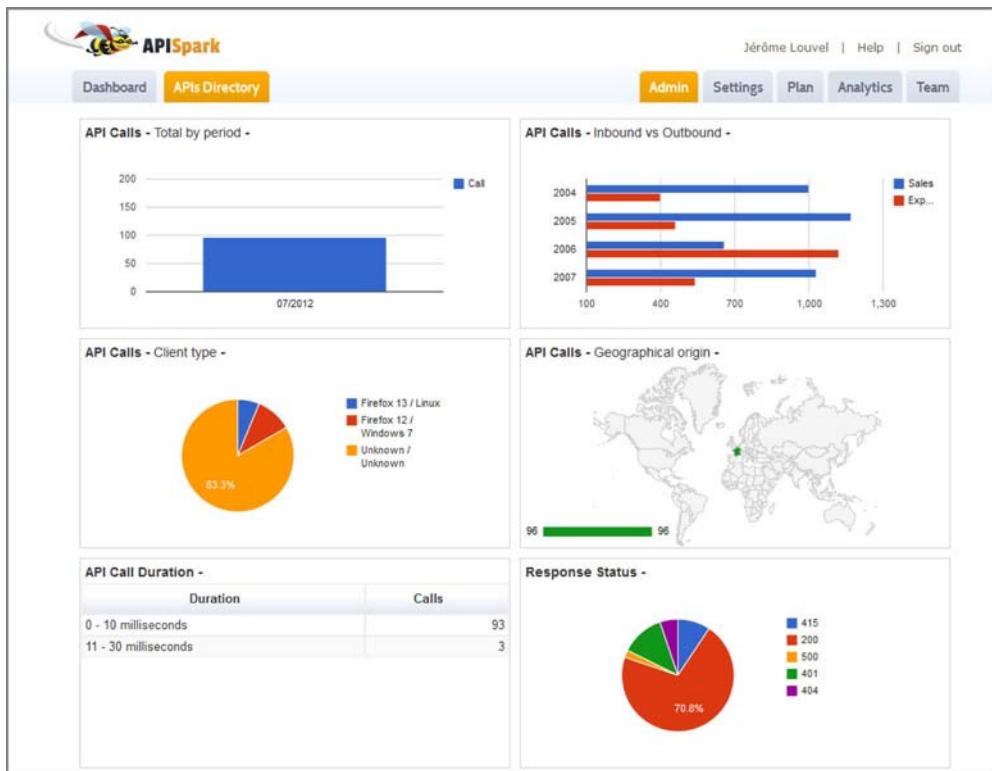


Figure 11.12 Web API analytics on APISpark

developed using Restlet. If your backend is Restlet-driven, though, you'll prefer the default GWT object serialization mechanism provided to reduce the mapping overhead.

RESTLET STACKS

In addition to the previous integrations, some open source projects propose a comprehensive stack based on Restlet to cover a more specific need. Atomogo (<http://code.google.com/p/atomogo/>) provides a complete AtomPub client and server solution, whereas Xeerkat (<http://code.google.com/p/xeerkat/>) offers an embeddable P2P computing framework based on the XMPP protocol (supported by Jabber and Google Talk).

Next we have Kauri (www.kauriproject.org), a Restlet-based Web App Framework including Spring configuration, module wiring, built-in template language, and client-side (JavaScript, jQuery, AJAX) forms framework.

In a similar vein, GoGoEgo (<http://code.google.com/p/gogego/>) offers a RESTful web content management system supporting WebDAV, using OSGi and scripting languages such as JavaScript and including a visual console developed in GWT.

Prudence (<http://threecrickets.com/prudence/>) proposes a scalable RESTful web development platform for the JVM, including advanced support for many scripting

languages. It can also act as an easily configurable deployment container for stand-alone Restlet applications, adding another option to the GAE, Java EE, and OSGi containers available via Restlet editions. On top of Prudence, there's a specific flavor called Savory Framework that uses the JavaScript language and offers ready-to-use drop-in features and integration with presentation frameworks such as ExtJS, Facebook, and Sencha Touch.

For additional projects you can also refer to the Restlet web site, which contains a more comprehensive directory of Restlet-based projects.

11.3.2 Contributing to Restlet

Using an open source project properly is a first, necessary step—reading available documentation and reaching out to users and developers in forums. Moving forward, you'll find many rewarding opportunities to go beyond fulfilling your immediate project needs.

Restlet has the advantage of being directly mapped to REST and HTTP concepts; therefore, the more you learn about Restlet, the more you understand the web and its architectural style. This knowledge and experience will enrich you professionally; even if tomorrow you have to use another technology, you won't have to learn everything again from scratch but instead can map new terms to stable concepts that you already know.

To go beyond this book, you should put what you learn here into action in your own projects. But we also encourage you to get involved more directly with the Restlet Community, depending on your time and areas of interest, as many others have done in the past (www.restlet.org/about/team). This project has been shaped not only by Restlet creators and core developers but also by the many other contributors. As the Restlet Framework expands beyond its initial roots into a more comprehensive Restlet Platform, many innovation opportunities lie ahead.

It's now your turn to apply this new knowledge and realize your web ideas. We hope you enjoyed reading this book as much as we enjoyed writing it, and we wish you a long and pleasant Restlet development experience.

11.4 Summary

In this chapter you learned about HTTP's past, present, and future, including the rewriting of HTTP/1.1 specifications into a more modular form (HTTP/1.1 bis) as well as the prospect of HTTP/2.0 that could take advantage of innovations from the SPDY protocol.

Then you saw the high-level roadmap of the Restlet Framework, including both the evolution of the framework itself and the Restlet API, engine, and extensions, and a broader plan for a Restlet Platform including a reusable Restlet Forge; a set of Restlet Apps, including a first Restlet Search app; a Restlet Studio offering an Eclipse-based IDE for Restlet development; a Restlet Cloud hosting solution, using OSGi; and finally a higher-level online APISpark service to create, host, and manage RESTful web APIs from a simple web browser.

We concluded with the Restlet community and how to contribute back to an open source project, with examples of third-party projects using Restlet in innovative ways.

With this discussion, we have come to the end of the topics planned for this book. We encourage you to keep reading the appendices as they contain valuable guidance and reference materials, such as how to design a RESTful web API with the ROA/D methodology.

This is the end of this book but only the start of your journey with REST and Restlet. We hope you'll realize your next web ideas with Restlet!

appendices

T

hese appendixes contain additional material that should be useful to new and experienced Restlet users.

Appendix A and appendix B provide a detailed overview of the framework as well as instructions to easily install and configure it in your preferred IDE, plus a list of recommended testing tools. Appendix C and appendix D provide reference material independent of Restlet, related to both REST architecture style and the ROA/D methodology for easy RESTful web API design.

Appendix E contains reference material that's practical to have in written form when developing with the Restlet Framework, such as the list of HTTP status codes, HTTP methods and the equivalent Restlet constants or REST concepts, and HTTP headers and the equivalent classes and properties in the Restlet API.

Appendix F offers information to get additional help during your RESTful web API development projects beyond this book.

Contrary to the main chapters, each appendix has been written in a fairly independent way so that you should be able to take advantage of their content while reading the rest of the book, or later on, while developing with Restlet.

appendix A

Overview of the Restlet Framework

This appendix continues the tour of the Restlet Framework started in the first chapter. It begins with a description of Restlet’s core module (which provides the Restlet API and its implementation, also known as the Restlet Engine) and extensions built on top of the core module. We then present the notion of *editions*, which is the way for the Restlet Framework to adapt to various technical contexts while still providing the same high-level API. We conclude this appendix by explaining the versioning scheme used, including the release tags such as *stable*, *testing*, and *unstable*.

A.1 Restlet API

The most important part of the Restlet Framework is its Restlet API. Once you learn it, you can understand most Restlet code and develop both client-side and server-side applications—or make use of many protocols beside HTTP. In contrast with the JAX-RS API, the Servlet API, or the `HttpURLConnection` class, Restlet uses a single uniform API, making it easy to write not only servers, but clients and even web proxies (applications acting both as server and client) in order to build cache systems or web API mashups.

Another benefit of building your applications on top of this Java API is that it doesn’t require any dependency on third-party libraries, besides the standard libraries provided by your target environment. Also, the whole API has been designed to be thread-safe, ensuring consistent and efficient behavior when multiple threads access an object, either at the same time (concurrency) or at a different time (changes visibility).

Therefore, if you’re careful regarding other dependencies, you can build your web applications on top of the Restlet API and ensure that they run equally well in

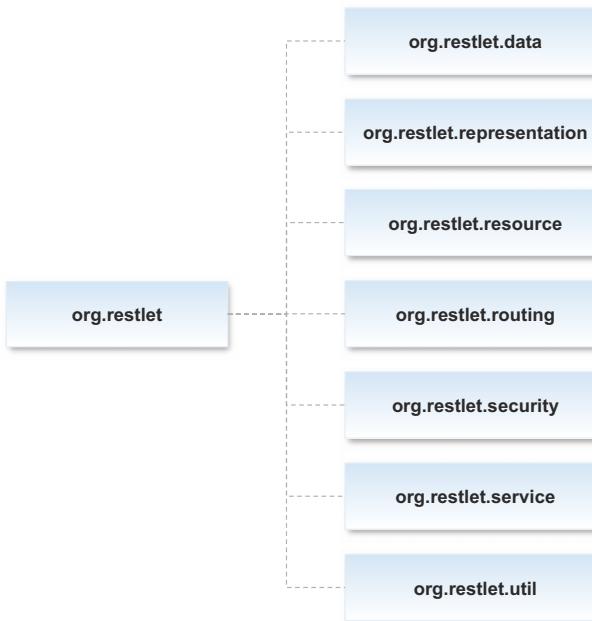


Figure A.1 The Restlet API packages

several environments, such as Java SE, Java EE, OSGi, and Google App Engine for a server-side application, or Java SE, OSGi, Android, and GWT for a client-side application.

By now, you’re probably eager to understand what the Restlet API looks like in detail. Under the root `org.restlet` Java package, you can find seven subpackages as illustrated in figure A.1.

Let’s review the purpose and content of each package.

A.1.1 Root package

The top-level package `org.restlet` contains the most important artifacts. First there’s the `Uniform` interface containing a single `handle(Request, Response)` method that embodies the principle of uniform interface for REST *resources* and the stateless nature of REST communications. Then there’s a hierarchy of core classes inheriting from `Restlet`. Figure A.2 illustrates their organization.

The `Restlet` class implements the `Uniform` interface and provides the equivalent of the widely known `javax.servlet.http.HttpServlet` class. There are important differences, such as the way requests and responses abstract low-level details of HTTP in Restlet where the Servlet API leaves a lot of work in the hands of the developer. A good example is that in Restlet you don’t have to manually parse HTTP headers—all that work is done for you, and the useful information is available as Java classes and properties. You can get more detail in the book’s chapters, including section 4.1.1 and section 7.4.2, as well as an exhaustive mapping list in section 3 of appendix E, titled “Mapping HTTP headers to Restlet properties.”

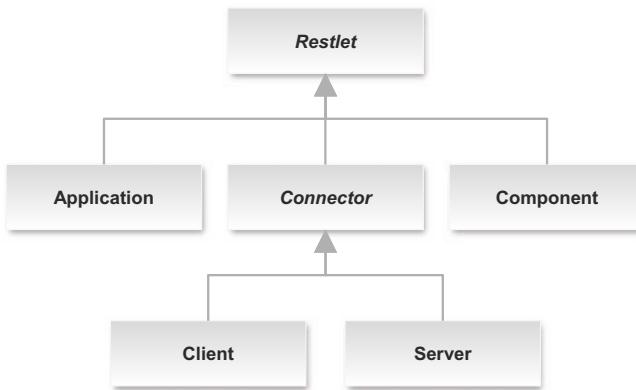


Figure A.2 Hierarchy of classes in the root package

Contrary to the Servlet API, you'll rarely need to directly subclass `Restlet` because the framework offers higher-level abstractions such as the `ServerResource` class. The advantage of the `Restlet` class is that its instances can be accessed concurrently by multiple threads in order to serve several requests at the same time, but this requires a good understanding of Java concurrency programming, equivalent to what is required for Servlet programming.

The `Application` class is commonly extended to develop complete web applications and is extensively covered in chapter 2. You also have the `Component`, `Connector`, `Client`, and `Server` classes corresponding exactly to the REST architecture elements that we introduce in chapter 1. Those classes are necessary when you deploy your applications, as explained in chapter 3.

The `Context` class provides contextual features to a set of `Restlet` instances that are part of the same container, typically a `Component` or an `Application`. You'll be able to store attributes and parameters in this context, or access features provided by the container such as a logger, default credentials, verifiers, or a way to invoke connectors.

The root package contains classes that embody the messages exchanged between components, either requests or responses, as illustrated in figure 1.7 where we discuss REST connectors.

Naturally, the `Restlet` API strictly respects REST and has a hierarchy between its `Message`, `Request`, and `Response` classes, as depicted in figure A.3.

A.1.2 Data package

The `org.restlet.data` package contains classes used to read or write those messages which are composed of several properties, such as the request method (`Method`, `Conditions`), response status (`Status`), URI references (`Reference`), authentication information

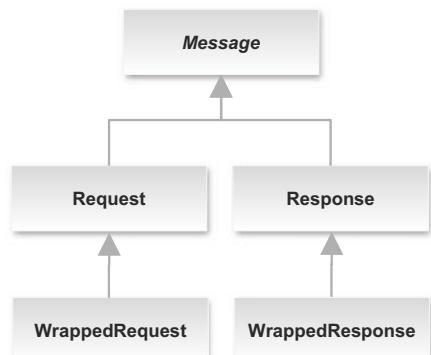


Figure A.3 Message hierarchy

(ChallengeScheme, ChallengeRequest, and ChallengeResponse), agent information (ClientInfo, Preference and ServerInfo), or representation metadata (CharacterSet, Encoding, Language, MediaType).

There are also classes to facilitate the management of URI query parameters and web forms (Form and Parameter classes).

A.1.3 Representation package

Not all data elements were included in the data package! We didn't cover the representations of *resources* that are sent or received in the body of the messages. As previously illustrated in figure 1.4, remember that in REST, clients manipulate *resources* through *representations* of their state.

Naturally, Restlet supports representations via a Representation class and a broad set of subclasses covering the most common types. Figure A.4 shows the top of this hierarchy of classes. The ancestor class Variant describes a representation with enough metadata to support content negotiation—one of the powerful and often forgotten features of HTTP.

The RepresentationInfo class adds a few more properties necessary to support conditional methods, another powerful feature of HTTP. Finally we reach the abstract Representation class, mother of all concrete representations in Restlet.

To produce representations, you need to provide concrete content. This content can be based on character strings (StringRepresentation and AppendableRepresentation), byte streams (StreamRepresentation, InputRepresentation, ByteArrayRepresentation, and OutputRepresentation), character streams (CharacterRepresentation, ReaderRepresentation, WriterRepresentation), or byte channels (ChannelRepresentation, ReadableRepresentation, WritableRepresentation).

There are also special cases for representations based on files (FileRepresentation), serializable Java objects (ObjectRepresentation), and empty ones (EmptyRepresentation). It's also possible to compute the digest (like a checksum) for any representation using a wrapper (DigesterRepresentation).

Other types of representation are available via extension packages, as you'll see later in this section. We cover this topic more extensively in chapter 4.

A.1.4 Resource package

The org.restlet.resource package contains the higher-level classes that you'll use to create client-side resource proxies (ClientResource) or extend to create server-side resource implementations (ServerResource). We used those two classes to write our “Hello World” example in chapter 1, and we cover their use in detail in section 2.5. Figure A.5 shows the hierarchy composed with the parent Resource class (renamed from UniformResource since version 2.1).

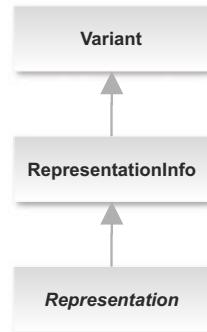


Figure A.4 Representation hierarchy

This package also defines the common annotations used in `ServerResource` subclasses to facilitate the mapping between RESTful methods (typically HTTP methods) and the handling Java methods: `@Get`, `@Put`, `@Delete`, and `@Post`. You'll find the convenient `Directory` class can expose a hierarchy of static files as RESTful resources (covered in section 7.1.3).

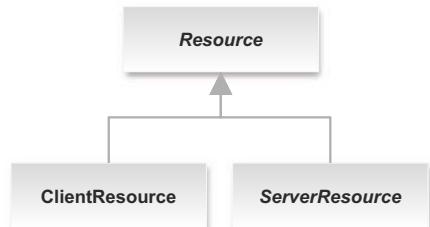


Figure A.5 Resource hierarchy

A.1.5 Routing package

The `org.restlet.routing` package contains all classes directly related to routing. When a request is received, there's a need to route it along a predefined path in order to reach the target `ServerResource` subclass or a `Restlet` instance that will effectively handle it and return a response. This need comes from the fact that your application will likely contain several resources with different URIs or will need to apply specific filters for different resources—for example, for security or validation reasons. Figure A.6 shows how routing is supported via a set of `Restlet` subclasses.

The `Filter` class preprocesses the request, invokes an attached `Restlet`, and post-processes the response. Several specialized filters are provided, like an `Extractor` (looking up data like cookies and query parameters and storing them as request attributes) and a `Validator` (validating an attribute's value against regular expressions). The `Redirector` class supports either client-side redirections (via special response statuses) or server-side redirections (often called *reverse proxying*).

The `Router` class attempts to match the URI in the request against URI patterns defined by your application to determine which `ServerResource` class or `Restlet` instance to target. There's also `VirtualHost`, a specialized router that adds the ability to match virtual hosts by domain name, IP address, and other properties.

A.1.6 Security package

Security is an important topic related to routing that deserves its own package: `org.restlet.security`. Two classic aspects are covered: authentication and authorization via a set of filters, as illustrated in figure A.7.

The authentication step is managed by the `Authenticator` class and the more specialized `ChallengeAuthenticator` subclass, helped by the `Enroler` interface, adding

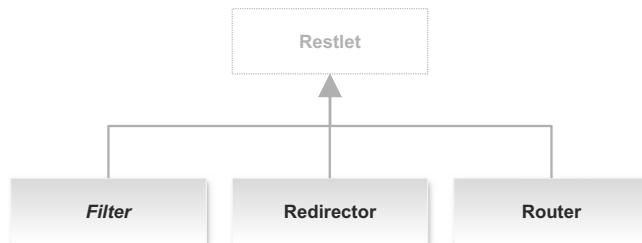


Figure A.6 Routing hierarchy

Role instances to the authenticated User, and the Verifier interface (with SecretVerifier, LocalVerifier, and MapVerifier implementations) to verify the user credentials during challenge authentication.

Then authorization is managed by the Authorizer class and more specialized subclasses (ConfidentialAuthorizer, MethodAuthorizer, and RoleAuthorizer). It can also be checked with a finer precision inside your server resources code via `isInRole()` method calls.

The Restlet security API also offers a clean separation between the unique way each application happens to define roles and the management of users, which is specific to each organization and deployment environment. Therefore, a notion of Realm is provided, exposing a contextual Verifier and Enroler. A default MemoryRealm is provided, supporting complex mappings between a Group and its User instances on one side and Role instances defined by a given application on the other side.

We discuss this dense and important topic at length in chapter 5.

A.1.7 Service package

One of the advantages of Restlet is that your application or component automatically benefits from powerful features via a set of services included in the `org.restlet.service` package. It includes a parent Service class and several subclasses (ConnectorService, ConnegService, ConverterService, DecoderService, EncoderService, LogService, MetadataService, RangeService, StatusService, TaskService, and TunnelService). Each service has a lifecycle with start and stop methods and can add inbound or outbound filters to the container for which they're enabled.

For example, RangeService supports the retrieval of partial representations of resources, without having to change anything in your Restlet code. You can disable this service, like all others, if necessary.

You may guess the purpose of some other services mentioned by their class name, but we describe them individually in the book when we discuss the topic to which they belong. In addition, we explain the common aspects to all services in section 2.3.4.

A.1.8 Util package

As its name implies, `org.restlet.util` contains several utility classes, such as wrapper classes and a string template mechanism (`Template`, `Variable`, and `Resolver`), covered in section 2.4.2 during a discussion of URI templates.

It also contains a generic `Series` class implementing the `java.util.List` interface to manage series of parameters that come in as `name=value` pairs. For example, the `Form` class that we mentioned in the Data package derives from this `Series` class.

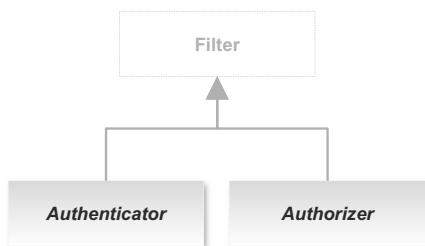


Figure A.7 Security hierarchy

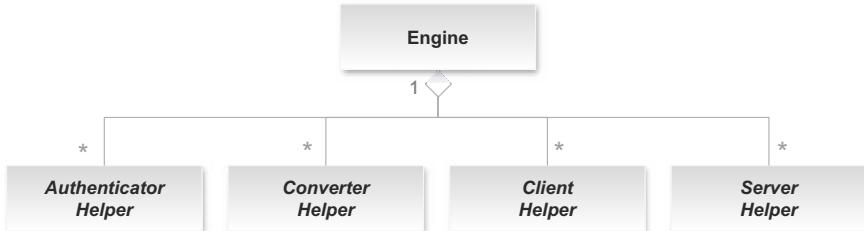


Figure A.8 The Restlet Engine can be extended with pluggable helpers.

A.2 Restlet Engine

Now that you have a good overview of the Restlet API and the features it offers, we suggest you briefly look under the hood and see how the Restlet Engine is designed. We won't cover the engine in detail, as it's rarely necessary for Restlet users to have such a deep level of knowledge of its internals.

Keep in mind that besides classes implementing the Restlet API, built-in HTTP client and server connectors can be used during development. For production environments, we currently recommend using the more robust HTTP connectors (based on Jetty or Simple) that are also provided in the Restlet distribution or to deploy to full-blown Servlet or OSGi containers. We cover the deployment of your applications more extensively in chapter 3.

The root package `org.restlet.engine` contains a singleton of the `Engine` class for the whole JVM. This `Engine` singleton supports all applications, components, and connectors hosted in the current JVM and is a registry managing several sorts of plug-ins that inherit the `Helper` class as illustrated in figure A.8.

`AuthenticatorHelper` instances support authentication schemes such as HTTP Basic, HTTP Digest, SMTP Plain, Amazon Web Services (S3 and all other services), or Microsoft SharedKey. `ConverterHelper` instances perform conversions between Java objects and Restlet representations to support the `ConverterService` mentioned in the description of the `Service` package. `ClientHelper` instances provide client connectors for URI schemes such as HTTP, FILE, or ZIP. `ServerHelper` instances do the same for server connectors.

The fact that new helpers can be registered, either dynamically by adding a specific JAR in the classpath or by programmatically configuring the `Engine` instance, is quite powerful. As we said, you'll rarely need to know more about this topic, so let's move right away to the extensions you can use in your Restlet applications.

A.3 Extensions

Even though the Restlet API has a broad features scope, it attempts to stay as neutral as possible. It doesn't, for example, enforce any particular presentation or persistence technology. Instead, a wide set of Restlet Extensions is offered in order to match the most common use cases. Three categories of extensions are provided:

- Those that add support for standards like Atom/AtomPub, cryptography, HTML forms, JAAS authentication, JAX-RS, JSON, OAuth, OData, OpenID, RDF, SIP, SSL, WADL, and XML.
- Those that provide new connectors such as Apache HTTP Client, JavaMail (SMTP and POP3 clients), JDBC, Jetty (HTTP server), Net (FTP and HTTP client based on `java.net.HttpURLConnection`), Simple Framework (HTTP server), and Servlet integration (HTTP pseudo-server).
- Those that provide integration with third-party services such as template engines like FreeMarker or Apache Velocity, multipart handling via Apache FileUpload, XML serialization via EMF, GWT, Jackson, JAXB, JiBX, and XStream. Extensions also provide integration with technologies such as Lucene, ROME, SLF4J, Spring, and Oracle XDB.

That provides a lot of options for your applications! Keep in mind that each extension is available in a dedicated `org.restlet.ext.<code>` package, where `<code>` is the standard or technology name such as `atom` or `xml`. In the next section, we explain how the Restlet Framework is distributed via several editions.

A.4 *Editions*

The core `org.restlet` module ships the Restlet API and the Restlet Engine. Initially available in standalone mode on top of Java SE, an optional integration with Java EE and Servlet containers has quickly been developed. This integration was easy, as it left the primary Restlet API almost intact and provides a new Servlet extension that bridges the Servlet API and the Restlet API. Then other platforms developed by Google appeared: Google Web Toolkit, Android, Google App Engine. They're similar only in the fact that they support a subset of the Java SE APIs. GWT is also particular in the sense that it proposes only a client API.

Because the Restlet API had to be manually adjusted to work in these different contexts, the idea of several editions of the core module emerged. An *edition* is dedicated to a given platform and provides the core module which defines the Restlet API available, its implementation, and a subset of extensions.

The six editions available in version 2.1 are Java SE, Java EE, GAE, GWT, Android, and OSGi. We include a table listing the extensions available by editions.

A.4.1 *Edition for Java SE*

This edition is aimed for development and deployment of Restlet applications inside a regular JVM. Because it relies on a standard JVM, few restrictions apply: it provides most of the extensions except the ones linked with the Java EE specification, such as the Servlet, Oracle XDB, and the ones specifically linked with the Google App Engine platform. It's compatible with release 1.5 of the Java SE and higher.

A.4.2 *Edition for Java EE*

This edition targets the development and deployment of Restlet applications on Java EE application servers, or more precisely on Servlet containers such as Tomcat and WebLogic. It contains all extensions except the ones for HTTP server connectors such as Simple and Jetty, because the Servlet container already provides that. Also, this edition doesn't contain the GAE extension specific to the GAE edition. It's compatible with Servlet API version 2.5 and above, as well as the Java EE 5.0 specification and above.

A.4.3 *Edition for OSGi environments*

This edition includes the core module and the whole set of extensions adapted to the Open Services Gateway initiative platform (OSGi) except the GAE extension. Contrary to the Java SE and Java EE edition previously discussed, this edition can work either using standalone HTTP servers or the pseudoserver provided by the Servlet extension bridge.

All these modules are plain OSGi bundles including an Activator class when necessary and a proper manifest file.

A.4.4 *Edition for Google App Engine*

This edition gathers the core module and a set of extensions adapted to the Google App Engine (GAE) solution. Due to the restrictions of the GAE platform based on Java 6, with a restricted list of APIs entirely or partly supported, we need to provide an adaptation of Restlet for this environment. Because GAE relies on the Servlet API for server-side aspects, it can be seen as an adaptation of the Java EE edition. The client API is supported via the Net extension, but not the NIO-based implementation of the internal connector, because sockets and threads can't be directly created. This edition is compatible with Google App Engine 1.4.3 and above.

A.4.5 *Edition for Google Web Toolkit*

GWT is a platform dedicated to the development of applications running in a web browser. By default, a custom GWT-RPC mechanism is provided to communicate with a custom Servlet-based server. The port of the Restlet Framework enables you to loosen this tight coupling and helps you expose your server resources on other back-end technologies. The communication is based on the same client API used in an asynchronous way. Also, due to underlying AJAX constraints, NIO channels and BIO streams can't be supported.

Restlet Framework 2.0 added support for annotated Restlet interfaces and automatic bean serialization into its GWT edition in a way that's consistent with other editions. This edition is compatible with GWT 2.2 and above.

A.4.6 Edition for Android

The port of Restlet on Android includes both client and server APIs. You may wonder about the reason to provide another client API addition to the one already shipped with Android, which relies on the Apache HTTP Client library. One reason is that the client-side API of the Restlet Framework provides higher-level features than the default HTTP client library, such as integrated support of conditional requests, content negotiation, and other protocols. It can also rely on the Apache HTTP Client using the related Restlet extension.

You may also wonder why you would have a web server on a mobile phone. Here are some reasons:

- You can test web applications during the development phase without having to consume network access, which might be limited by cost or availability in some areas.
- You can allow third-party applications on other phones or other platforms to remotely access your device. This requires strong security mechanisms provided in part by the Restlet Framework as well as network-level authorizations by the carrier, if any.
- You can run local Android applications that are using the internal web browser and behaving like regular hypermedia applications.

Contrary to other editions, the Android edition can't use Restlet's autodiscovery mechanism for connectors and converters provided as Restlet extensions. This is due to a limitation in the way Android repackages JAR files, leaving out the descriptor files in the META-INF/services/ packages used by the Restlet Framework for autodiscovery. The workaround consist of manually registering those additional connectors and converters in the Restlet engine. Here's an example for the Jackson converter:

```
Engine.getInstance().getRegisteredConverters().add(new JacksonConverter());
```

You may also face another blocking error. The internal HTTP client has been rewritten using the java.nio.package. This may lead, on some Android devices, to this kind of exception: `java.net.SocketException: Bad address family`. In this case, you can turn off the IPv6 preference as follows:

```
System.setProperty("java.net.preferIPv6Addresses", "false");
```

A.4.7 Matrix of extensions per edition

Table A.1 exposes the list of the developed extensions (by their short name, which is the abbreviation of their root package name: `org.restlet.ext.<extension>`) and their availability by edition.

Table A.1 Developed extensions by edition

Extensions	JSE	JEE	OSGi	GAE	GWT	Android	Description
atom	X	X	X	X	X	X	Support for the Atom syndication and the AtomPub (Atom Publication Protocol) standards in the 1.0 version
crypto	X	X	X	X		X	Support for cryptography and HTTP authentication schemes
emf	X	X	X	X			Integration with Eclipse Modeling Framework
fileupload	X	X	X	X			Integration with Apache FileUpload
freemarker	X	X	X	X			Integration with FreeMarker template engine
gae					X		Integration to the Google App Engine UserService for the GAE edition
gwt	X	X	X	X			Server-side integration with GWT for object serialization
html	X	X	X	X		X	Support for the HTML (HyperText Markup Language) standard, particularly multipart file upload
httpclient	X	X	X			X	Integration with Apache Commons HTTP Client
jaas	X	X	X	X		X	Support for JAAS based security
jackson	X	X	X	X		X	Integration with Jackson
javamail	X	X	X				Integration with JavaMail
jaxb	X	X	X	X			Integration with Java XML Binding
jaxrs	X	X	X	X			Implementation of JAX-RS (JSR-311)
jdbc	X	X	X				Integration with Java DataBase Connectivity (JDBC)
jetty	X		X				Integration with Jetty
jibx	X	X	X	X			Integration with JiBX
json	X	X	X	X	X	X	Support for JSON representations
lucene	X	X	X	X			Integration with Apache Lucene, Solr, and Tika subprojects
net	X	X	X	X		X	Integration with java.net.HttpURLConnection class
oauth	X	X	X	X			Support for OAuth HTTP authentication
odata	X	X	X	X		X	Integration with OData services

Table A.1 Developed extensions by edition (continued)

Extensions	JSE	JEE	OSGi	GAE	GWT	Android	Description
openid	X	X	X	X			Support for OpenID authentication
rdf	X	X	X	X		X	Support for the RDF parsing and generation
rome	X	X	X	X			Support for syndicated representations via the ROME library
sdc	X	X	X				Integration with Google Secure Data Connector on the cloudsider
servlet		X	X	X			Integration with Servlet API
simple	X		X				Integration with Simple framework
sip	X	X	X			X	Support for Session Initiation Protocol (SIP)
slf4j	X	X	X				Support for the SLF4J logging bridge
spring	X	X	X	X			Integration with Spring Framework
ssl	X	X	X			X	Utilities to configure SSL support
velocity	X	X	X	X			Integration with Apache Velocity
wadl	X	X	X	X			Support for the WADL specification
xdb		X	X				Integration within OracleJVM via the Oracle XML DB feature
xml	X	X	X	X	X	X	Support for the XML documents
xstream	X	X	X				Integration with XStream

A.5 Restlet versioning

It's important that you understand the Restlet versioning to assess which version is best to use for your projects.

A.5.1 Logical versions

Apart from the classical version numbers used, the Restlet Framework offers three versions, following the naming convention of the Debian Linux project:

- *Stable* is the release recommended for applications in production. The API of this release is frozen, and only bug fixes are made.
- *Testing* is the release recommended for new developments. The community is involved, and contributions of any kind (feedback, bugs, missing features, code, patches, and bug fixes) are welcomed.
- *Unstable* is the release where active development happens. It corresponds to builds of the development trunk passing all unit tests.

The unstable release is generated every eight hours, assuming the code has been updated. The testing and stable releases are tagged in the revision control system (currently a GitHub repository) and are refreshed once the set of changes is important enough.

Following the Git naming, the unstable and testing releases are taken from the master branch, whereas the stable release is built from a numbered branch, such as the 2.1 branch. The unstable release equals the testing version plus a set of enhancements and bugs fixes, but sometimes a regression can be introduced.

A.5.2 Versioning scheme

Restlet versions are composed of a standard triplet: *major.minor.release*. A major version corresponds to deep changes in the Restlet API without a guarantee of incremental compatibility, whereas the minor version denotes lighter changes: deprecated classes or methods are removed, and extensions can also be added or removed.

Release is typically a numeric value. It starts from 0, then is incremented each time a bug-fixing version is released. This value can also be *milestone* (abbreviated *m*) or *release candidate* (abbreviated *rc*) concatenated with an increment (for example: 2.1m5, 2.1rc2, and so on). Milestones are released in the active phase of development where the framework can change freely. The intent of candidate releases is to freeze the features set and the API and only change it in case of design issues or bugs.

Generally speaking, the lifecycle of releases lasts about two years, even if there's a goal to lower it. Version 1.0.0 was released in April 2007; 1.1.0 in October 2008; 2.0.0 in July 2010; and 2.1 is planned for September 2012.

appendix B

Installing the Restlet Framework

The goal of this appendix is to help you install the Restlet Framework and get started with it, from a tooling perspective. We describe several ways to retrieve the framework artifacts, called *distributions*. We guide you through the basic installation and usage steps for major IDEs, showing in detail how to run a first Restlet program.

We also review recommended testing tools and illustrate their basic use with Restlet. This provides an opportunity to focus on the key task of debugging Restlet-based programs.

B.1 Restlet distributions

The Restlet Framework is composed of a core module that depends only on Java SE, plus a set of extensions that generally depend on third-party libraries. Initially, the framework was distributed as a zip containing the Java binaries and source code of all of these artifacts. Then an automated installer for Windows was added, later a Maven repository, and recently an Eclipse update site. This section details these distributions and gives hints about how to use them. We'll begin the tour with the Maven repository.

B.1.1 Maven repository

Maven is a software tool for the management of Java projects and for build automation. Created by Jason van Zyl in 2002, it is hosted by the Apache Software Foundation. Among other features that make this project popular, Maven allows users to keep their installation up-to-date by automatically retrieving dependent JAR files. Some Restlet users were frustrated by having to regularly check for the fresh bits and promoted this solution. It's been decided to provide a public Maven repository

available at <http://maven.restlet.org>. This is updated on a daily basis and stores all Restlet artifacts and third-party libraries that aren't part of the central Maven repository (<http://search.maven.org>) or other public ones.

Once Maven has been installed in your environment, you should declare the Restlet public repository either in the POM configuration file of your project or parent project. Add the following snippet of text in the <repositories> section:

```
<repository>
  <id>maven-restlet</id>
  <name>Public online Restlet repository</name>
  <url>http://maven.restlet.org</url>
</repository>
```

As an alternative, you can declare the repository for all your projects. Go to the installation directory of your Maven copy. Open and edit the conf/settings.xml file and add to the <profiles> section the following code:

```
<profile>
  <id>restlet</id>
  <repositories>
    <repository>
      <id>maven-restlet</id>
      <name>Public online Restlet repository</name>
      <url>http://maven.restlet.org</url>
    </repository>
  </repositories>
</profile>
```

After the </profiles> section add the following:

```
<activeProfiles>
  <activeProfile>restlet</activeProfile>
</activeProfiles>
```

With the introduction of the notion of *edition*, the group identifiers have evolved to match the template org.restlet.<edition>, where <edition> is the code of an edition:

- jse for the Java SE edition
- jee for the Java EE edition
- osgi for the OSGi edition
- gwt for the Google Web Toolkit edition
- gae for the Google App Engine edition
- android for the Android edition

If the artifact is a Restlet extension, the artifact identifiers follow the template org.restlet.ext.<extension>, where <extension> is the name of the extension. Note that the artifact ID is also the name of the root package of the extension.

If the artifact is a third-party library, its artifact identifier follows the template: org.restlet.lib.<jar identifier>, where <jar identifier> is more or less the name of the root package of the third-party library.

Regarding the version number, there are two cases. Assuming that the version is divided into three parts (major, minor, and patch), in case the *patch* part is a numerical value, the Maven version is as follows: major.minor.patch. Otherwise, the release is either a milestone or a release candidate, and the version is respectively as follows: major.minor-M<increment> or major.minor-RC<increment>, where <increment> is a numerical value starting from 0. Additional details on versioning are available in appendix A, section A.5.

Each project based on the Restlet Framework needs to declare at least one dependency: the Restlet core module. According to your needs, you should complete the list of dependencies with the required extensions and connectors. For example, assuming your project is a web server delivering static files, you could use an extension HTTP server connector such as the one based on Simple.

Because your Maven client correctly references the Restlet online repository, open and edit the pom.xml file for your project and add the following lines of text into the <dependencies> section. You should use the most recent version number available at the time you're reading instead of 2.1-RC2, such as 2.1.0:

```
<dependency>
    <groupId>org.restlet.jse</groupId>
    <artifactId>org.restlet</artifactId>
    <version>2.1-RC2</version>
</dependency>
<dependency>
    <groupId>org.restlet.jse</groupId>
    <artifactId>org.restlet.ext.simple</artifactId>
    <version>2.1-RC2</version>
</dependency>
```

The next section introduces a less integrated way to get the several Restlet distributions by using compressed archives files.

B.1.2 Zip files

The zip file is available for all releases and by edition. All are available from www.restlet.org/downloads/.

The zip contains the following directories and files:

- *docs*—The Javadocs of the Restlet API, the Restlet Engine, and the extensions
- *lib*—The JAR files of the Restlet API, the Restlet Engine, the extensions, and their dependencies
- *src*—The source code of the framework and the provided examples
- *changes.txt*—The list of fixed bugs and enhancements by release
- *readme.txt*—Summary of the Restlet project
- *copyright.txt*, *license.txt*, *trademarks.txt*—Some legal materials

Once downloaded, decompress the zip file into the desired directory. When developing, you'll need to have a closer look at the lib directory in order to pick up the

needed JAR files to add to your classpath (they aren't all needed at the same time), and the docs directory, in order to read the Javadocs in your web browser.

On Windows-based computers, an executable installer enables you to automate the installation of a distribution.

B.1.3 Windows Installer

The Windows Installer is available for all releases and all editions at www.restlet.org/downloads/. This distribution is more or less the self-extracting version of the zip file based on the NSIS installation software, with the addition of an easy way to uninstall it.

The installation process is simple and requires your attention only in two places: to specify the installation directory and to accept the license. Figures B.1–B.5 are snapshots of the installation process. Double-clicking the executable file bring up the screen in figure B.1.

Click the Next button, and you're asked to read and accept the license (figure B.2).

After you agree to the license, enter the root directory to put the files into and click Next (figure B.3).

The next step is to confirm the Start Menu folder and launch the installation by clicking Install (figure B.4).

The installation will take a few minutes. Validate the last installation step (figure B.5).

The next section is dedicated to Eclipse users interested in the development of Eclipse plug-ins or OSGi-based applications. An Eclipse p2 repository provides the Restlet modules as OSGi bundles ready to be installed in the Eclipse IDE.

B.1.4 Eclipse update site

This update site is only available for the Restlet edition for OSGi environments since version 2.1. It allows the retrieval of Restlet modules as plain OSGi bundles (including their dependencies) directly from the Eclipse IDE. They're gathered by type of



Figure B.1 Front page of the installation process

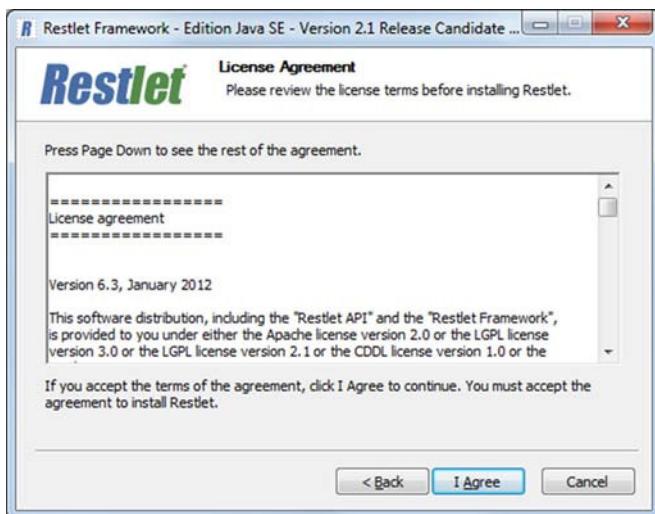


Figure B.2 Read and accept the license.

module: Core module, support of standards, pluggable connectors, and integrations with third-party libraries.

Regular Eclipse IDE development

This Eclipse update site distribution is only useful for pure OSGi developments such as in an Eclipse RCP or Eclipse RT environment. For regular Eclipse IDE use of the Restlet Framework, such as for Java SE or Java EE, refer to section B.2.1.

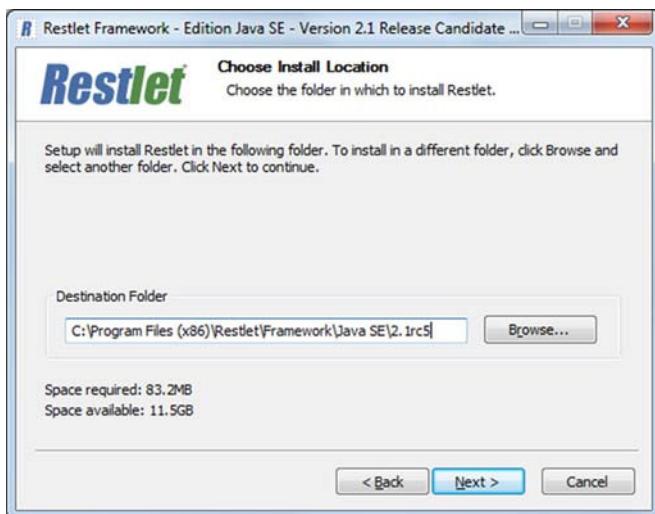


Figure B.3 Specify the installation directory.

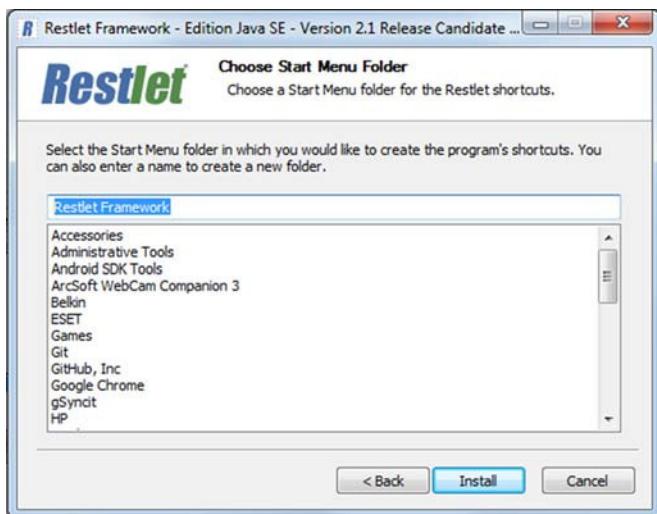


Figure B.4 Choose the Start Menu folder.

The URL of the update site is based on the following pattern: http://p2.restlet.org/<major_version>.<minor_version>. It provides all tagged releases (such as current snapshot, or milestones, releases candidates, and so on) and allows for easy upgrade.

The next screenshots illustrate how to install the bundles into your Eclipse IDE. Choose Help > Install New Software. Enter the URL of the desired repository in the upper field, and click Add (figure B.6).

Make your selection from the list of available items, which are listed by category (Restlet Core, Pluggable connectors, and so forth). After you make your selection, click Finish (figure B.7).

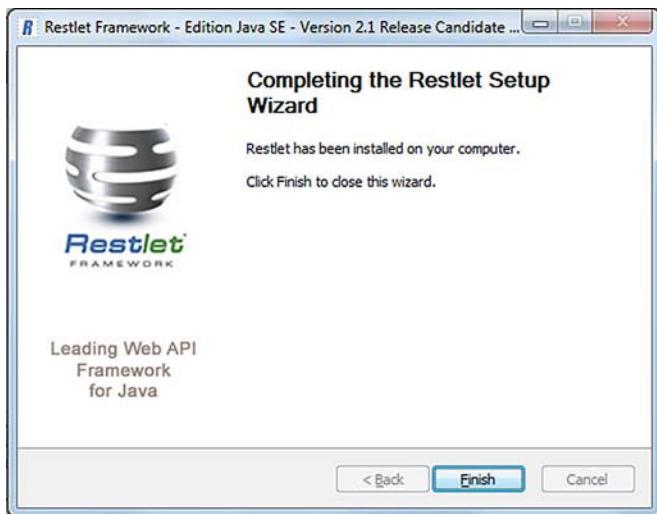


Figure B.5 End of the installation process

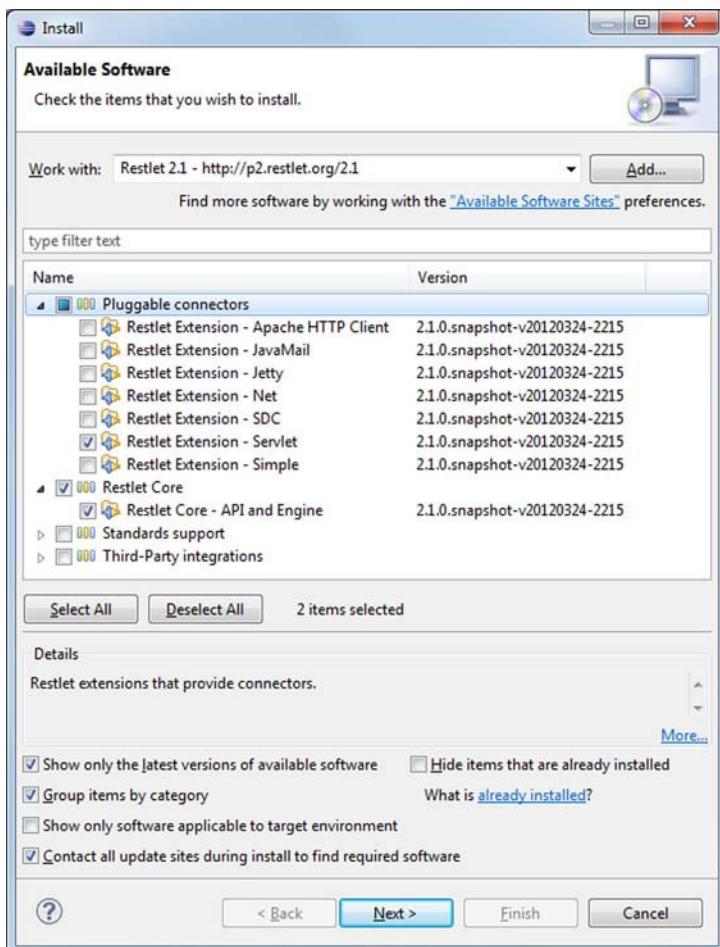


Figure B.6 List of available items

After a moment, which includes the detection of dependencies, you'll be warned that you're about to install unsigned content. Stick to your choice and click OK (figure B.8).

The installation process goes on, and after a moment you'll be asked to restart Eclipse in order to finalize the installation (figure B.9).

The next section deals with the installation of the Restlet Framework into usual Integrated Development Environments (IDEs), such as Eclipse, NetBeans, and IntelliJ.

B.2 Setting up your IDE

The aim of this section is to give Java beginners a short review of the steps required to code a simple program based on the Restlet Framework with three major IDEs: Eclipse (version 3.6, aka Helios), NetBeans (version 7.0.1), and IntelliJ IDEA (version 10.5.1).

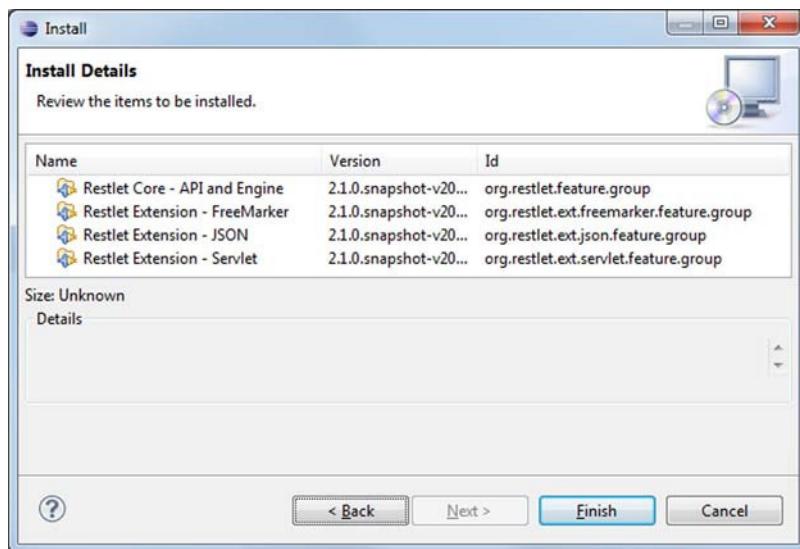


Figure B.7 List of selected items

This guide will help beginners go through the basic tasks needed to create a project and update the list of dependencies.

Although the three IDEs propose unsurprisingly distinct interfaces, note that they share at least some basic concepts that make it easy to go from one IDE to another if needed. This guide is illustrated by a simple test combining a simple server Restlet that serves the “hello, world” text and a simple ClientResource that gets it and prints it to the console. It depends only on the core module of the Restlet: org.restlet.jar.



Figure B.8 Warning: the content is unsigned



Figure B.9 End of the installation

The whole test code is located in a single Java class called `HelloWorld`, whose package name is `hello`. In the following listing, you'll find the content of this class.

Listing B.1 Sample HelloWord program

```

package hello;

import org.restlet.Request;
import org.restlet.Response;
import org.restlet.Restlet;
import org.restlet.Server;
import org.restlet.data.MediaType;
import org.restlet.data.Protocol;
import org.restlet.resource.ClientResource;

public class HelloWorld {

    public static void main(String[] args) throws Exception {
        Server server = new Server(Protocol.HTTP, 8182, new Restlet() { ←
            @Override
            public void handle(Request request, Response response) {
                response.setEntity("hello, world", MediaType.TEXT_PLAIN);
            }
        });
        server.start(); ← Start server

        new ClientResource ("http://localhost:8182").get(). ←
        write(System.out);
        server.stop(); ← Retrieve message
    } ← from server
}

```

The remainder of this section assumes that the Restlet Framework has already been installed via the Windows Installer or the zip distribution so that the archives are available via the file system. Let's start with the Eclipse IDE.

B.2.1 Eclipse

Eclipse is an open source project developed by the Eclipse community (www.eclipse.org). For the purpose of this book, we present a basic configuration of a Java application using Eclipse 3.6, known as Helios. Once Eclipse is started, go to the Java perspective and choose File > New > Java Project (figure B.10).

The first page of the wizard is where you enter the name of the project. Call it `test-Restlet` and leave the other parameters as their default values. Because the Next button leads you to define specific or advanced parameters, you'll ignore it and finish the process by clicking Finish (figure B.11).

Once the project is created, you need to add the dependency to the Restlet Framework, which, in that simple case, consists of a single Java archive: the code module `org.restlet.jar`. Open the contextual menu of the project by right-clicking the name of the project. Follow the Build Path submenu (figure B.12).

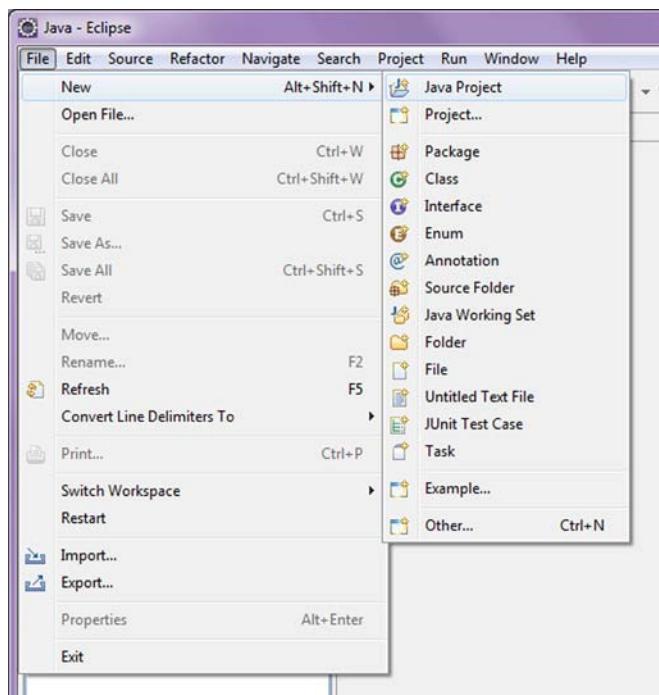


Figure B.10 Create a new project.

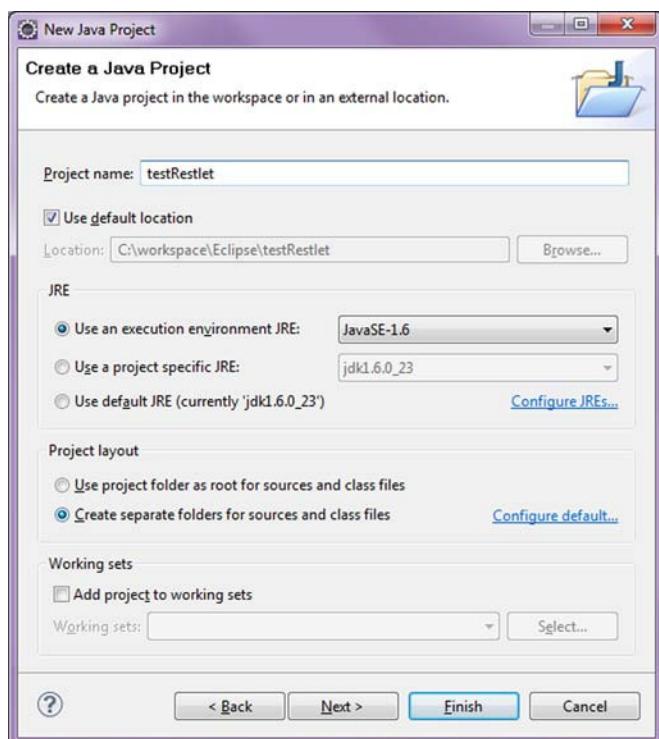


Figure B.11 Enter the project name.

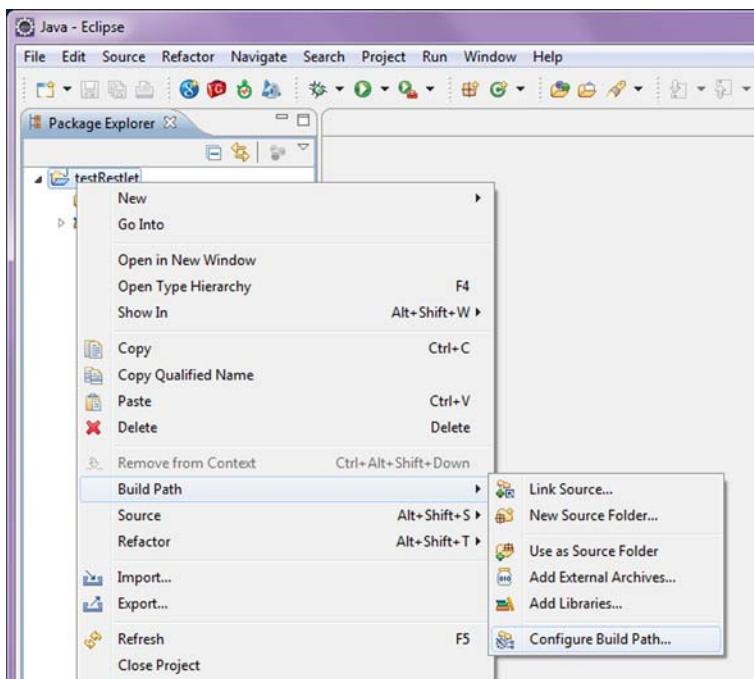


Figure B.12 Open the contextual menu.

Click Configure Build Path, and make sure the Libraries tab is selected. You should see the window shown in figure B.13.

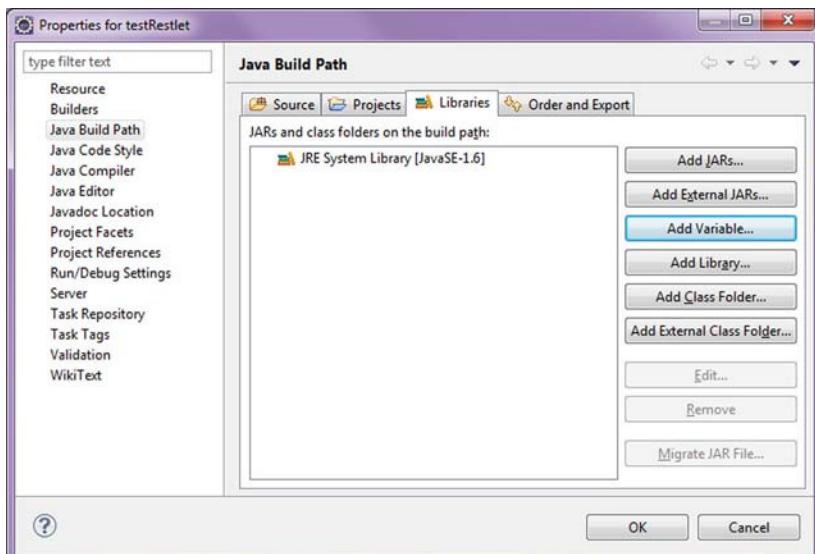


Figure B.13 Configure the build path.

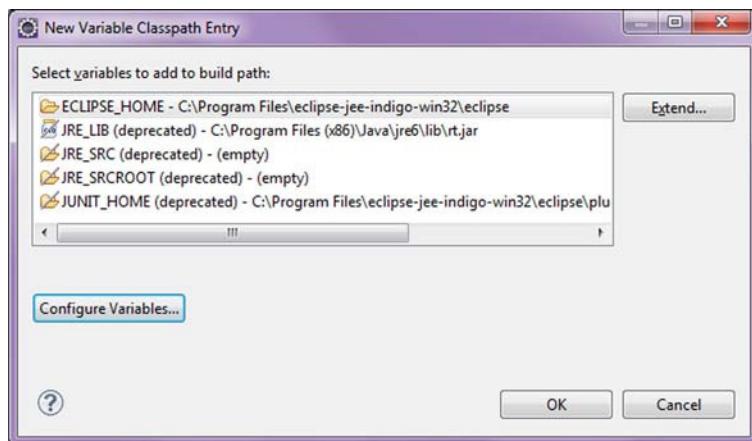


Figure B.14 Creating and extending a variable, first page

A direct way to add an archive is to click the Add External JARs button and browse your disk to select the desired archive. One drawback with this is that your project now depends on your filesystem. Eclipse offers two smarter ways to achieve this. The first is based on the declaration of variables which can be extended. The other is based on the declaration of the user library—refer to the documentation of the Eclipse project for more detail on this feature. In this case, you’ll create a RESTLET_HOME variable pointing to the root directory that contains Restlet libraries, and then this variable will be extended to list real Java archives. To achieve these two tasks, click the Add Variable button. You’ll see the dialog in figure B.14.

The first step is to create the variable. Click the Configure Variables button and then New.

In this case, the variable is called RESTLET_HOME and points to a local directory. You can either browse your file system using the Folder menu or enter the directory manually. Here, you use the root directory of Restlet previously installed using the Windows Installer (figure B.15).

Once the variable has been created, you’re redirected to the window that lists the available variables (figure B.16).

Because you’ve only defined a directory, you need to extend it. Choose the RESTLET_HOME variable and click the Extend button to reveal the extensions. This is a simple browser that will help you to choose the single entry used to extend the variable. In this case, the entry org.restlet.jar is located under the lib directory (figure B.17).

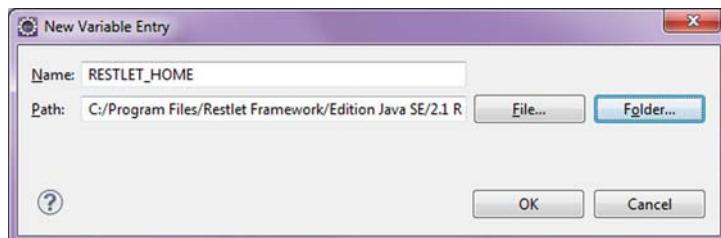


Figure B.15 Creating a variable

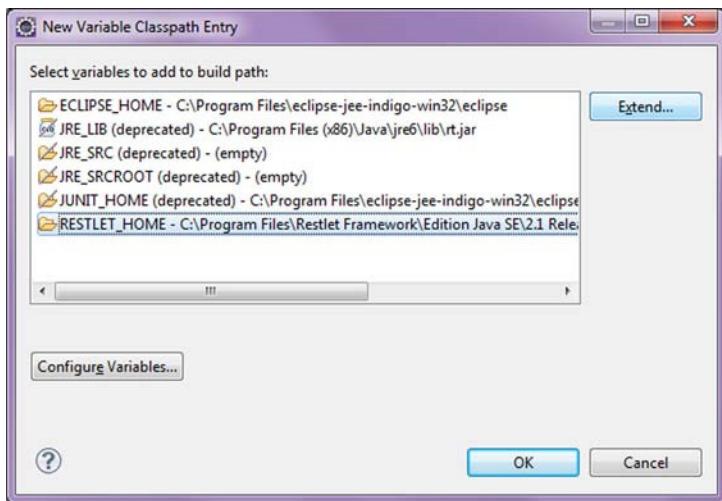


Figure B.16 Listing variables

Once the JAR file is selected, click OK and return to the Java Build Path dialog to see the new entry added to it (figure B.18).

As a side note, classpath variables and user libraries can be configured outside a project in the preferences of Eclipse. They're both available under the submenu Java > Build Path (figure B.19).

The project is created and its build path configured. Create a new package called `hello` and a new class called `HelloWorld`, and copy and paste the code specified in listing B.1. Figure B.20 gives a view of what you should see in your Eclipse session. Display the contextual menu by right-clicking the class. It will allow you to run the main method; choose Run As > Java Application.

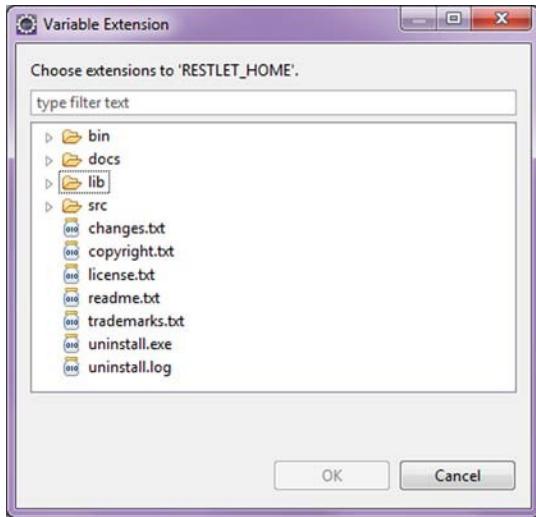


Figure B.17 Locating the entry to extend the RESTLET_HOME variable

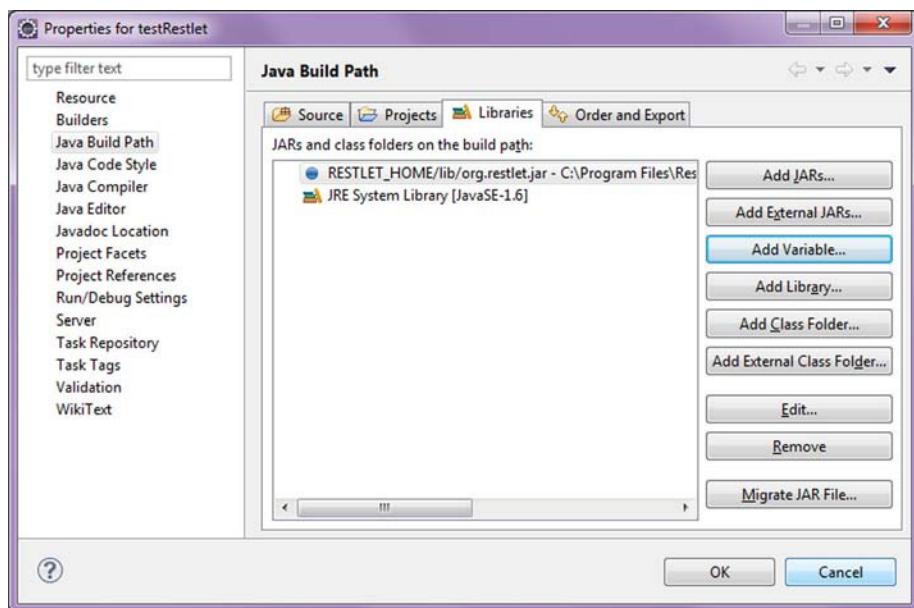


Figure B.18 Extending variables

Figure B.21 illustrates the result of the code once it has run. You should see in the Console view some log trace in red and the “hello, world” string printed in black.

Note that Eclipse is also integrated with Maven using the m2 plug-in. In addition, as seen in the previous section, Eclipse provides its own distribution system known as *update sites*. In the next section, you’ll repeat the same tasks using another open source IDE called NetBeans.

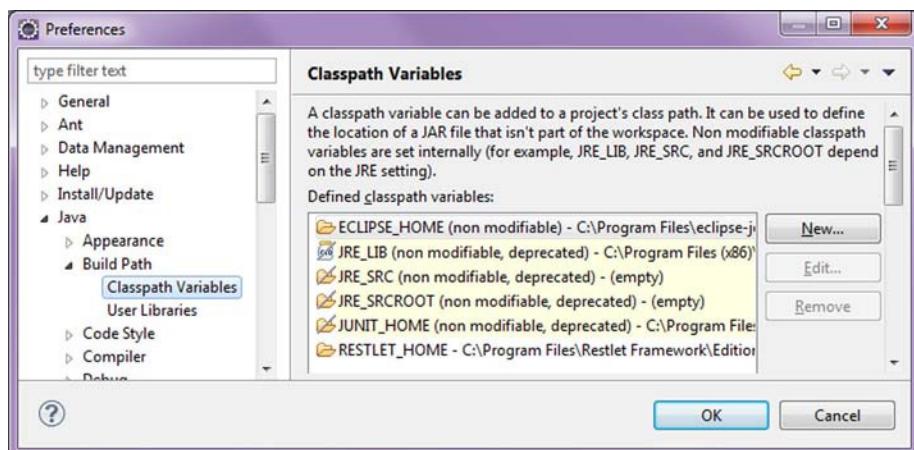


Figure B.19 Managing variables and user libraries in Eclipse

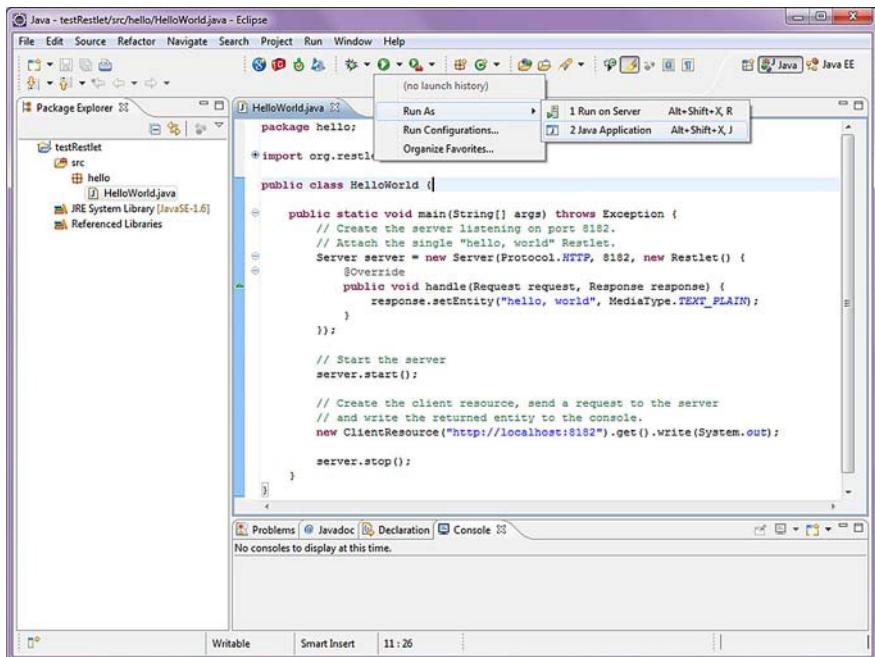


Figure B.20 Run the main class.

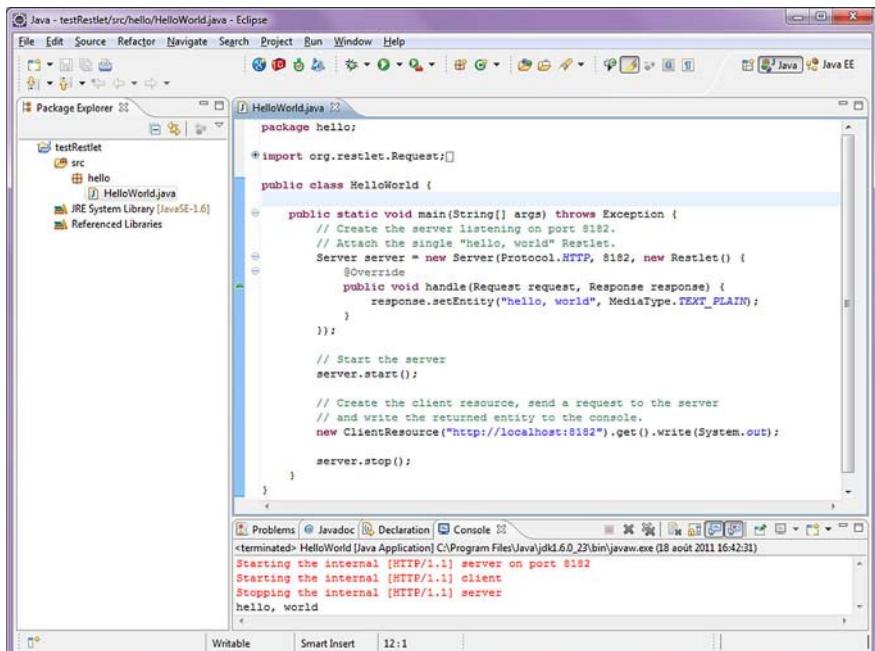


Figure B.21 The Console view

B.2.2 NetBeans

NetBeans is the development environment provided by Oracle (www.NetBeans.org). This section illustrates the basics steps of creating a simple Java project and running it using NetBeans version 7.0.1.

Let's begin with the creation of a brand-new project. Choose File > New project (figure B.22).

Choose Java Application in the first step of the creation process (figure B.23) and click Next.

On the second pane, enter the name of the project; the location will be built automatically (figure B.24).

Once the project is created, it's time to configure the dependency to the Restlet Framework (the single core module). Right-click the Libraries item (figure B.25).

You can directly browse your local disks until you find the org.restlet.jar file using the Add JAR/Folder option. A better way is to first configure a container of Java archives for the whole workspace. This will ease the configuration of the next projects and prevent your project from depending directly on your filesystem. To do so, NetBeans proposes the concept of the library. Select the Add Library option and click Create. Enter the name of the library—for example, RestletLibrary2.1—and choose the Class Libraries type (figure B.26).

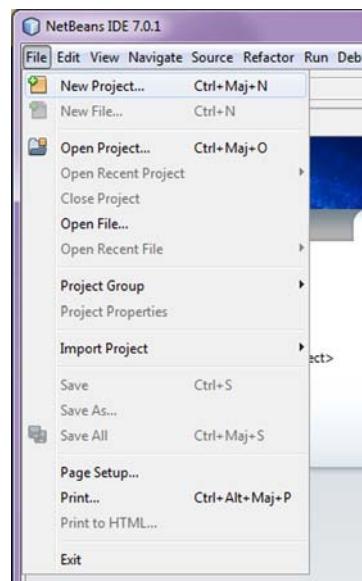


Figure B.22 Create a new project.

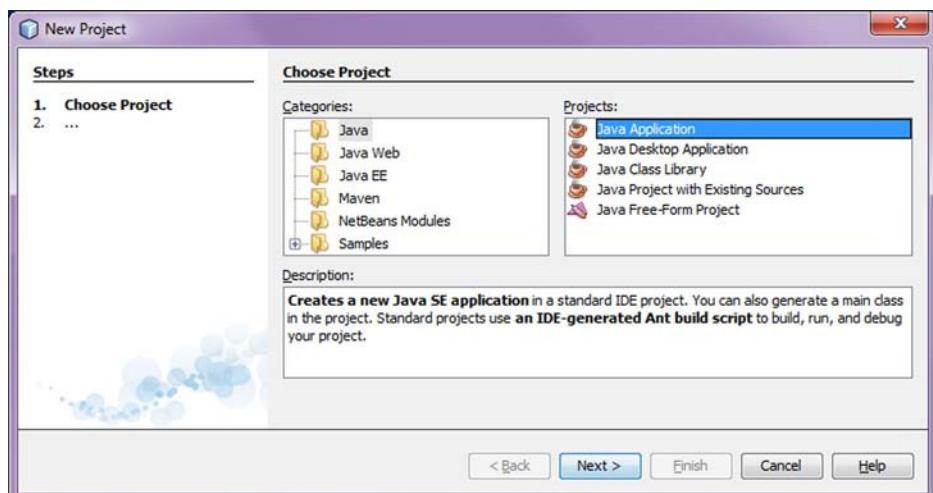


Figure B.23 Create a new Java project.

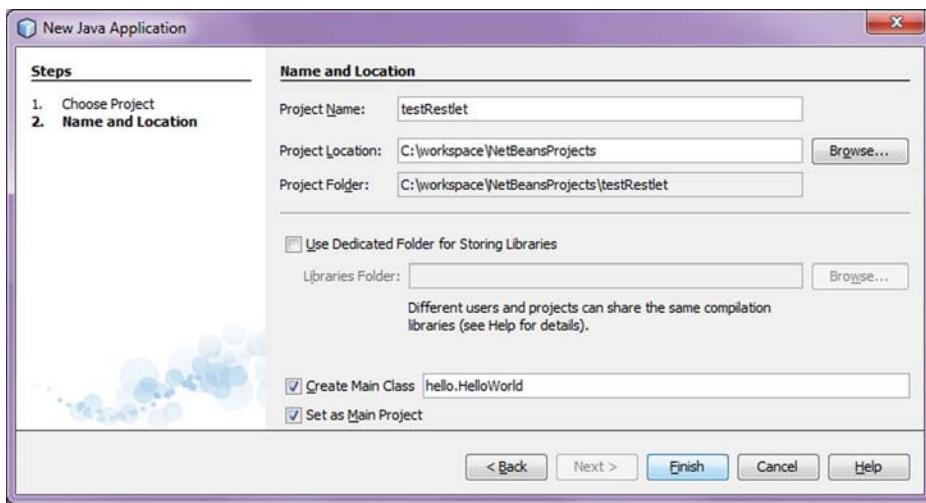


Figure B.24 Choose the project name and location.

The next step consists of gathering the list of Java archives that will be part of this library. Use the Add JAR/Folder button to pick each JAR file that you want to use. You can pick them one by one (figure B.27).

At the end of the process (after a few additional clicks), you can select this library for your project (figure B.28).

NOTE You can manage the libraries of your NetBeans workspace using the Tools/Libraries global menu.

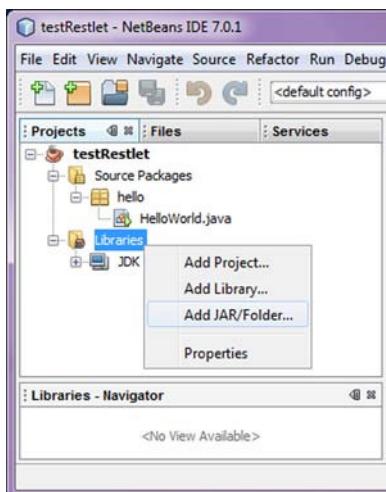


Figure B.25 Update dependencies.

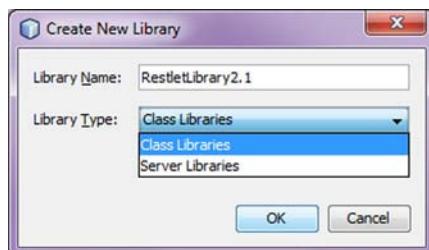


Figure B.26 Add a new global library.

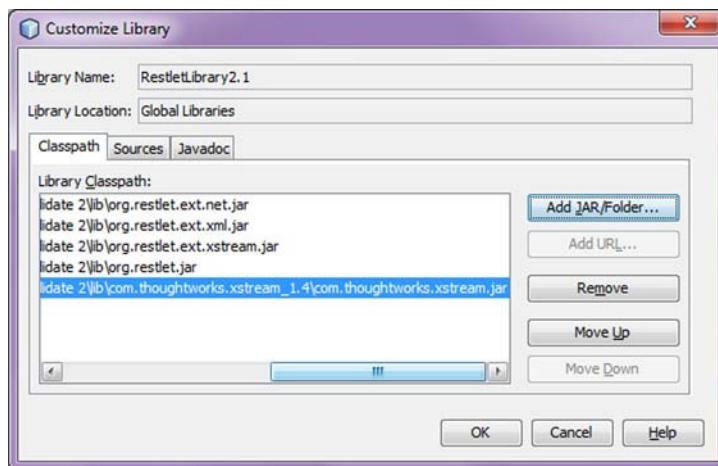


Figure B.27 Complete the library.

You're done with the project creation and configuration. Create a new package called `hello` and a new class called `HelloWorld`, and copy and paste the code from listing B.1.

Figure B.29 gives a view of what you should see in your NetBeans session. Display the contextual menu by right-clicking the class. It will allow you to run the main method by choosing the Run File option.

Figure B.30 illustrates the result of the execution of the `main` method. The Output view contains the log traces in red and the “hello, world” string in black.

It's time to end this tour of development environments by introducing a popular commercial IDE called IntelliJ IDEA.

B.2.3 IntelliJ IDEA

As stated by the originators of IntelliJ IDEA (www.jetbrains.com/idea/): “IntelliJ IDEA is an intelligent Java IDE intensely focused on developer productivity. Our code editor is consistently called the best in the industry, and we support all the major languages and technologies with your productivity and teamwork in mind.”

If you want to try it, download the community release—which is free but supports only the Java SE and not Java EE. Unsurprisingly, the story begins by choosing File > New Project (figure B.31).

Enter the name, and choose Java Module as the project type. In the next panel (figure B.32), choose Create Project from Scratch and click Next.

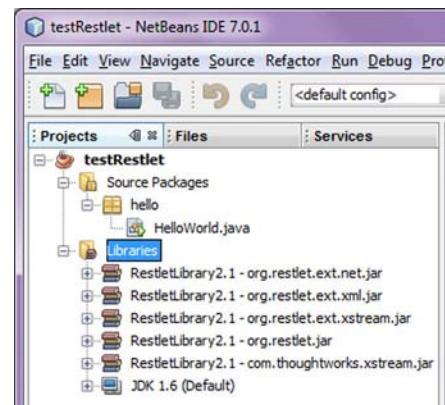


Figure B.28 Add the library dependence to the project.

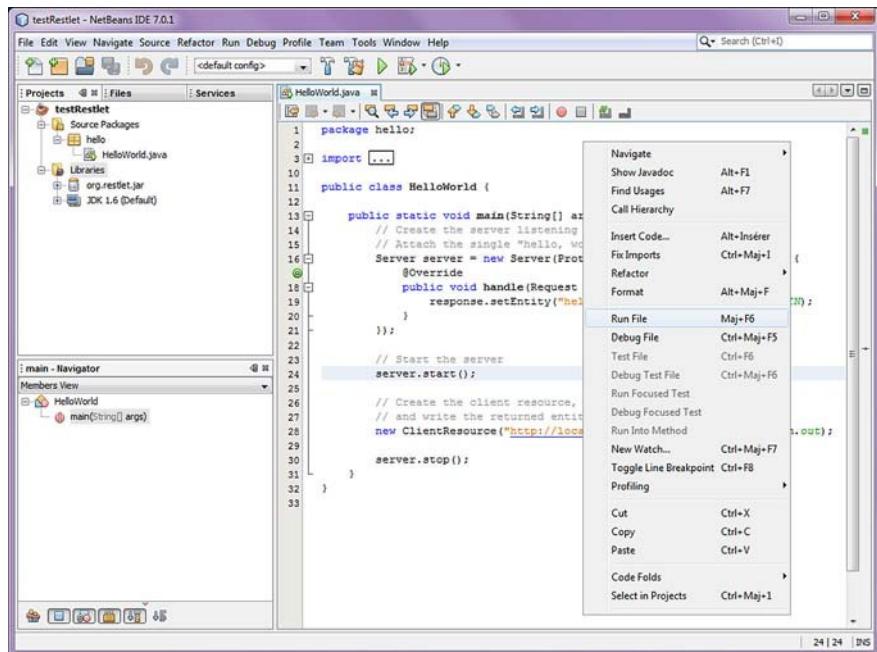


Figure B.29 Run the main class.

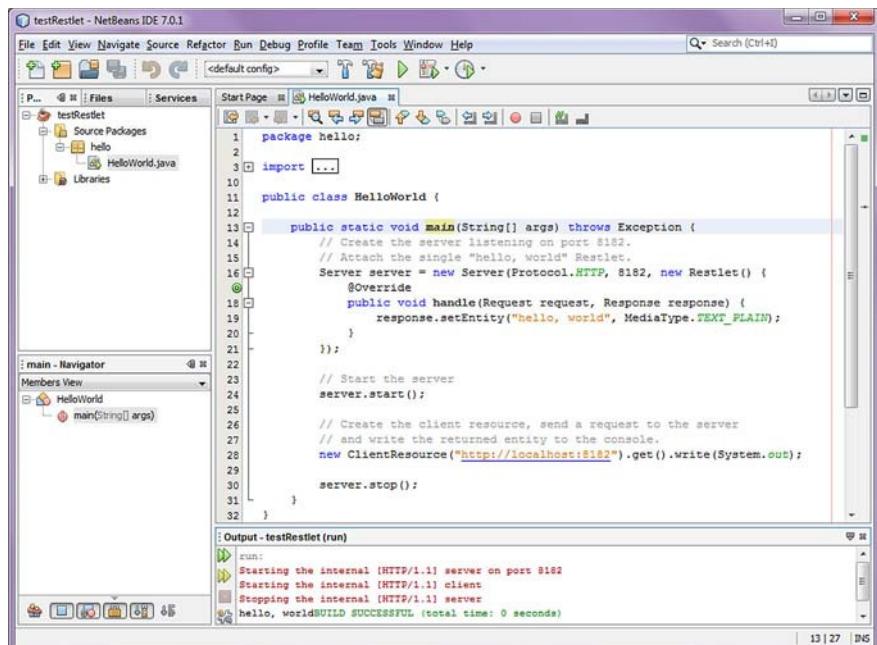


Figure B.30 The Output view

Figure B.33 shows you the window where you enter the name of the project.

Leave the default values of the source directory, and click Next (figure B.34).

Leave the default JDK unless you have a special need (figure B.35).

The dialog in figure B.36 ends the project creation process. Click Finish.

Once the project is created, you configure its dependency with only the core module of the Restlet Framework. Choose File > Project Structure (figure B.37).

In the left panel, choose Modules. In the right panel, select the Dependencies tab and click the Add button, using the Single-Entry Module Library option (or “JARs or directories” in version 11). See figure B.38.

You can explore the filesystem to look for the org.restlet.jar file. Once found, select it. The project is then created and configured. Create a new package called hello and a new class called `HelloWorld`, and copy and paste the code from listing B.1. Figure B.39 shows what you should see in your IntelliJ session. Display the contextual menu by right-clicking the class. You can run the main method by choosing the Run `HelloWorld.main()` option (figure B.39).

Figure B.40 illustrates the result of the execution of the `main` method. The Output view contains the log traces in red and the “hello, world” string in black.

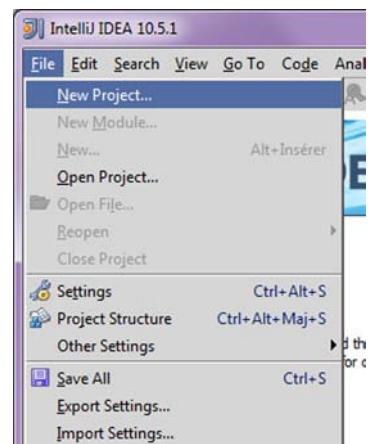


Figure B.31 Choose File > New Project.

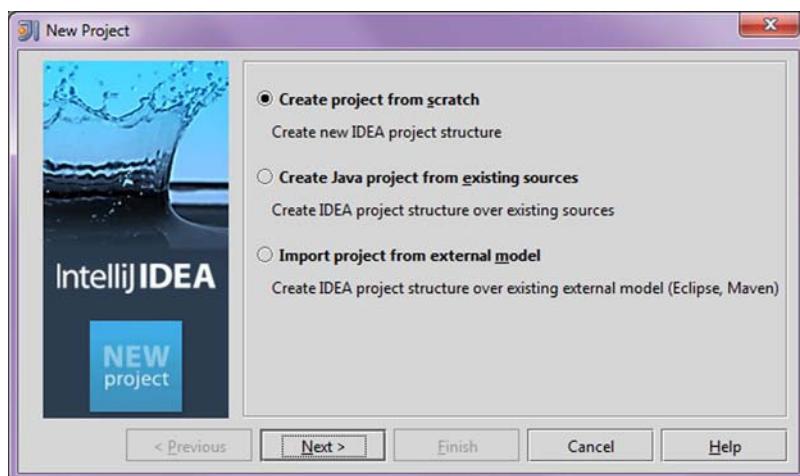


Figure B.32 Create a new project.

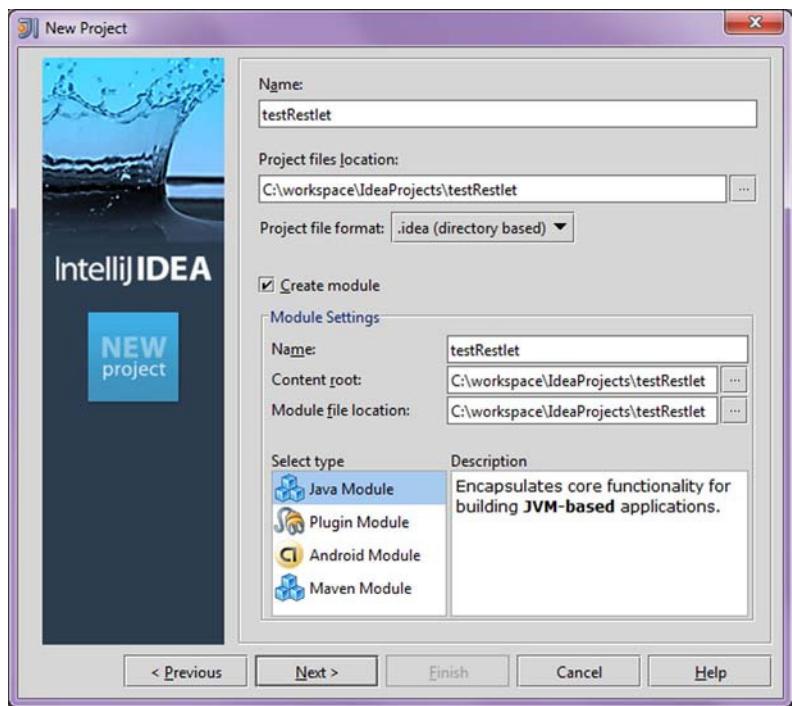


Figure B.33 Specify the project name, location, and type.

This concludes the steps for creating, configuring, and running a Restlet project inside IDEs. But this isn't the end of the appendix. We'll next discuss a less friendly way to develop a Restlet application, using only the command line.

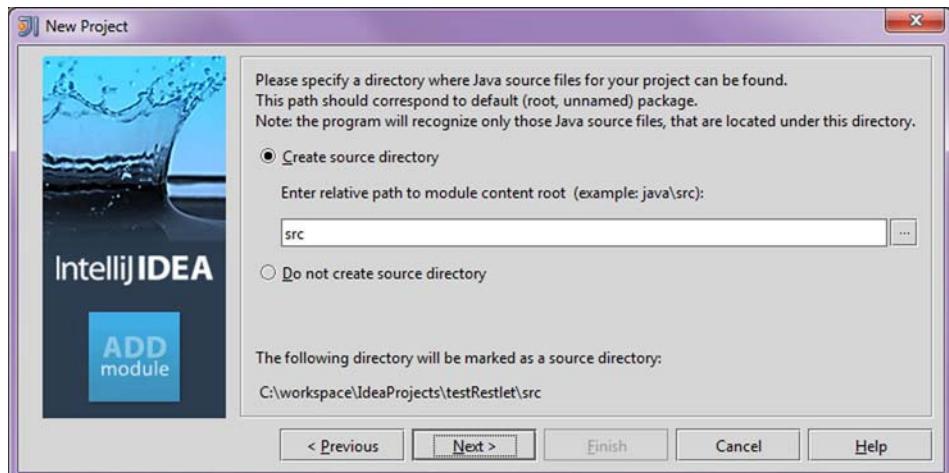


Figure B.34 Specify the source directory.

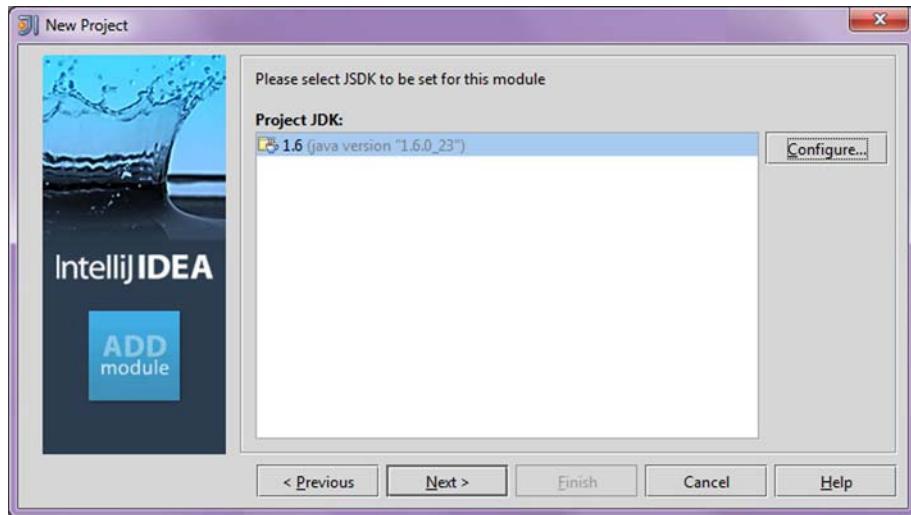


Figure B.35 Specify the JDK.

B.2.4 Command line

Let's go back to basics, relying only on the simple tools provided by the JDK: `javac` for compiling your source code and `java` to run the compiled code, plus a simple text editor. Once the content of the code has been written (in this case, the sample Java class `HelloWorld.java`), you can compile and launch your code with only two command lines, as illustrated in figure B.41, assuming that your `org.restlet.jar` file is available in the current directory.

Note that you have to correctly and cautiously set the classpath for each command via the `-cp` parameter. The classpath should contain at least the core module of the Restlet Framework.

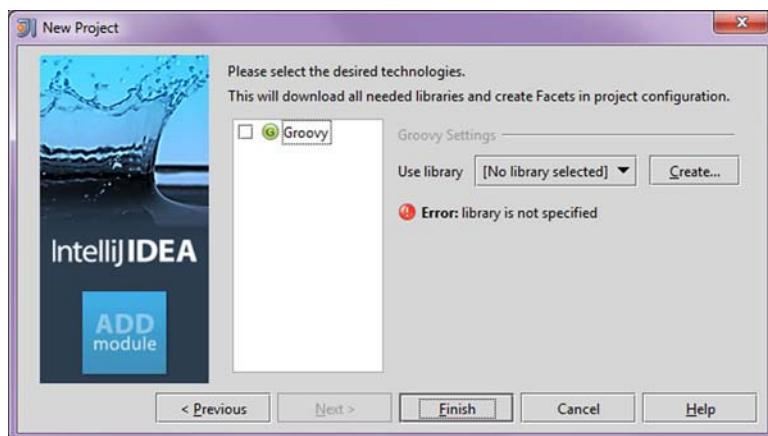


Figure B.36
Final step

B.3 Suggested testing tools

This section reviews a set of common tools used for testing web APIs covering unit tests (JUnit, TestNG) and integration tests (cURL, RESTClient, ClientResource). We end with advice on debugging problems you may face during Restlet development.

B.3.1 Unit testing

This section illustrates the testing features of the Restlet Framework. The main one is that you can programmatically invoke all the API classes (Component, Application, ServerResource, and so on) used during the development of your application, without going through regular network layers. This operation consists of creating the desired instance of the Request class with all its properties set (URI, method, entity, media type preferences, and so on) and invoking the `handle(Request, Response)` method. This method is available for any subclass of `org.restlet.Restlet`, such as Component, Application, Router, Filter. A similar `handle()` method is also available for subclasses of `ServerResource`.

The rest of this section discusses how to do unit testing using the common JUnit and TestNG tools to automate and structure your test suites.

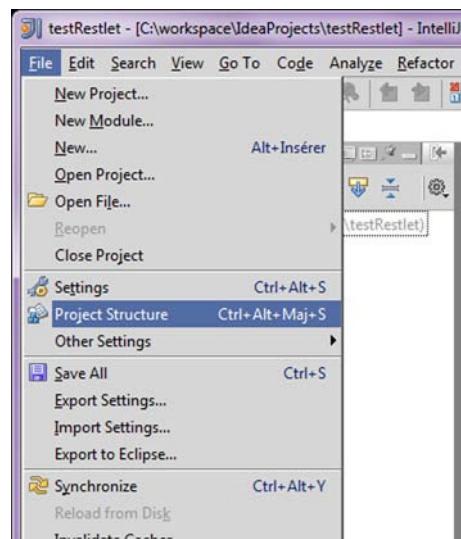


Figure B.37 Begin configuring the project dependency.

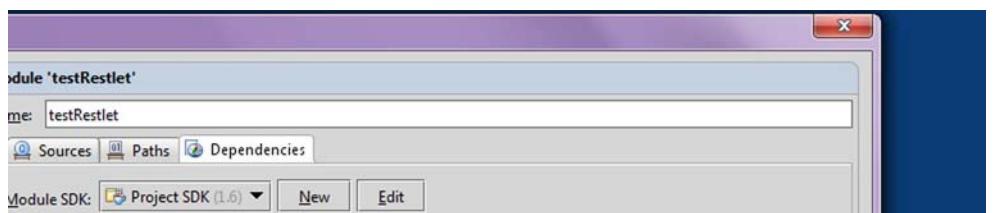


Figure B.38 Adding a dependency

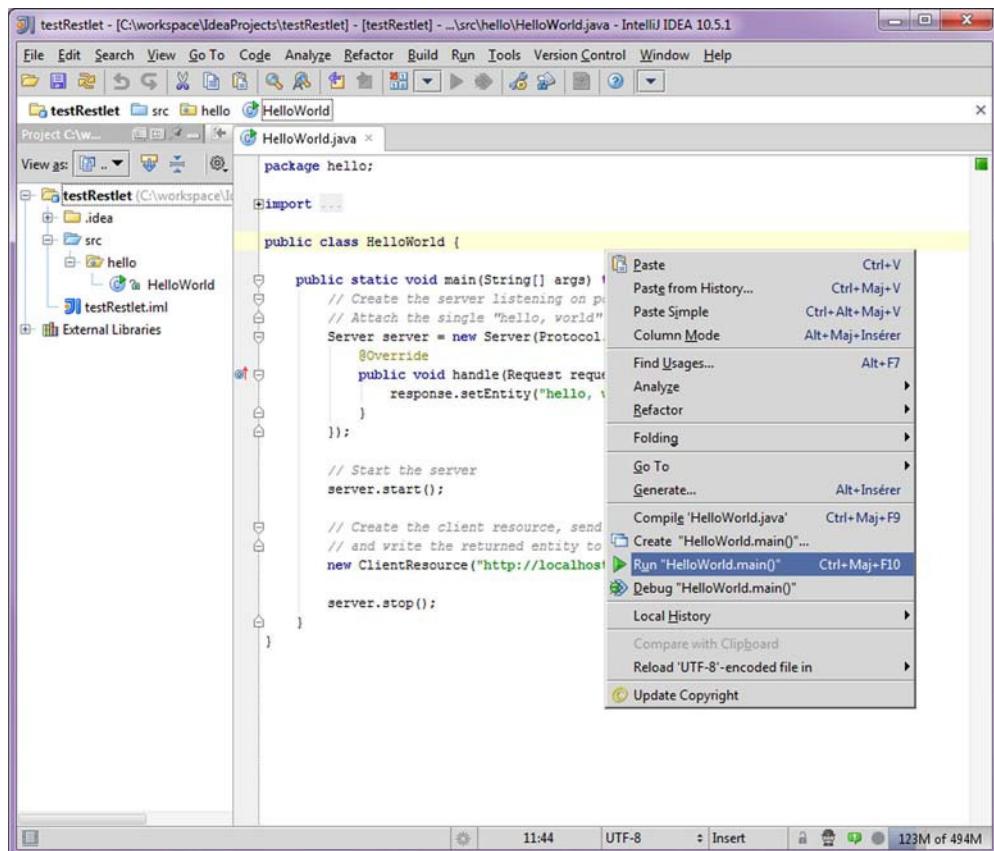


Figure B.39 Run the main class.



Figure B.40 The Output view



Figure B.41 Compile and launch the code.

JUNIT

JUnit is a widely known testing framework integrated into several tools and IDEs such as Eclipse, NetBeans, and IntelliJ IDEA. We'll illustrate it using the mail management application developed in this book. We'll focus on a particular resource and test it using three distinct layers of the application: the server resource itself, the hosting application, and the component.

The following listing contains a simple JUnit test case that instantiates the `RootServerResource` class, builds a request/response as it would be received, and lets the resource handle the call.

Listing B.2 Unit test for `RootServerResource`

```
import junit.framework.TestCase;
import org.restlet.Context;
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.data.Method;
import org.restlet.example.book.restlet.ch04.sec6.server.
    RootServerResource;
public class RootServerResourceTestCase extends TestCase {
    public void testComponent() throws Exception {
        RootServerResource resource = new RootServerResource();
        Request request = new Request(Method.GET, "/");
        Response response = new Response(request);
        resource.init(new Context(), request, response);
        resource.handle();
        assertTrue(response.getStatus().isSuccess());
        assertNotNull(response.getEntityAsText());
    }
}
```

Initialize resource, handle call → **Instantiate RootServerResource**

← **Prepare test HTTP call**

← **Test response**

← **Test content of response's entity**

In this case the target server resource is simple, and its logic doesn't rely on services offered by the parent application, such as the tunnel service. In real life, server resources are frequently using services and data from their parent application, shared with a sibling resource. It's therefore better to test them in their natural context.

The following listing contains a simple JUnit test case that builds a request/response as it would be received by the server connector and directly invokes a local instance of the `MailServerApplication` class.

Listing B.3 Unit test for Application

```

import junit.framework.TestCase;
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.data.Method;
import org.restlet.example.book.restlet.
    ↵      ch04.sec6.server.MailServerApplication;
public class MailApplicationTestCase extends TestCase {
    public void testApplication() throws Exception {
        MailServerApplication application = new MailServerApplication();
        Request request = new Request(Method.GET,
            "http://localhost:8111/");
        Response response = new Response(request);
        application.handle(request, response);
        assertTrue(response.getStatus().isSuccess());
        assertEquals("Welcome to the RESTful Mail Server application !",
            response.getEntityAsText());
    }
}

```

Another way to issue the client request is to create a `ClientResource` instance and locally attach the application to it using its `setNext(...)` method.

To complete the test suite, you can test the resource at an even higher level using a parent Component. The following listing shows almost the same test code except that it targets the `MailServerComponent` class.

Listing B.4 Unit test for Component

```

import junit.framework.TestCase;
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.data.Method;
import org.restlet.example.book.restlet.
    ↵      ch04.sec6.server.MailServerComponent;
public class MailComponentTestCase extends TestCase {
    public void testComponent() throws Exception {
        MailServerComponent component = new MailServerComponent();
        component.start();
        Request request = new Request(Method.GET, "http://localhost:8111/");
        Response response = new Response(request);
        component.handle(request, response);
        assertTrue(response.getStatus().isSuccess());
        assertEquals("Welcome to the RESTful Mail Server application !",
            response.getEntityAsText());
        component.stop();
    }
}

```

As a side note, the editions for Java SE and Java EE include a test module called `org.restlet.test` that contains the whole test suite for the Restlet Framework. These tests are all written using JUnit and offer a good start point for discovering Restlet code.

TESTNG

Inspired by JUnit, TestNG offers additional features such as running tests concurrently using thread pools. It's also integrated by various tools and IDE such as Eclipse, IntelliJ IDEA, and more. The following listing is a transposition of the test case used for testing the Component class, but using 10 concurrent threads issuing a total of 100 calls.

Listing B.5 Unit test for Component

```
import org.restlet.Request;
import org.restlet.Response;
import org.restlet.data.Method;
import org.restlet.example.book.restlet.ch03.sec3.server.
    MailServerComponent;
import org.testng.Assert;
import org.testng.annotations.AfterSuite;
import org.testng.annotations.BeforeSuite;
import org.testng.annotations.Test;

public class MailComponentTestCase {

    private final MailServerComponent component;

    public MailComponentTestCase() throws Exception {
        component = new MailServerComponent();           ← Instantiate Restlet component
    }

    @BeforeSuite
    public void beforeSuite() throws Exception {
        component.start();                            ← Start component's HTTP server
    }

    @Test(threadPoolSize = 10, invocationCount = 100, timeOut = 5000)
    public void makeCall() {
        Request request = new Request(Method.GET,
            "http://localhost:8111/");
        Response response = component.handle(request);

        Assert.assertTrue(response.getStatus().isSuccess());
        Assert.assertEquals(
            "Welcome to the RESTful Mail Server application !",
            response.getEntityAsText());
    }

    @AfterSuite
    public void afterSuite() throws Exception {
        component.stop();                           ← Stop component's HTTP server
    }
}
```

TestNG is also a good tool for integration testing, end to end, functional, and more.

B.3.2 Integration testing

This section introduces a set of tools that allows testing your application at the network level.

cURL ([HTTP://CURL.HAXX.SE/](http://curl.haxx.se/))

cURL is a command-line tool that supports several communication protocols including HTTP, HTTPS, and FTP. It's available for a wide range of OSs, including numerous Linux-based OSs, Windows, and MacOS.

It offers a large set of commands, but we'll stick to the simpler ones here. The first command allows you to get the representation of a resource. Give the URL of the remote resource as a single argument to the cURL command like this:

```
C:\>curl http://localhost:8111/  
Welcome to the RESTful Mail Server application !
```

To get the whole set of response headers, add the `-D` option. The headers will be dumped to the provided text file:

```
C:\>curl -D header.txt http://localhost:8111/  
Welcome to the RESTful Mail Server application !  
C:\>more header.txt  
HTTP/1.1 200 OK  
Date: Sun, 18 Sep 2011 16:21:57 GMT  
Accept-Ranges: bytes  
Server: Restlet-Framework/2.1.m5  
Content-Length: 48  
Content-Type: text/plain; charset=UTF-8
```

cURL is complete enough to give you the ability to manually specify all aspects of the request including headers (use the `-H` options), method (`-X`), and entity (`--data`, `--data-binary`, `-T`). We'll let you discover the other capabilities of this tool. For now we'll introduce RESTClient, which can be seen as a graphical version of cURL.

RESTCLIENT

RESTClient (<http://code.google.com/p/rest-client/>) is a Java application to test RESTful web services. You can use it to test any kind of HTTP server.

You'll repeat the same GET method as in the preceding section. The window is more or less divided into two parts. The upper part lets you configure the request—for example, the URL of the remote resource—and the method (figure B.42).

The lower part is dedicated to the introspection of the response attributes: headers, body, and so forth. You can glance quickly at the set of response headers (figure B.43).

As you may have noticed, this tool offers a simple but powerful interface for precisely testing your web APIs. In addition, writing test scripts using Java code against a RESTClient Java API is possible.

The following section adds the Restlet Framework to the heap.

CLIENTRESOURCE (JAVA CODE)

In case you need to programmatically test your REST API, you may find it useful to rely on the client API offered by the Restlet Framework. It's able to give you fine control on

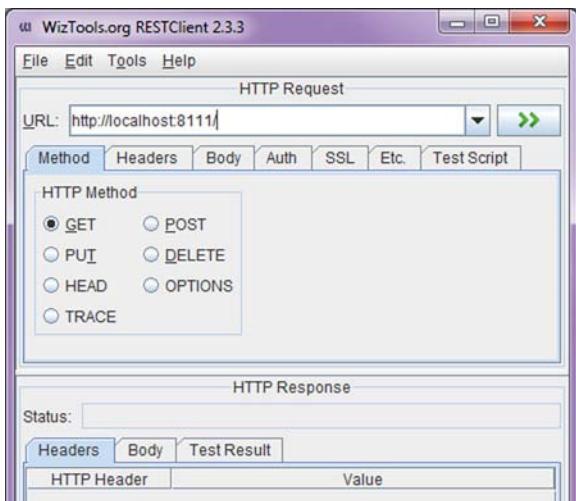


Figure B.42 Specify the request.

the request and access to all parts of the response: headers, entity, and entity attributes. Figure B.44 illustrates this in a few lines of code.

That's our short review of tools that help you test your web API. We'll end this appendix by sharing best debugging practices.

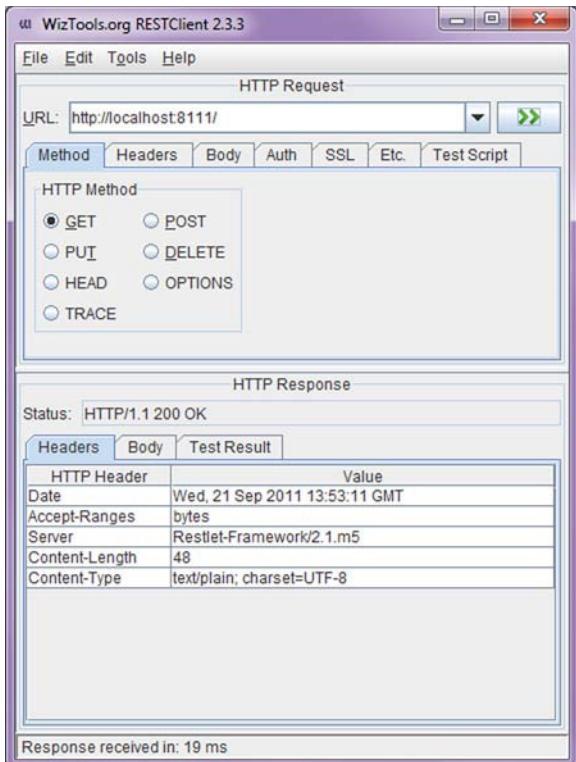


Figure B.43 Response headers

The screenshot shows the Eclipse IDE interface. The top window displays the Java code for `TestClient.java`. The code uses `ClientResource` to interact with a local host and prints the response headers. The bottom window shows the execution console output, which includes the application's welcome message and the printed response headers.

```

Java - testCache/src/test/TestClient.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
TestClient.java
package test;

import java.io.IOException;

public class TestClient {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) throws ResourceException,
        IOException {
        ClientResource cr = new ClientResource("http://localhost:8111/");

        Representation rep = cr.get();
        System.out.println("Response's entity           : " + rep.getText());
        System.out.println("Entity's size (in octets) : " + rep.getSize());
        System.out.println("Entity's media type       : " + rep.getMediaType());
        System.out.println("Entity's modification date: "
            + rep.getModificationDate());

        System.out.println("List of response's headers:");
        Series<Header> headers = (Series<Header>) cr
            .getResponses()
            .getAttributes()
            .get(org.restlet.engine.header.HeaderConstants.ATTRIBUTE_HEADERS);
        for (Header header : headers) {
            System.out.println(" " + header);
        }
    }
}

Problems Javadoc Declaration Console
<terminated> TestClient [Java Application] C:\Program Files\Java\jdk1.6.0_23\bin\javaw.exe (21 sept. 2011 21:29:20)
Starting the internal [HTTP/1.1] client
Response's entity           : Welcome to the RESTful Mail Server application !
Entity's size (in octets) : 48
Entity's media type       : text/plain
Entity's modification date: null
List of response's headers:
  +[Date: Wed, 21 Sep 2011 19:29:21 GMT]
  +[Accept-Ranges: bytes]
  +[Server: Restlet-Framework/2.1m5]
  +[Content-Length: 48]
  +[Content-Type: text/plain; charset=UTF-8]

```

Figure B.44 Use of a ClientResource to test a remote resource

B.3.3 Debugging problems

How do you debug a Restlet application? On a running system, the main source of debugging information is the log file recording the application's activity. We'll consider some of the logging features in the Restlet Framework that ease the configuration of log traces and the exploitation of these traces. Then we'll describe debugging practices when developing an application within an IDE.

The Restlet Framework log feature is based on the standard `java.util.logging` log API provided by Java SE (since version 1.4). By default, Restlet components generate Apache/IIS-like logs, which facilitate their exploitation by standard tools such as Analog. Thanks to the standard log API, you can also configure the format, the location of the log files, the rotation of the log files, and more. By default, the configuration of the logs relies on the system property `java.util.logging.config.file`, which locates the standard configuration. In case this property is missing, it's still possible to configure the log programmatically thanks to the `org.restlet.engine.Engine`

class. This relies on two static members: `logLevel` and `restletLogLevel`. The first line of code in the following listing turns off the log level for all loggers, and the second one turns on all traces for the loggers with names starting with `org.restlet`.

Listing B.6 Programmaticaly configuring the log level

```
Engine.setLogLevel(Level.OFF);           ← Turn off log traces for all loggers
Engine.setRestletLogLevel(Level.ALL);    ← Specify log for org.restlet* loggers
```

Another feature is the ability to restrict the generated logs according to the properties of the incoming request. Each Restlet component has a dedicated `LogService` object that copes with the access traces. The `LogService` class enables subclasses to customize the `isLoggable(Request)` method, which returns a Boolean. By default, it looks up the `Request.loggable` property that you can easily modify.

You can also create your own implementation of the `LogService` class and override this method to enable a log for a particular set of URIs, method, or any other attributes of the incoming request. The following listing shows how to enable traces for Internet Explorer browsers only.

Listing B.7 Programmaticaly enable log traces for MSIE clients

```
public class MyLogService extends LogService {

    @Override
    public boolean isLoggable(Request request) {
        return request.getClientInfo().getAgent() != null
            && request.getClientInfo().getAgent().contains("MSIE");
    }
}
```

When something goes wrong, and no trace can easily let you know what is going on, it's easy to use the step-by-step debugger of your IDE. To get a hold on the current request, you need to override the `handle(Request, Response)` method of the specific Restlet you want to start inspection from, put a breakpoint inside it, and follow the processing. You should attach the source code of the Restlet Framework available in the distributions to let you step inside the Restlet code and be able to follow the processing chain down to the problem you encounter.

Another rarely used feature offered by Java and easily usable with modern IDEs such as Eclipse is the ability to remotely debug a running JVM, including doing step-by-step debugging. If your server is hidden behind a firewall, it's still possible to use this feature by setting a local port forwarding through an SSH session using tools such as Putty on Windows.

appendix C

Introducing the REST architecture style

This appendix will give you a minimal REST background that will help you understand the main concepts at the core of the Restlet Framework. Understanding REST is key to using the Restlet Framework because it's the official architecture style of the web and because the Restlet API is a direct mapping of REST and HTTP concepts to Java. We'll also discuss the relationship between REST and HTTP, comparing it with the historical RPC-like alternatives such as SOAP.

C.1 Supporting all web features with REST

Billions of people use the web in one form or another. Millions of developers create web applications. Yet only a small number of them have heard about REST, which defines the architectural style of the web. REST was created by Roy T. Fielding, the primary architect of HTTP 1.1, the backbone protocol of the web.

REST is the acronym for *REpresentational State Transfer*. It's a set of principles that, when correctly applied, helps in building software architectures and applications that benefit from all the qualities of the web. Those qualities are numerous and include greater scalability, efficient network use, and independent evolution of clients and servers (also called *loose coupling*).

Now, would you build a Gothic church or an art deco hotel without knowing about the Gothic or art deco architectural styles? Certainly not! The same is true for the web; you shouldn't build web applications without knowing about the architectural style of the web. By understanding the principles defined by REST and applying them in your development, you'll be able to create distributed systems with highly valuable properties such as scalability, loose coupling, performance, and simplicity.

This section gives you the big picture of the web—its increasing role as the center of our information system and its impact on all types of applications and platforms like mobile devices. Next we discuss REST and its architecture elements and explain its relationship to the HTTP protocol.

C.1.1 *The all-embracing web*

Everybody knows about the Web 1.0, the web of hyperlinked text documents built on top of the internet, where web browsers and web servers are kings using the HTTP protocol to exchange those documents.

Later the AJAX technique offered a way to asynchronously send HTTP calls from web pages, clearing the path for Web 2.0 and the Rich Internet Applications that were more complex, reactive, and more social. This new way of using the web gave rise to a whole new range of frameworks, such as jQuery and Google Web Toolkit, that tried to facilitate the development of dynamic, JavaScript-based applications hosted in web browsers.

Many efforts were made to realize Tim Berners-Lee's vision of a web of machines talking to each other on behalf of their human owners in a more meaningful way. This is the Semantic Web, often called Web 3.0, that we cover in more detail in chapter 10. This led to the Linked Data movement and the creation of a web of hyperdata, mostly parallel to the previous forms of the web.

Cloud computing introduced a radical change in the way we build and provision applications at web scale. Making extensive use of web APIs, as exemplified by Amazon Web Services, cloud computing drives significant cost reductions, simplifies maintenance, and allows organizations to be much more reactive and focused than when managing the infrastructure themselves.

The latest wave has been led by mobile phones and tablets becoming smarter, touting fully featured web access, and offering new interaction experiences thanks to native apps and marketplaces. In countries where computers aren't widespread, mobile phones have even become the primary way of accessing the web. Even though the web browser isn't always at center stage, web standards such as HTTP and JSON connect those applications, often using back ends in the cloud to store and share data and do heavy processing.

We're now seeing the rise of the internet of things, also called the web of things, making the web more present in our daily lives and more transparent at the same time, entering our home in new forms such as connected body scales, thermostats, or power regulators. Hypermedia isn't always the best UI, and HTML 5 might not be the answer to all our interaction needs, but the web is still at work to exchange information via web APIs and allow more consistent experiences across all those machines.

Adapting to all those types of webs, as visualized in figure C.1, represents a big challenge when designing your web applications. They need to scale and adapt to various environments such as small screens, stay manageable, and evolve quickly to address new market trends or user needs.

These are a few examples showing how much the web is becoming ubiquitous and exerting a strong attraction on information systems. In the last ten years, we all have tried to adapt to it, exposing internal applications and services to the web, developing web gateways or tunnels, and building specific applications for each type of web.

We believe it's time to think differently—to build from the web rather than to constantly adapt to it. With REST in your mind, you'll be able to envision the web in a unified and empowering way. With Restlet in your hands, you'll be able to map REST concepts directly to your design and code and build unified web applications capable of supporting all those types of web at the same time.

C.1.2 How REST explains the architecture elements of the web

One of the masterminds behind the web is Roy T. Fielding. In 1994 he started working as a doctorate student at University of California Irvine, contributing to the specification of several web standards and rationalizing the design choices that were made.

In 2000, he published a PhD thesis under the rather abstract title *Architectural Styles and the Design of Network-based Software Architectures*, which included a chapter with an even more cryptic title: “REpresentational State Transfer (REST).” [12] This thesis remained confidential for a few years, but as REST was constantly promoted and interest in it grew quickly, the thesis became widely read and is still debated today.

Why so much interest in this text that's rather abstract and hard to digest? Well, REST explains the architecture of the web so well that it illuminates many features and design choices that were made by the builders of the web. It contains a blueprint that all web applications should follow.

The thesis starts with a general discussion of software architectures, continuing with specifics of network-based application architectures. Fielding then proposes a classification of architectural styles such as *pipe and filter*, *client-server*, *layered systems*, *remote sessions*, *virtual machines*, *code on demand*, and *distributed objects*. Building architects could have the same discussion about building architectural styles such as Roman or Gothic.

Fielding describes software architectures as a configuration of architectural elements, which are, in the case of REST, *components*, *connectors*, *resources*, *representations*, and a few data elements. In the following sections we take a look at the higher-level architectural elements, because this provides a good way to start comprehending REST and Restlet.

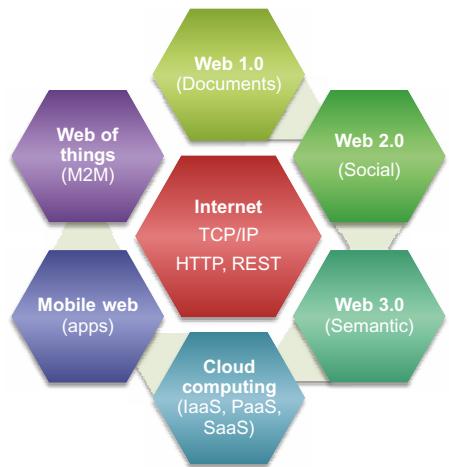


Figure C.1 Multiple forms of the web using REST and HTTP

RESOURCES

Let's start our exploration with the most important element: *resources*. They're the building blocks of the web. A resource can be anything of interest that an application wants to expose on the network for other applications to use. It can be the balance of a bank account, the current temperature in Paris, a Shakespearean text, a video of a play, a collection of pictures, a blog post, and so on.

One important characteristic of those resources is that they're linked together by embedded hyperlinks in HTML documents or URI references in feeds pointing to recent blog posts. In exactly the way it was envisioned and named by its creator Tim Berners-Lee, the web is a distributed set of resources that can be navigated and discovered dynamically.

What's amazing is that all those resources—retrieved, updated, or deleted by users, humans, or robots—form a complex system that's constantly evolving and growing. Figure C.2 illustrates this and shows the potential relationships between resources via hyperlinks. In practice, only a few relationships will be used and may sometimes even be broken. We all know the 404 “Not found” error pages that are annoying but that also prove the liveliness of the web.

To allow users to retrieve them from anywhere in the world, each resource is given a unique name called a Uniform Resource Identifier or URI (for example, <http://www.paris.fr/weather>), enabling identification and remote access on the internet.

As shown in figure C.3, a resource can expose its state via representations containing both metadata (such as size, media type, or character set) and content (binary image or text document). The representation of a confirmation of purchase on eBay could be an HTML document; for a wedding picture it could be a JPEG binary stream; for a contact in an address book web service it could be an XML fragment; and so on.

Let's look closely at a *resource* to see how it's typically implemented in an information system. The circle in figure C.4 delimits the *resource* from its external environment, which interacts with it.

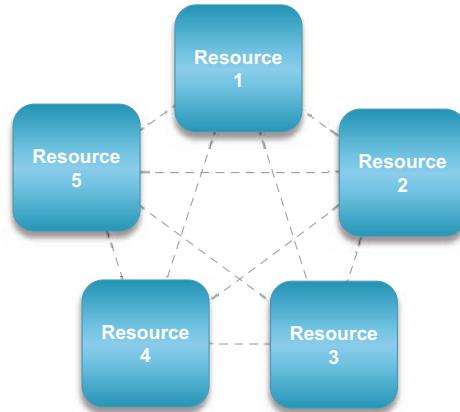


Figure C.2 The web as a graph of potentially hyperlinked resources



Figure C.3 Relationships between resources, identifiers, and representations

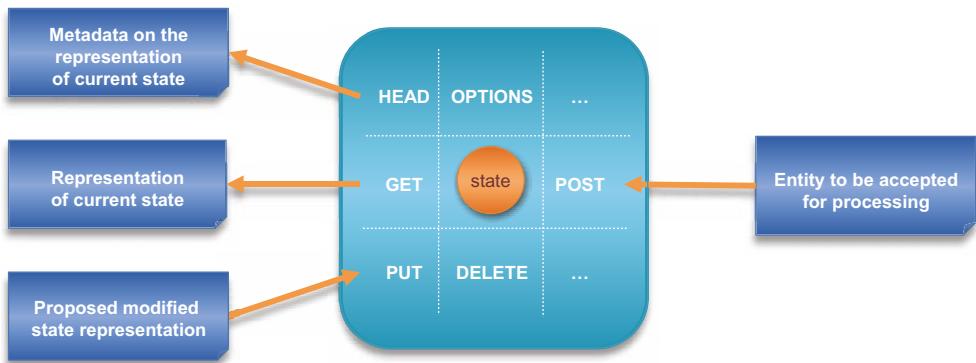


Figure C.4 Anatomy of a resource

The *resource* is composed of some state depicted at the center, managed in any way that makes sense (like in a database, in a file, or computed dynamically) and of standard methods (like GET, PUT, DELETE, or POST explained right after) that together define its *uniform interface*.

To give you an idea of the importance of *resources*, you can compare them to objects in the *object-oriented* paradigm. They are the backbone of your web applications, exposing both state and behavior. Compared to the object-oriented paradigm, the standard resource methods (usually HTTP methods) are similar to object methods, with a key difference: they're limited in number and have a behavior that's predefined by a protocol such as HTTP. Because applications use predefined methods, instead of defining their own, to express interactions over the network, the network infrastructure can understand to a certain degree the semantics of the interactions it's carrying on. From this understanding comes the ability to provide a number of key services such as caching and automatic message reemission, things an infrastructure can't do blindly without understanding some of the application-level semantics of the interactions.

REST and persistent state

REST itself doesn't define where or how the state of resources should be stored, only how it can be retrieved (via GET) or provided (via PUT and POST). It's also possible to have a read-only resource (only supporting GET). Otherwise, the resource state could be provided by any means such as a filesystem, a combination of other resources, physical sensors, and so on.

In addition, RESTful databases such as Apache CouchDB (<http://couchdb.apache.org>) exist, blurring even more the separation between the web and traditional information systems where relational databases are king. Although REST communications are stateless, REST resources are definitely stateful!

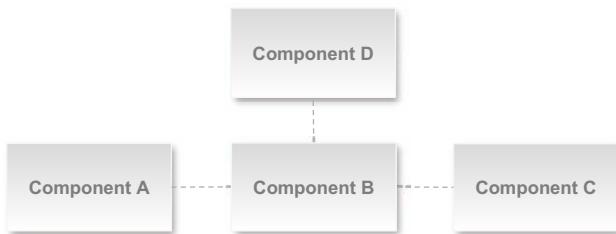


Figure C.5 Components are coarse-grained elements (HTTP server, client, proxy) distributed on the web

Another key advantage of this uniform interface principle is the reduction of coupling between clients and servers. The fact that you don't need to generate stubs or proxies and recompile your web browser each time you want to visit a new web site is a benefit of manipulating resources through a uniform interface. In appendix D, we push this comparison further and talk in more detail about the *resource-oriented* paradigm that REST introduces.

COMPONENTS

We've seen that resources are everywhere on the web, but that doesn't tell us much about how they're effectively distributed, managed, and accessed. For this, REST offers the notion of distributed components, coarse-grained software elements that encapsulate state and behavior and expose them on a distributed network such as the internet. It's important to understand that in this book and in the Restlet API, components aren't typical developer components but architectural actors in a distributed system.

Figure C.5 illustrates four components, distributed on the web. The dotted links correspond to a network communication, typically a protocol like HTTP on top of TCP/IP.

There are many types of components, such as *user agents*, *origin servers*, *gateways*, and *proxies*. Well-known user agents are web browsers (like Mozilla Firefox) and common origin servers are web servers and web engines (like Microsoft IIS and Apache Tomcat). Note that the Restlet Framework provides all the technology needed to develop such components.

CONNECTORS

Components need to communicate among themselves, and for that they use *connectors*, as illustrated in figure C.6, to abstract the communication details such as the network management (like TCP/IP sockets) and the protocol (like HTTP connections).

In REST, those communications are stateless. The client sends a request that's self-sufficient, the server returns a response, and the communication is ended. More

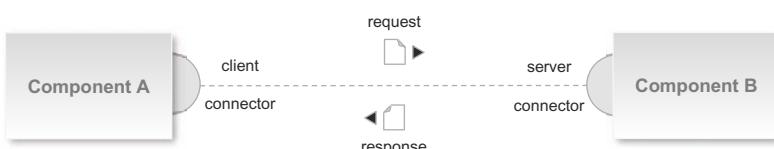


Figure C.6 Components communicate via connectors.

precisely, in the case of HTTP a socket connection is established that can be made persistent and reused for several transactions.

There are many types of connectors, such as *client connectors*, *server connectors*, *caches*, *resolvers*, and *tunnels*. The most common connectors are client connectors (like the Apache HTTP Client library), which send requests, and server connectors (like the Jetty HTTP Server library), which listen for incoming requests and send back responses. The Restlet Framework also provides a comprehensive set of connectors for protocols such as HTTP, POP3, SMTP, FILE, and many more.

SAMPLE INTERACTION

Let's now consider a more complete scenario, depicted in figure C.7, where a user agent (like a web browser) would be the client of an origin server, invoking methods on target resources identified by URIs. Typically a first GET request would retrieve Representation 1.1 of Resource 1 as an HTML document that would then be displayed in a web browser.

Then, following a hyperlink found in the representation, the user agent could retrieve a second representation of the same resource (Representation 1.2) in a different format (like a PDF document). Following another hyperlink, the user agent could access another resource (Resource 2) and retrieve its representation as another Representation 2 HTML document. This sort of interaction doesn't have to be limited to one user agent and one origin server or only to GET methods.

C.1.3 Understanding the relationship between REST and HTTP

You may still be unclear about the exact relationship between REST and HTTP. REST is definitely different in nature from HTTP because it doesn't identify the same type of things. REST is an abstract architectural style, HTTP is a concrete communication protocol.

HTTP is the main protocol to manipulate resources on the web and was at the basis of REST formalization. It's a client-server network protocol: a client application, be it a

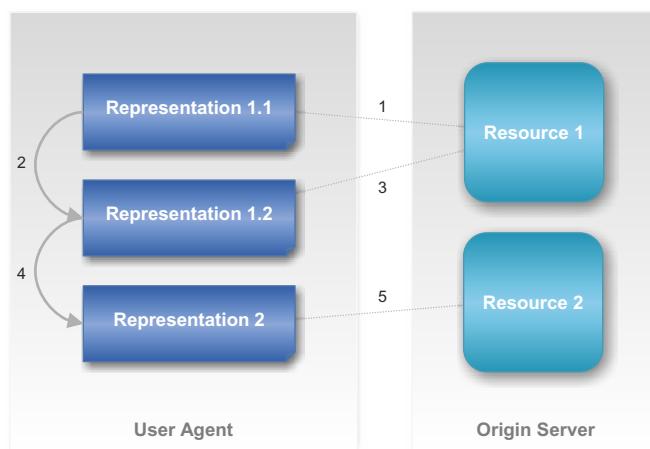


Figure C.7 Interaction between a *user agent* and an *origin server*

web browser or any other kind of program, sends a request to a server application, which processes it and returns a response. This response can be an HTML document (in the case of a typical web site), an XML document (in the case of a typical web service), or any other kind of data. The fact that presentation information (in HTML, for instance) and raw data (in XML, for instance) are dealt with in the same way is particularly interesting: it unifies two kinds of distributed systems that have often been implemented with completely separate technologies in the past.

But HTTP doesn't merely define a request/reply message exchange pattern: it goes much further to specify the semantics of the interactions between clients and servers. With HTTP, you don't say "send this request to this application" or "send back this response." The request targets a specific resource, identified by a URI, and manipulates it using methods defined by HTTP, such as GET, PUT, POST, and DELETE. Therefore an HTTP client program expresses things such as "give me a representation of the current state of this resource" (with GET), or "create or update this resource with the representation I'm sending to you" (with PUT), or "remove this resource" (with DELETE), and so forth.

Aside from this difference in nature, REST and HTTP are best friends because HTTP embodies the principles of REST. Consequently, creating RESTful applications is facilitated by the use of HTTP. But it isn't enough to use HTTP or an HTTP toolkit to make your application RESTful. In the same way that you can create non-object-oriented programs using an object-oriented language, you can create non-RESTful applications using the HTTP protocol. This isn't necessarily bad in itself, but if you want your application to benefit from the qualities of the REST architecture style, you need to actively apply its principles.

In the end, REST isn't dependent on a specific protocol such as HTTP. Its principles could be applied to other protocols. Roy T. Fielding has been considering a successor to HTTP—code-named Waka [28]—that would also be RESTful.

C.2

How REST became an alternative to RPC

Since the creation of the web in 1991, its spread has been so deep and fast that at the turn of the millennium, many believed that we understood it inside out. People had been developing websites, web clients, and web applications for 10 years, and the next step was to develop programmatic, machine-to-machine web services.

In the 1980s, a way of building network-based distributed systems arose. Named RPC, for *Remote Procedure Call*, it was adapted to object-oriented programming in the 1990s under the name *distributed objects*. The idea behind this approach was both brilliant and fascinating. It consisted of generalizing to distributed interactions a notion that works wonderfully well in everyday programming: *procedure call*, or *method invocation*. Using this approach, procedures and methods are no longer confined to the innards of a given program. They can be exposed by a program on the network and called from other programs, providing a high-level and familiar model for application-to-application interactions.

A number of toolkits were produced to support this distributed programming model and make its use as transparent as possible. The idea was to make calling a remote procedure as easy as calling a local one. To achieve this, an infrastructure of some sort had to kick in to marshal data (arguments and result) over the network and to coordinate the client and server process. The ultimate goal was to completely mask the distributed nature of the interaction, giving the illusion of calling a local routine and achieving full *location transparency* of the components in a distributed system.

In figure C.8 you can see some of the main RPC and distributed objects specifications and technologies produced over the years. The latest generations of RPC-inspired protocols such as XML-RPC, SOAP RPC, and AMF (a protocol defined for Adobe Flex) have in common the ability to use HTTP as their message transport layer.

In the 1990s, with the advent of object-oriented programming, distributed objects systems became hyped as the general solution for distributed computing. They would mask the complexity of implementing distributed systems and lead us into a bright, pleasant programming world!

Unfortunately, that turned out not to be the case. Although such technologies were relatively convenient for producing prototypes, creating robust, scalable, and working distributed systems was a nightmare. Moreover interoperability was limited, and coupling was strong between distributed components, making it difficult to evolve one component without disrupting existing callers.

Although RPC was a seductive idea worth exploring, it turned out to be a bad one, at least in the context of the current computing technology. In 1994 a group of researchers at Sun published *A Note on Distributed Computing*, considered to be one of the most important papers in the field—a must-read for every software architect. It convincingly explained why trying to deal with remote objects as if they were local was doomed to failure. Another reminder was given by Martin Fowler in his first law of distributed computing.

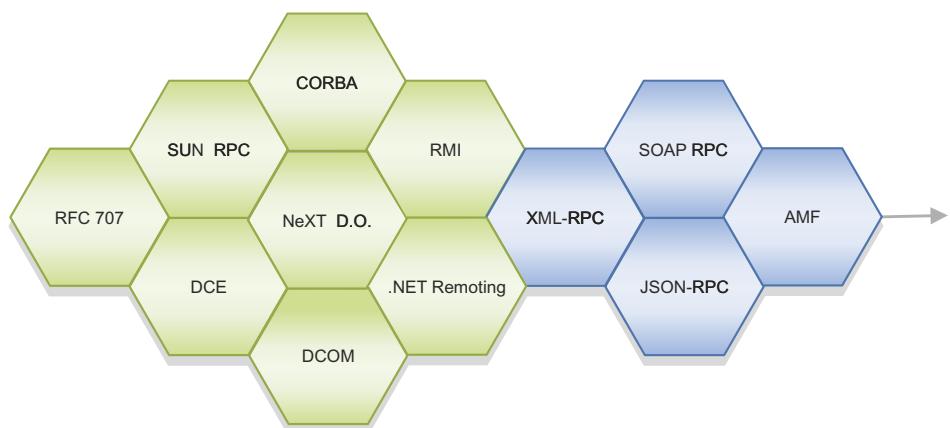


Figure C.8 Evolution of the RPC and distributed objects technologies

Early warnings against “transparent” distributed objects systems

“We argue that objects that interact in a distributed system need to be dealt with in ways that are intrinsically different from objects that interact in a single address space. These differences are required because distributed systems require that the programmer be aware of latency, have a different model of memory access, and take into account issues of concurrency and partial failure [...] A better approach is to accept that there are irreconcilable differences between local and distributed computing, and to be conscious of those differences at all stages of the design and implementation of distributed applications [...]”

“Programming a distributed application will require the use of different techniques than those used for nondistributed applications. Programming a distributed application will require thinking about the problem in a different way than before it was thought about when the solution was a nondistributed application. But that’s only to be expected. Distributed objects are different from local objects, and keeping that difference visible will keep the programmer from forgetting the difference and making mistakes.”

—Extracts from *A Note on Distributed Computing* [30]

Despite these early warnings, the software industry proceeded with the RPC approach, trying to fix problems along the way while still pursuing the goal of location transparency. Some havoc ensued. For example, some realized that because it was impossible to manipulate remote objects like local ones, the only solution to preserve location transparency was to deal with local objects as if they were remote. This culminated with the introduction of components models such as COM/DCOM and EJB, where we had to deal with the complexities of distributed computing regardless of the nature, distributed or not, of the application in development.

Meanwhile, an incredibly successful distributed platform was spreading like fire: the web, with HTTP as its core. Although the web was mainly used for user-facing applications (such as using a browser to interact with a remote server), some thought about a new way of using it: generalized programmatic application-to-application interactions over the network. Upon reception of an HTTP request, instead of returning an HTML document, an application could return raw data (in the form of an XML fragment, for instance) to another application (not necessarily a web browser). The idea of web services was born. Great inventions often use existing things in new ways—think about how people are using their phone line for watching movies or playing online games—and this simple idea had the distinctive mark of such inventions. It quickly became popular: using HTTP would allow using the existing infrastructure of the web, ensuring interoperability and making worldwide distributed interactions possible.

At that point, two main approaches emerged. The first, supported by industry heavyweights such as Microsoft, IBM, and Sun, elected to use HTTP in what appears now to be a rather awkward way: as a mere low-level transport layer. This way uses only a tiny part of HTTP, mainly the bits that allow moving some data from one network

How things develop

Karl Popper's comment, made in the context of biological evolution, also applies well to the development of the web: "It is always the existing structure of the network that determines possible variations or accretions. It is the existing network that holds the potential for the inventions to come."

node to another. On top of that, other protocols, such as SOAP, were defined to provide higher-level mechanisms, such as an extensible envelope abstraction, a way to convey application-level errors, and so on. This is awkward, because HTTP isn't a transport layer protocol, like TCP and UDP are, but a higher-level protocol that already provides features such as extensible envelope, error management, support for application-level semantics, and more. Effectively, this approach consists of reinventing the wheel and putting somewhat useless layers and layers on top of the existing HTTP layer, while producing all sorts of mismatches and complexities along the way.

Besides, although the natural architectural style of HTTP is REST, most of the web services toolkits were still designed according to the RPC style. They suffered consequently from many of the same woes as their CORBA and DCOM ancestors, including strong coupling—a fatal flaw for large-scale distributed systems spawning multiple organizations or development teams, given the high cost of coordination required to evolve such strong coupled systems over time. SOAP misused HTTP so much that it nullified many of its benefits. In particular, SOAP didn't allow the use of URIs to identify important resources, contradicting one of the fundamental ideas behind the web. SOAP also initially only used the HTTP POST method to emit requests, even when the goal was pure information retrieval—something HTTP GET was designed and optimized for. It was like owning a Ferrari but driving it only in reverse.

What an odd concept!

In his MIX'07 session "Navigating the Programmable Web" [31], Don Box talk with great humor, as always, about mistakes made in the early web services approaches, notably in the design of SOAP, of which he was one of the main authors.

"You could implement 'get' on top of POST if you wanted to, but you're ignoring this fantastic piece of technology [that is the web]... Many of us tried to do 'get' on top of POST and it works, but it doesn't work anywhere near as well as when you do 'get' on top of GET... What an odd concept!"

Instead of expressing requests and responses with HTTP semantics, these SOAP and WS*-based web services technologies produced messages with semantics that were opaque for the web infrastructure, making it impossible for this infrastructure to perform the essential services it was capable of, such as caching, content negotiation, automatic request reemission, incremental transmission, and so on. On the other

hand, it allowed SOAP to be transport-protocol neutral—to be able to be tunneled over other protocols, not only over HTTP. “Great, my programs aren’t adherents to HTTP,” developers could have thought. But this was trading an adherence to HTTP for an adherence to SOAP, which was a questionable move in a world where the existing web infrastructure and technical ecosystem is HTTP-based, not SOAP-based.

All these mismatches between the web and these web services initiatives probably came from the fact that, in the late 1990s, people crafting these technologies were just like most of us at that time: they didn’t understand HTTP and the web, and they were coming from the RPC world. The Fielding thesis wasn’t even published.

But a few questioned this approach and investigated, using HTTP and its native architectural style to its full extent. If our account is correct, it was two people, Roy Fielding and Mark Baker, who started it all. They challenged the mainstream web services approach and started to explain how and why using HTTP the way it was designed was an idea worth exploring, even in this new context of application-to-application interactions. It may seem obvious now, but for many it was not so clear back in the early 2000s. Many in our field, including one of the authors of this book, are directly indebted to Mark Baker for being tipped in this direction.

As years passed, REST became more and more popular as an alternative to RPC. It brought powerful architectural properties to distributed systems, including loose coupling, scalability, performance, and so on. Besides, directly using the HTTP semantics instead of relying on the SOAP/WS-* stack proved to be beneficial in most situations where using the web as a communication channel was a good option to begin with. REST is certainly not the only interesting architectural style, and it isn’t suited to every context. But as an alternative to RPC, it constitutes progress in the field of large-scale distributed systems.

The HTTP specification itself changed in status. A few years before, most of us thought it wasn’t something an application developer had to bother with. We tended to see it as something low-level, like the TCP specification. We now realize that it’s a high-level protocol, designed to directly convey our applications’ semantics.

Along the way, epic efforts—in particular in the W3C TAG working group—succeeded in making the SOAP/WS-* stack a little less web-hostile. But it wasn’t enough for the stack to ever recover, and it probably won’t escape the fate of its predecessor, CORBA.

Finally, development frameworks dedicated to helping implementing RESTful systems appeared. In the Java world, Restlet pioneered this movement, which also happened on other platforms and is now embraced by the big vendors including Microsoft, IBM, and Oracle.

appendix D

Designing a RESTful web API

As described in appendix C, REST introduces a new paradigm for distributed systems. It may still feel unfamiliar, and you probably have many questions about how you can use it for your own projects. You may even have the feeling of coming late to the party, having to catch up with yet another huge set of concepts and techniques that others are already mastering. We think it's time to share a little secret with you, which is also good news: you aren't late; in fact, you're among the first ones.

This fact is one of the most exciting aspects of the current dynamic around REST: we're witnessing the discovery and refinement of what makes the web so powerful and how it can be used for new things—in particular, in the field of enterprise computing (integration, cloud computing, Mobile Web, Semantic Web, and more). We're at the beginning of mass adoption of REST in these domains, and a lot of fascinating discussions and innovations are happening in the REST space, similar to the early years of object-oriented programming. Researchers and practitioners around the world are proposing and discussing REST design and development practices, aware that some of them will be keys to the future of the web. Even better, many of these discussions and developments are public, and you can take part in them or benefit from them in your own projects.

To that end, in this appendix we'll look at RESTful web projects from a methodological perspective. We won't cover Restlet specifically here: our goal is to give you guidance on how to carry on projects involving REST and how to design RESTful web APIs. The meat of the matter is the presentation of ROA/D, a pragmatic project methodology dedicated to Resource-Oriented Analysis & Design. ROA/D is based on our consulting experience at Restlet SAS since 2008, helping

our customers to design their RESTful web APIs, and on the feedback of users in the Restlet community.

We'll start with an introduction to ROA/D, giving an overview of the four main phases of a project, which are composed of iterations over a number of steps. Along the road we'll detail each step of the methodology by providing guidance and best practices, using UML for the visual communication. In the spirit of Manning's *in Action* series of books, we'll illustrate major concepts and practices in this appendix around an example: the RESTful email system that's implemented across the book's chapters.

D.1 **Succeeding in a RESTful project: the ROA/D methodology**

Choosing the Restlet Framework to build your next web application will take you a long way toward successfully using REST. This is the purpose of a framework after all—to make sure that the right things are easy to do and offer the necessary tooling.

As you discovered in the book chapters, Restlet is perfect to implement RESTful applications. But as you can create Java programs that break all object-oriented principles, you can also create Restlet programs that break all REST principles. You need additional guidance, not software, to reach your goal.

In this section we discuss the impact of REST on project-development practices and explain why it requires a change in development practices. Then we introduce the ROA/D methodology and highlight its similarities and differences with common methodologies such as Object-Oriented Analysis and Design (OOA/D). And we review usual project phases to see where REST, Restlet, and ROA/D matter the most.

D.1.1 **Introducing Resource-Oriented Analysis & Design (ROA/D)**

After reading chapter 1, we hope you felt the disruptive nature of REST and now understand why REST forces you to rethink web development. REST requires you to think in *resources* instead of the objects of the object-oriented world or the tables of the relational world. Those resources are identified by URIs and hyperlinked together, exposing representations of their state. They also offer a uniform way to interact with them, mainly via the HTTP standard methods (GET, PUT, DELETE, POST, and so on).

Designing a RESTful web API while providing useful features to its users isn't trivial. It requires special skills to understand the problem domain, analyze the requirements, and design the set of resources composing the API. Figure D.1 illustrates how we position the RESTful web API.

We define a RESTful web API as a set of hyperlinked resources, forming a subset of the web, exposed by a web service or a website, and consumed by web clients. Defining this API isn't easy and requires you to

- Analyze the requirements of your web project and the features to provide
- Design the set of resources, their granularity, their URIs, the content and structure of their representations, the standard methods that they expose (maybe depending on the user's role), and the hyperlinks between themselves

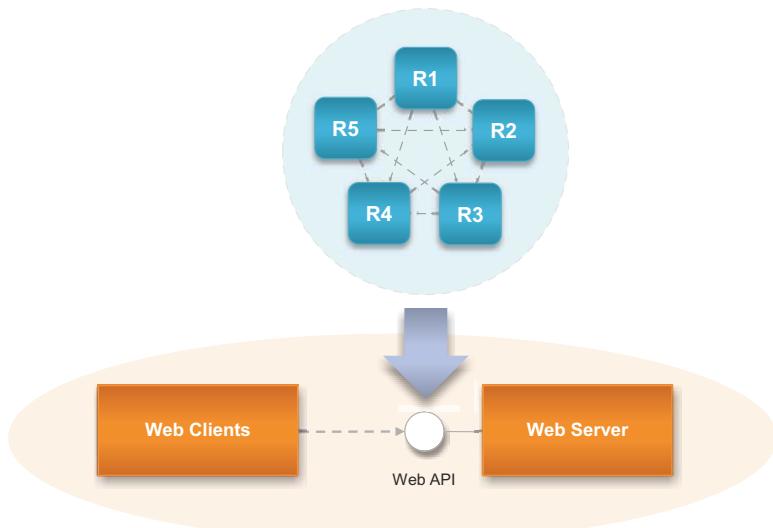


Figure D.1 Overview of a RESTful web API

Once you have your API defined, you can

- Implement it using technologies like the Restlet Framework
- Test it to make sure it satisfies requirements and doesn't break in the future
- Deploy it so that your users can use it

That's a lot of work ahead, and you need to be well organized to succeed. To guide you during this journey, we propose you use ROA/D, a methodology that ensures that you won't miss important steps during your RESTful web projects. We didn't reinvent the wheel—ROA/D fits nicely into popular development processes such as SCRUM, XP (eXtreme Programming), and UP (Unified Process). All those processes are iterative in nature, as opposed to the classic waterfall processes. Each of those project iterations is composed of a set of steps as illustrated in figure D.2.

The earlier you're in a project, the more important the first steps (*requirements gathering*, *requirements analysis*, and *solution design*) are. As you move forward in your project, requirements and design should become more and more stable. Your focus will shift more on the later steps (*design implementation* and *implementation testing*). During the last iterations, you should have successfully deployed your project into production and ensured that users are using it successfully.

Figure D.3 illustrates traditional project phases that every software development project follows, more or less formally. RESTful projects, no exception, also follow this series of phases.

Even if those phases are sequential, they keep an iterative nature. Several iterations will likely be necessary during each phase, with a shifting focus from *requirements*, *analysis*, and *design* during *inception* and *elaboration* phases, toward a focus on *implementation*.

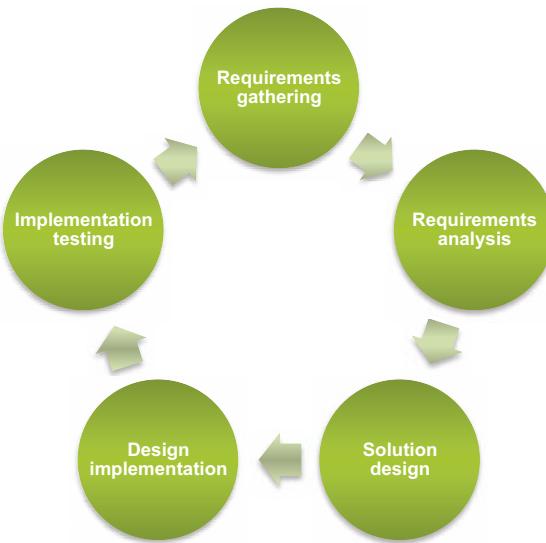


Figure D.2 Typical project iteration steps

and *testing* during *construction* and *transition* phases. As you move from the beginning to the end of this process, your project should get more and more stable, encouraging early feedback and addition of requirements, while giving opportunities for features scope reduction in order to meet project goals in terms of time and budget.

As its name implies, the ROA/D methodology is valuable during the *inception* and *elaboration* phases, when you progressively define and stabilize your RESTful web API. Using the Restlet Framework progressively increases to reach a peak during the *construction* phase. This methodology is an adaptation of the popular OOA/D methodology to RESTful application development.

If you're interested in learning more about OOA/D, we highly recommend Craig Larman's book *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* [32]. Considering that Restlet is an object-oriented framework to implement resource-oriented designs in Java, OOA/D skills are valuable, in addition to those of ROA/D, to succeed in your REST project.

While applying ROA/D we also use Unified Modeling Language (UML) as the recommended visual communication language rather than invent yet another visual notation. Doing so gives us the opportunity to clarify a common misconception: UML is neither a methodology nor a development process; it's different from OOA/D,

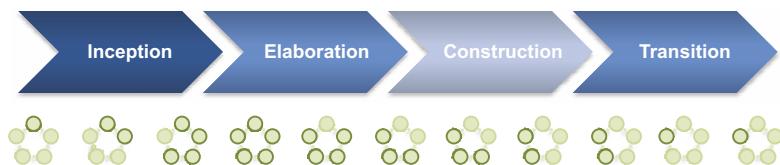


Figure D.3 Typical project phases defined byUP

ROA/D, SCRUM, XP, and UP but also complementary to them. UML is a standardized visual diagramming notation that facilitates the communication between participants in a software project. It facilitates the representation and sharing of your models, which is extremely important, but it doesn't say whether your models are good or bad.

The following subsections review each phase of a typical ROA/D project.

D.1.2 *Inception* the project

During *inception*, the project vision should be defined, more or less formally, producing a business case and a set of initial requirements. Those requirements will be refined during the next phases. At this point, you're looking for an overall feature scope and main use cases.

This phase is also a good time to clarify the overall time frame and budget of the project, assess its technical feasibility (for example, developing proof of concepts prototypes, UI sketches, and a logical architecture), and financial feasibility (for example, making buy-or-build decisions and ROI estimates). At the end of this phase, you should decide whether or not to continue, and under what conditions.

D.1.3 *Elaborating* the solution

During the *elaboration* phase you identify the majority of the requirements, define a domain model, detail the logical architecture, and reduce the main risks, making sure that the team is properly trained and organized.

This is another phase where several iterations should happen, considering the unstable state of the project. It's a great time for analysis and design workshops and technical prototypes that form reusable subsets of the final system. This is clearly the phase where the ROA/D methodology is the most useful, because your RESTful web API is specified with a good level of details and your Restlet-based architecture is in place.

At the end of this phase, you should have a clear idea of the amount of effort remaining and the resources needed to successfully complete the system and deliver the project.

D.1.4 *Constructing* the solution

At the beginning of this phase, most of the uncertainties should have disappeared, and the team should be ready to efficiently implement the remaining features of the project. It should be a time of active coding (using the Restlet Framework, APISpark, or a similar technology) and steady progress.

There's still a good chance to have changes in scope and requirements along the way, but their impact should now be marginal. Otherwise, it means that the *elaboration* phase wasn't correctly followed; maybe because of lack of time or because the external environment forced in some changes that couldn't be correctly taken into account at that point.

Again, the iterative nature of the process should ensure that you identify blocking issues as early as possible. At the end of this phase, all the features of the project should have been completely developed, and the quality level should be good.

D.1.5 Transitioning the project

The project should follow final rounds of testing and bug fixing and be deployed in production. At this point, no new feature should be added, the code base should be frozen, and only fixes should be made after peer review.

This is also a phase where the migration procedures from the previous solution, if they exist, should be tested and, finally, executed. Users should also receive the proper training to ensure a smooth adaptation without too much impact on their productivity.

Each of the phases we've described is composed of iterations over a number of steps (you saw them in figure D.2). Let's examine these steps, one by one, for a typical ROA/D iteration, starting from the fundamental requirements-gathering step.

D.2 Gathering requirements

During the first step of an ROA/D iteration, you collect, write, and organize the requirements of an application. To illustrate this we reintroduce you to the example application developed in this book and show you how this step can be carried out in the context of this particular example.

D.2.1 Collecting requirements from the sources

Gathering requirements is essential because it's the basis of the agreement between users and developers. They should be understandable by users and should be as free as possible of implementation details; the goal is to define the *what*, not the *how*. Frequently, users come up with a solution to the problem they face, but discussing and understanding the real issue is essential, because the naïve solution may not be the best in the end.

In an ideal world, all requirements would be known and described in advance, before the next steps begin and during the project *inception*. In practice, those requirements are initially incomplete and evolve during the *inception* and *elaboration* phases to stabilize during the *construction* phase. Each of the iterations gives an opportunity to validate, refine, or remove the requirements—in particular, when users can test or react to proposed solutions.

For our example application we wanted to find a domain that everyone would be familiar with along with a problem for which REST would be a particularly good fit. One of the most successful internet applications has been email. Its principle is simple: allow the asynchronous exchange of messages between people or programs across the internet. The SMTP protocol is used to exchange email messages, and the POP protocol is used to retrieve them from email boxes. Both protocols were defined in the 1980s—well before the HTTP protocol that now powers the web. Here's our challenge for this book: build a RESTful email system that could be a basic alternative to classic email!

D.2.2 Classifying requirements by priority

The main deliverable for this step consists of a list of textual requirements, ordered by priority. Note that requirements are also sometimes called *stories* to underline the descriptive nature of the intended behavior of the application.

It's now time to look at the requirements of our example project. It would take too much space to describe all project iterations, so we'll only show you the result. Keep in mind that this book project took us several months, during which we iterated over our example application and refined our requirements for it. We present the result obtained after the *inception* phase. We identified more than 20 requirements that we classified into three priority groups.

PRIORITY 1: CORE REQUIREMENTS

First we described the foundation of the application—the core concepts and needs on which more advanced requirements are defined:

- Mail is a textual message exchanged between a sender and receivers.
- Mail is composed of a subject, date, and textual body.
- Mail can have the following statuses: draft, sending, sent, receiving, or received.
- Contacts are records about potential mail receivers.
- Mail addresses are defined using HTTP URIs.
- Contacts are composed of a name and a mail address.
- Several accounts can be associated to a single user, called the *owner*.
- Accounts contain the mails and contacts of their unique owner.
- Mail servers manage several accounts.
- Users can have two roles: application administrator and account owner.
- Administrators can create and delete accounts.
- At least one user with an administrator role exists.

PRIORITY 2: GETTING USERS ORGANIZED

With the core requirements, you can build a RESTful mail server. But to make it usable by end users, you need to add a series of requirements. Some are illustrated in figure D.4, providing an overview of the RESTful mail system.

- Mailboxes can be accessed and managed by their owners from a simple mail client (like a regular HTML browser), a rich mail client (like an AJAX web browser), or from a mobile mail client (like a smartphone).
- Mail can be marked by the owner with any number of textual tags.
- Mail status is managed using application tags.
- Account owners can create web feeds, tracking the latest mail with a given set of tags.
- Accounts contain the feeds of their owners.

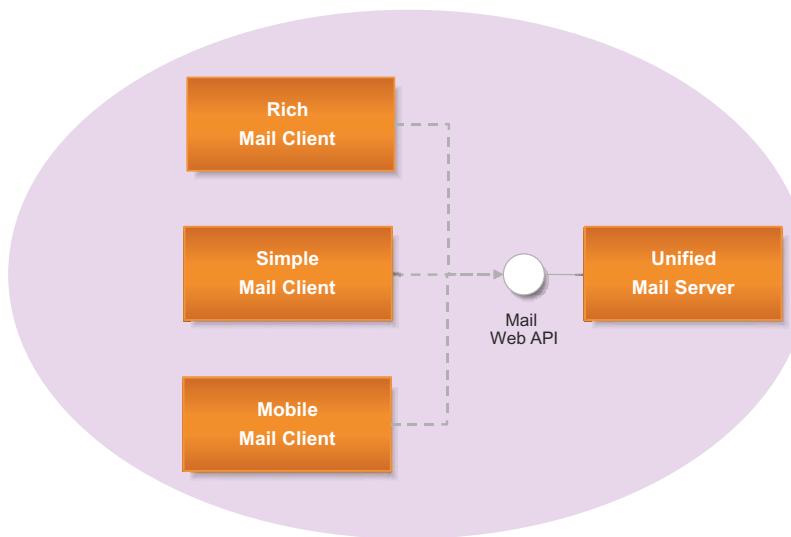


Figure D.4 Overview of our RESTful mail system

PRIORITY 3: INTEGRATION WITH CLASSICAL MAIL

To make the mail system usable in an existing context, you need to provide gateways to the classic mail system—the purpose of this final series of requirements.

- Contacts can also contain a classical mail address (`name@domain.tld`).
- Mail can be sent using the classical SMTP protocol when a contact has no URI address.
- Mail can be received using the classical SMTP protocol and associated to a mailbox.
- Mail can be retrieved from remote servers via the classical POP3 protocol.
- Contacts can be exposed using the FOAF language, providing a mapping to their SMTP mail address in addition to their URI address.

Once the requirements are gathered, you can move to the next step: analyzing them.

D.3 *Analyzing requirements*

You can now go to the next step of an ROA/D iteration and analyze collected requirements in order to understand the problem domain and present them as more practical artifacts. The goal is to investigate and describe how the system will be used and what its main functions are. Analysis can also be helpful to identify additional requirements.

This section contains only a high-level description, and we recommend that you consult a more complete reference such as the *Applying UML and Patterns* book from Craig Larman [32]. We cover the tasks that are typically done during the analysis steps:

- Describe the main usage scenarios, with UML use case diagrams.
- Define the domain model, with UML class diagrams.
- Describe the main system sequences, with UML sequence diagrams.

D.3.1 Describing usage scenarios

To explain what we expect from a system, it's useful to describe how we'll interact with it, as if it were a black box. For this purpose we define main usage scenarios in the text and illustrate them with UML use-case diagrams. Those diagrams describe interactions between the RESTful mail system and the outside world, also called *external actors*. While reading the requirements, we identified these actors: mail senders, receivers, account owners, and administrators. The first use case in figure D.5 shows an interaction between a mail sender and receivers using our system. At first sight, it looks trivial, but it's important to describe the evident things in order to coin key terminology and ensure that you don't lose the overall picture of the system.

Before effectively exchanging emails, being able to manage the mail application is essential. This is the role of the administrators, illustrated in figure D.6. Those power users can create new accounts, grant them roles, and delete them.

As explained in the requirements, an account is associated with one and only one user, but that doesn't prevent a user from owning several accounts. Now let's focus on the interactions between an account owner and the mail system, illustrated in figure D.7.

An account owner composes new mails by connecting to the system using a mail client, creating a draft mail, and entering the receivers, subject, and message. He can write his message in several shots, saving draft mails and editing them again later. Once satisfied with a draft, he can ask the mail server to send the mail. Section D.3.3 goes over exactly what happens when a mail is sent out, using a system sequence diagram.

The account owner can consult the mailbox and select mails he wants to read, associating tags to stay organized. The mail status is also exposed via special tags. And he can create web feeds based on a selected set of tags and point his feed reader to the given URI to retrieve the matching mails.

D.3.2 Defining the domain model

The most important artifact resulting from the analysis is the *domain model*. It's a rather conceptual model that's the basis of other more concrete models created during the design steps. It serves to identify the main domain entities and their relationships.



Figure D.5 Sender exchanging an email with receivers



Figure D.6 System administrator managing accounts

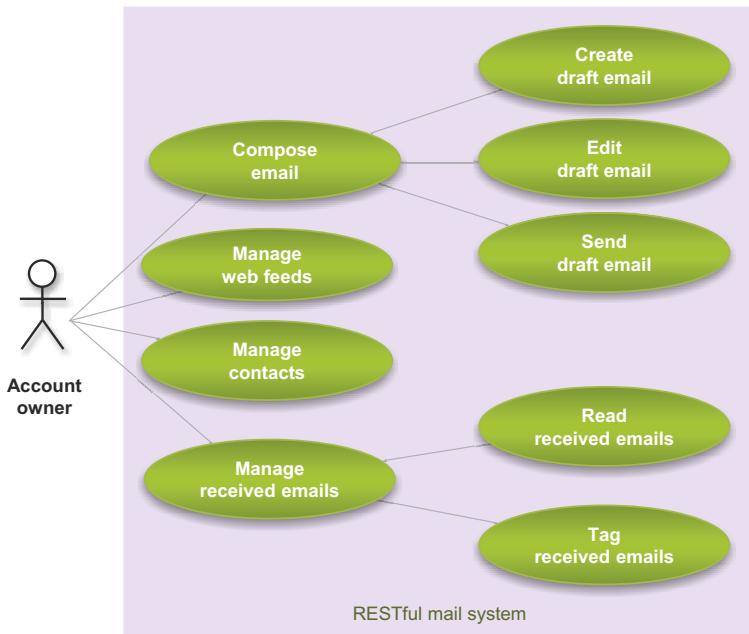


Figure D.7 Account owner composing mails and managing web feeds, contacts, and received mail

While reading the textual requirements, you should look for the most important nouns to identify these entities and their properties. Note that the required analysis skills are the same as the ones necessary in OOA/D. The domain model is still an object-oriented artifact that you later derive into resource-oriented artifacts.

Figure D.8 identifies six main entities in the example mail system and illustrates their relationships using a standard UML class diagram.

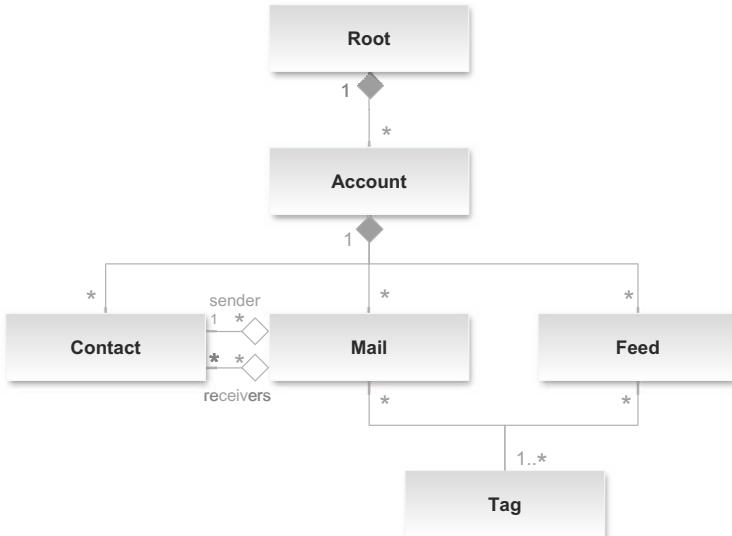


Figure D.8 Domain model of the mail system

If you're not too familiar with UML, note that the links with a black diamond indicate a composition; those with a white diamond indicate aggregation. Composition is a stronger form of association, indicating that the lifecycle of the contained class is directly connected to the one of its aggregate.

For example, the Account class is composed of several occurrences of the Contact, Mail, and Feed classes. You can also read the associations the other way around. The Contact class is associated to one and only one Account, and the same is true for the Mail and Feed classes.

There are interesting links between Mail, Feed, and Tag. They have no diamonds, indicating that there's no “part of” relationship—no aggregation or composition. It's a more balanced association relationship corresponding to the notion of “match” between the set of tags and mails or feeds.

As you can see, there's nothing specific to REST and resource orientation so far; you can rely on your existing analysis and UML knowledge. But this is a key step for the ROA/D methodology because it allows you to more easily build your resource model later on and bind it with a persistent object model.

D.3.3 Describing system sequences

Although the use-case diagrams are useful to identify the actors and list the main uses of the application, they don't describe precisely the input and output events to the system and the sequence of actions that we discussed previously. That's the role of system sequence diagrams. Figure D.9 details the typical sequence of actions involved when a user wants to compose and send an email.

The actors interacting are listed at the top of the UML sequence diagram, and each one of them has a vertical axis representing the time. The arrows illustrate the messages exchanged between the system parts (Sending mail server and Receiving mail server) and the actors (Mail client).

Because the domain is relatively simple, this analysis task may not seem too compelling. But as the complexity of the domain grows, you'll likely rely on it more often. This is also a good preparation for the interaction design in the next steps.

Adding a textual description of the diagram and the various interactions would be useful. Also, you could define additional sequence diagrams—for example, regarding the use cases of the administrator. We leave that to you as an exercise.

Once the requirements are analyzed, you can progress to the next step: designing a concrete software solution that matches your requirements and your understanding of the problem domain.

D.4 Designing the solution

After much effort spent specifying *what* you need to build and then understanding and describing this *what*, here comes the *solution design* step of an ROA/D iteration. In this step you actively take into account the specifics of the resource-oriented paradigm and think about *how* to implement the solution. In this section we discuss ROA/D's

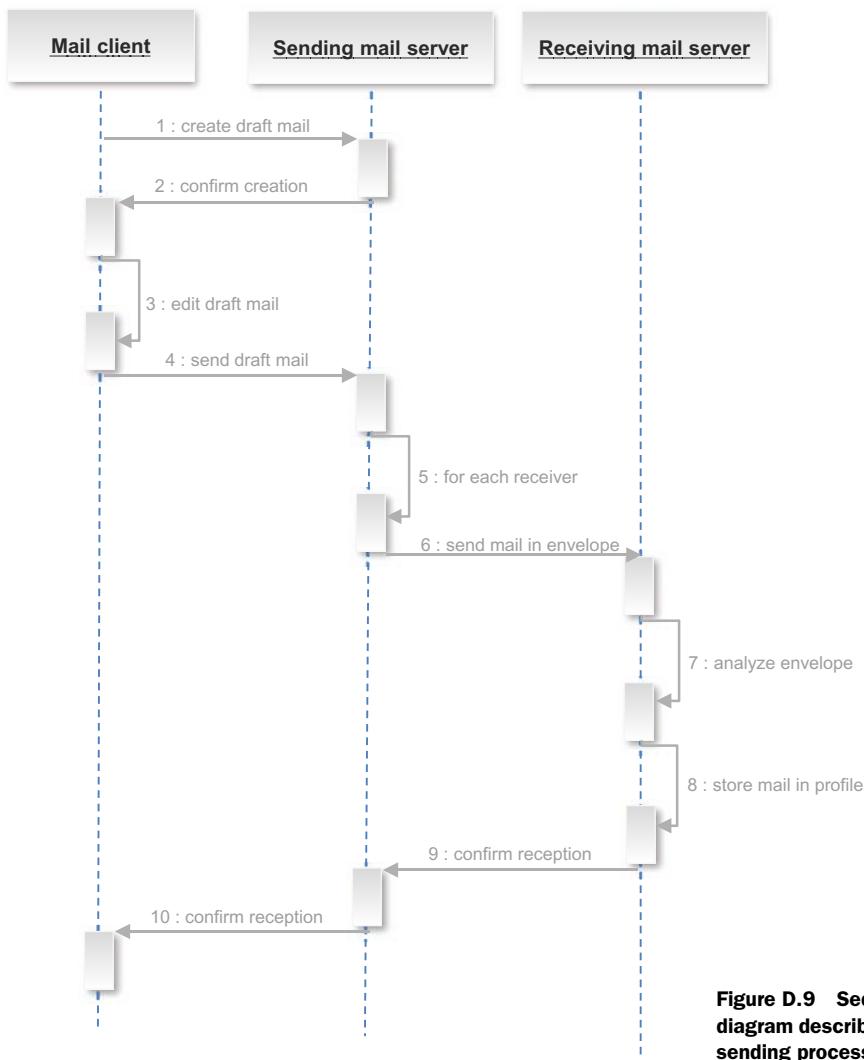
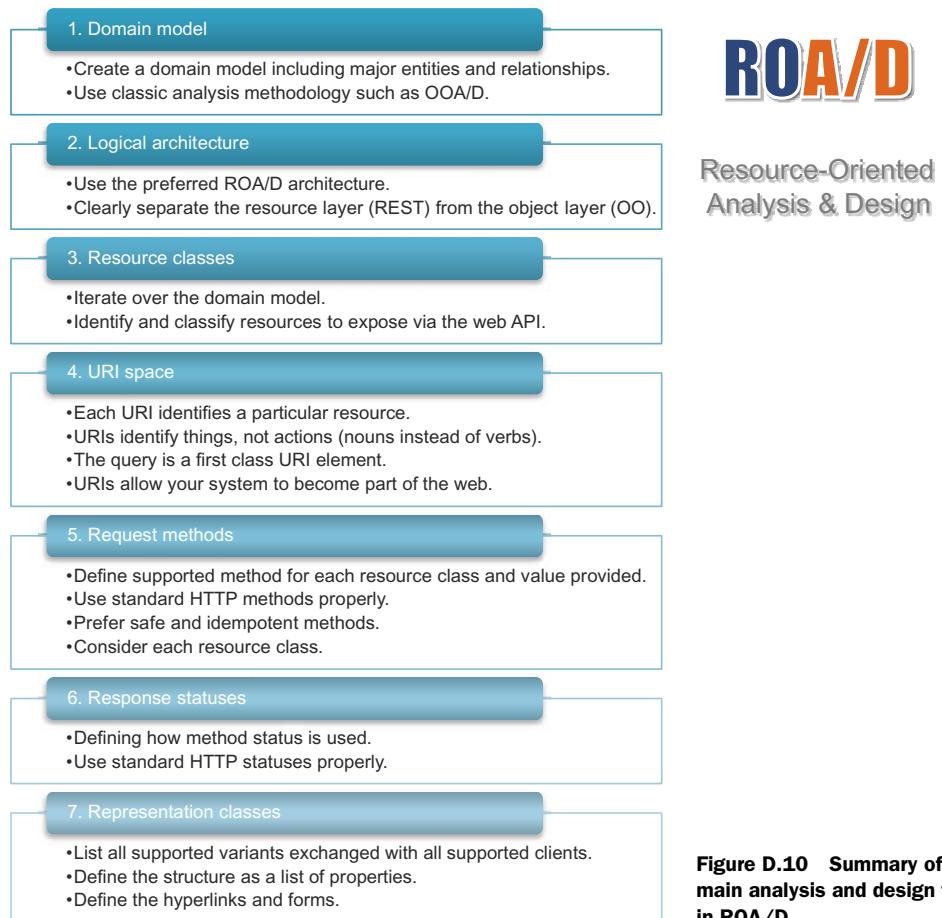


Figure D.9 Sequence diagram describing the email sending process

design guidelines (summarized in figure D.10) and show how they can be applied to the example mail system (interleaving design guidelines with application of these guidelines to our example).

D.4.1 Defining the logical architecture

Your first design task is to define a coarse-grained architecture of your solution. At this point we assume that you chose to develop a RESTful web project using an object-oriented language such as Java. Figure D.11 proposes the preferred logical architecture to use when applying the ROA/D methodology. It's composed of four layers building on each other as suggested by the pyramidal layout.



Resource-Oriented
Analysis & Design

Figure D.10 Summary of main analysis and design tasks in ROA/D

At the top you have the UI layer supporting the interactions with users—for example, via a web browser or a programmatic HTTP client. To respect REST principles, those user interfaces must use hypermedia to animate the application and change its state. If the user is a human, it should rely on hypertext displayed in a browser or similar environment, letting the user follow hyperlinks and send forms to interact with the system. If the user is a robot—a programmatic client—then hypermedia still applies and becomes hyperdata, supported by languages such as XML, JSON, or RDF.

Below the UI layer is the *resources layer* that's manipulated by transferring representations back and forth, invoking HTTP methods on the resources. In this layer, typically implemented using a REST framework such as Restlet, it's essential to respect another REST principle that says no state related to client interactions should be maintained—for example, in the form of session objects. The responsibilities of this layer are the identification of resources by URIs, the formatting and parsing of representations of those resources, and the routing of calls or security enforcement.

You could attempt to directly persist those resources into a data store, but in many cases introducing another layer is preferable. The *objects* layer lets you model more easily and extensively the complexity of your domain model and the associated logic. For example, you can define domain-specific object methods and have all the power of object-oriented programming.

The *data* layer supports the persistence of the objects layer and indirectly of the resources layer. Various types of technologies are available, the most popular being the file system and databases (cloud-scale databases such as Cassandra, relational databases such as MySQL, and object databases such as db4o). You can use an Object-Relational Mapping (ORM) tool such as Hibernate to automatically persist your object model into a relational database.

To summarize, each layer builds on top of the other, and the lower-level layers have no knowledge of layers above them, favoring separation of concerns and reusability. The highest layers are the most specific to a given application, whereas the lowest are the most generic and reusable.

Each layer adds its unique features and advantages, relying on the best paradigm for the job at hand. This is illustrated in figure D.12 with another architecture diagram proposing a more topological view, showing the constitution of a server component in nested layers, interacting with a client component.

In the rest of the book, we apply this architecture to the email system example:

- For the UI layer, we rely on your preferred web browser and on a programmatic client for Android mobiles.
- For the resources layer, we use the Restlet Framework, which is the central topic of this book.
- The objects layer is built using Plain Old Java Objects (POJOs) and persisted into the data layer.
- For the data layer we use in-memory persistence for simplicity, but in a more realistic situation we would use a proper database technology.

D.4.2 *Deriving the resource model*

Now that you have a clearer picture of the target architecture and the main technological choices, it's time to find out how you derive the initial domain model produced during the requirements analysis step into a resource model for the resources layer and into an object model for the objects layer. The production of a design object model from a conceptual domain model is a topic that has been extensively documented in the

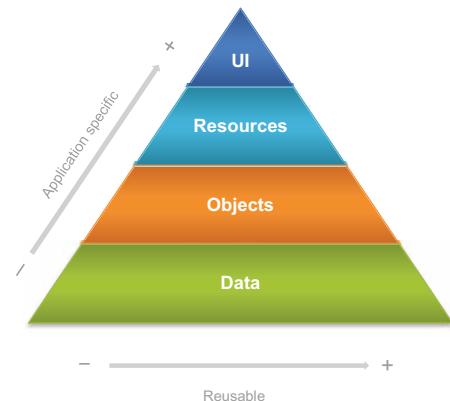


Figure D.11 Layers of the ROA/D logical architecture

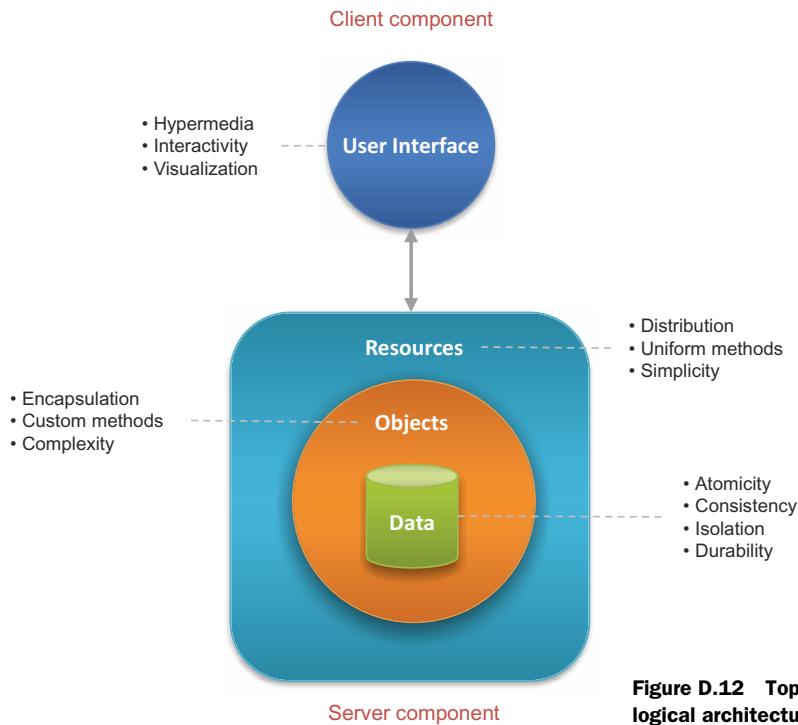


Figure D.12 Topology of the ROA/D logical architecture

OOA/D methodology. We recommend that you consult a dedicated book [32] if you need assistance in this area.

In addition, if you want to manually derive a relational model from our domain model, you also have methodologies like the IDEF1 standard [33] at hand. ORM tools can generate the relational model from the object model for you, or even the other way around.

The remaining question is the derivation of the resource model from the domain model. This is the main specificity of our ROA/D methodology as illustrated in figure D.13.

Resource-Object Mapping (ROM) shown in figure D.13 is the equivalent of ORM but between resources and objects layers. As you see in the book's chapters, the Restlet Framework greatly simplifies this mapping.

For now, let's detail the tasks to complete in order to derive the domain model into the resource model.

D.4.3 Identifying and classifying the resources

At this point in the design of your RESTful web API, you should identify the resources you'll expose and classify them. Note that we use the terms class, type, and category interchangeably in their most common sense: a group of things that have similar characteristics.

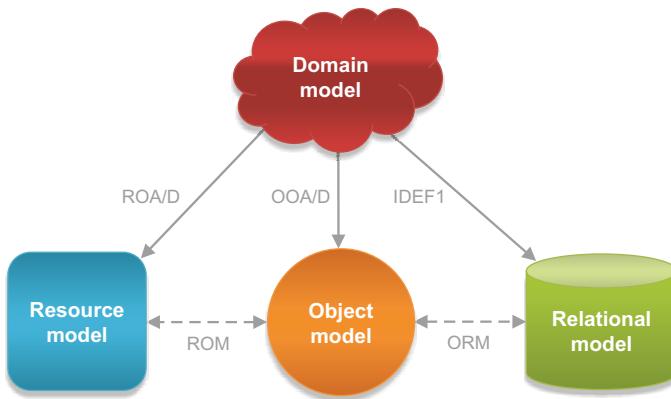


Figure D.13 Deriving the domain model

In general, because a resource can wrap any concept, listing all the potential types that you can find in applications is difficult. But you typically find entity resources, corresponding to domain entities in the domain model, collection resources to manage a set of entity resources, and also service resources focusing on processes such as form submission acceptors and search query processors.

Besides this general typology, the most important question to ask is what information from your domain model needs to be exposed via the web API, and in which form? For sure, any class of objects in the domain model that should be publicly accessible should correspond to a class of resources. As a matter of simplicity, the object class and resource class should be given similar names.

Relations with *n-n* cardinality between two kinds of objects (like the “receivers” relation between the Contact and Mail object classes) are also good candidates to become resource classes. Resources that act as containers of other resources should produce a collection resource class and an item resource class. For example, if an application’s main role is to manage mail messages, you should have a “Mails” collection resource class and a “Mail” entity resource class. Public methods of façade domain classes are also candidates to become service resource classes.

Note that all domain categories of objects don’t need to be directly accessible and therefore won’t become resource types, but may instead become part of a representation class for a parent resource. Now, let’s go back to the design of the RESTful email system. Based on our domain model, we easily identify the following resource classes:

- *Root*—Top resource of the RESTful mail system
- *Account*—Holder of contacts, mails, and feeds
- *Contact*—Holder of information on a known person
- *Mail*—Message exchanged by users via their accounts
- *Feed*—Selection of mails based on matching tags

You may have noticed that the Tag class of the domain model isn’t part of this list. In fact, we considered that even though it could be thought of as a resource, there isn’t

enough interest in the example for it. We handle them instead using simple text tokens stored as properties of the related classes Mail and Feed.

Among this initial list of resource classes, only Root has a single occurrence. For all others we need to introduce collection resources to manage the creation (POST and PUT methods) and the listing (via GET method) of contained resources. This gives us these additional resource classes:

- *Accounts*—Container of account resources
- *Contacts*—Container of contact resources in a given account
- *Mails*—Container of mail resources in a given account
- *Feeds*—Container of feed resources in a given account

When in doubt regarding a potential resource, ask yourself: do I need or want this potential resource to be accessible on the web, given a public URI?

D.4.4 Defining the URI space

As discussed in chapter 1, a resource is anything of interest that a server decides to expose and to uniquely identify with a URI. When you surf the web, each web page is a resource. Your browser typically retrieves its representation by using its URL, which is a kind of URI, along with the GET method. The URL contains enough information for the network infrastructure to find the computer hosting the resource and to identify the resource on the computer.

EACH URI IDENTIFIES A PARTICULAR RESOURCE

To adopt this model you must decide what resources your server-side program exposes and assign each one a URI. As you saw in the previous section, resources are typically application-level entities you want to expose to remote clients. For example, if your application provides weather information, you may want to expose a report about the current weather in Paris at <http://www.myGreatWeatherApp.com/Paris/>. The current weather in Chicago could be described at <http://www.myGreatWeatherApp.com/Chicago/>. And so forth. Or maybe you want to provide finer-grained access to weather information and define separate resources for temperature, hygrometry, and so on. The current hygrometry in Paris could be identified by <http://www.myGreatWeatherApp.com/Paris/hygrometry> and consist of a number.

Maybe providing weather information for whole cities isn't all you want to do. Your application might use a more precise and global location system like the WGS84 (World Geodetic System), which is used by GPS. In that case you may decide to create a resource space that associates a weather report to each place on earth, localized by its coordinates. The current weather under the Eiffel tower could be at <http://www.myGreatWeatherApp.com/48.85834817715778,2.294543981552124>.

CRAFTING YOUR URIs

Each resource instance must be uniquely identifiable using a URI. The first thing is to choose the base URI of your application that you associate to a root resource—for example, <http://www.mydomain.org/myApplication/v1/>. Note that the appended /v1/ lets

you serve several versions of the same application at the same time in the future, facilitating the migration from one version to another without breaking existing clients.

HTTP URIs—or URLs to be more precise—support a hierarchical naming system. The hierarchy is denoted by the / character inserted between adjacent elements, also called *segments*. This arrangement encourages you to organize your resources into a hierarchy, according to your own rules. The URI of each resource class is generally the concatenation of the base URI of your application with a resource class name and a trailing slash character (except for leaf resources, which can't have any child resource). Here's the URI for our example Accounts resource class: `http://localhost:8111/accounts/`.

For each root you should append to the parent URI, and as you dive deeper inside the hierarchy append a leading slash and the chosen name of the current level. For each child resource class append a constant name for nonrepeating resources or a unique identifier, followed by a trailing slash if children exist. Because listing all potential values of the identifier would be cumbersome, we instead use URI variables like {myId}, following the URI Templates standard [34].

For example, an Accounts resource class has the following URI template: `http://localhost:8111/accounts/{accountId}`. This is continued recursively for each child resource class until we reach the leaves. Figure D.14 illustrates the example resource classes using the UML class diagram. We tried to use a layout that reproduces the URI hierarchy like nested folders and added UML notes as URI hints.

Choosing URIs for your resources is always an important design task. They're part of the UI of your application because end users do see and manipulate them. Assume that URIs identify concepts and make them as user-friendly as possible.

In the sample mail application, because we aren't planning on hosting it with a registered domain name, we use the generic localhost domain as the base URI: `http://localhost:8111/`.

Note that leaf resources don't have a trailing slash character to reinforce the fact that they don't contain anything. Again, this is purely a convenience that helps users of your application, but it's not an aspect that determines whether or not your design is RESTful.

URIs IDENTIFY THINGS, NOT ACTIONS

When designing your URIs, keep in mind that they identify things, not actions. The action a client asks a server to perform is expressed using the HTTP method, and the request URI denotes the primary target of this action. The presence of verbs in URIs is often a good indicator of a problem in the design of a RESTful web API. If you come up with URIs that contain verbs in imperative form, you might be trying to encode requested actions inside your URIs. Double-check your design and make sure you aren't bypassing, or reinventing, the HTTP method system (see section D.4.5).

Verbs that aren't used to request actions from the server can be perfectly valid URI elements. In the online English dictionary service provided by Merriam-Webster, the

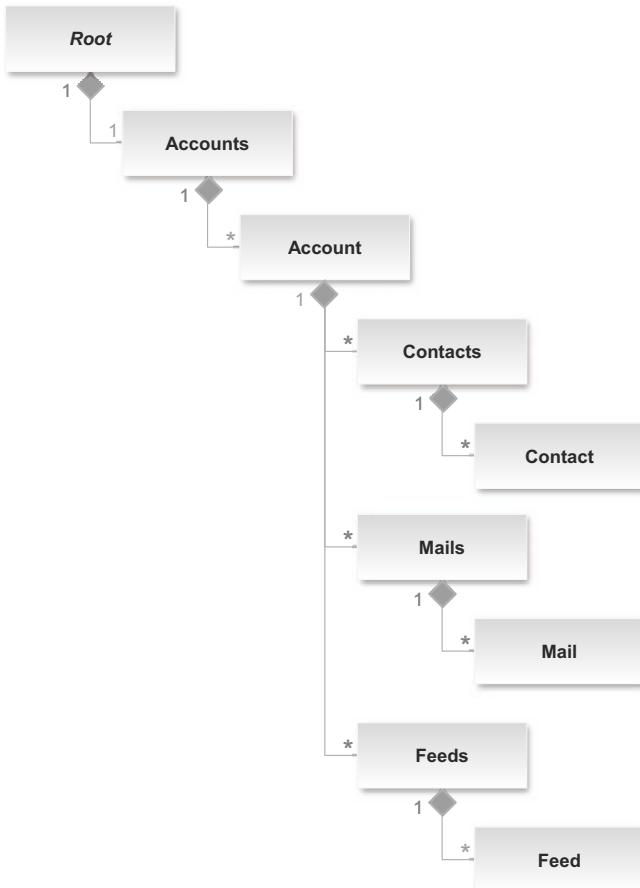


Figure D.14 Example resources hierarchy as a UML class diagram

URI for the definition of the verb *compute* is <http://www.merriam-webster.com/dictionary/compute>. There are also cases where verbs in URIs are fine even if they describe actions to be performed by the server.

THE QUERY IS A FIRST-CLASS URI ELEMENT

The result of a Google search for the word *Restlet* has the following URI: <http://www.google.com/search?q=Restlet>. This is in line with REST and HTTP because this URI, as a whole, still references a thing: the result of searching for the word *Restlet*. Indeed, things defined as being the result of some action are still things. This thing is still manipulated using HTTP methods; in this example you can get its representation using the GET method. That's what your web browser does when you use the Google website.

On the other hand, the AWS SimpleDB API uses POST methods on URIs, such as <https://sdb.amazonaws.com/?Action=DeleteDomain&DomainName=MyDomain> (the actual URI has additional query parameters that were removed for clarity) that should have been expressed as DELETE methods on a URI, like <https://sdb.amazonaws.com/domains/MyDomain>.

The Google search example also answers a common question about the status of the query string: is the query an integral part of the URI and consequently participates in resource identification? The answer is yes, as `http://www.google.com/search?q=Restlet` and `http://www.google.com/search?q=pumpkin` refer to two different resources, each the result of a different search. The query is an integral part of the URI, like other path components (`/search`, for example). Still, you can use different URIs to refer to the same resource if needed, but that's a general ability of URIs, not something related to the use of the query string in particular. Remember that the query is a first-class URI element and that two URIs that differ only in the query can still refer to two different resources.

URIs ALLOW YOUR SYSTEM TO BECOME PART OF THE WEB

In effect, URIs are pointers. You can use them wherever needed to refer to resources. But they differ from pointers in traditional programming languages in one key aspect: a pointer in a C program, or a reference to an object in Java, is meaningful only inside the application in which it exists. A URI is meaningful for everyone, for every system.

With URIs, pointers break the application boundaries. Your resources can be referenced by other systems, and you can reference resources provided by others. Your resources can be accessed and processed by a number of tools such as web browsers, search engines, and so on. Exposing resources identified by URIs allows your system to be part of the web and to enrich it.

D.4.5 Defining allowed methods

With HTTP, resources are manipulated through a few predefined operations, formally called *methods*. Each HTTP request message sent by a client includes the name of a method. HTTP defines the semantics of these methods at a generic level (see table D.1). Your application then specializes the meaning of each method, for each resource or execution context.

Table D.1 Four main HTTP methods and what the client is asking the server to do when using them

Method	What the client is asking	Type
GET	Give me back a representation of the target resource.	Safe and idempotent
PUT	Create or update the target resource with the representation I'm sending you in the body of this request.	Nonsafe and idempotent
POST	Make the target resource process the data I'm sending to you in the body of this request.	Nonsafe and nonidempotent
DELETE	Delete the target resource and its associated state.	Nonsafe and idempotent

HTTP states that the DELETE method requests that the target resource be deleted from the server. The exact meaning of such an operation is then determined by each application. In our email application, deleting a resource that represents an account could remove the account from the system, including all related information in the database, such as the contained contacts and mails.

USE HTTP METHODS PROPERLY

To benefit fully from REST and HTTP, you must use HTTP methods according to the semantics mandated by HTTP. For example, the GET method is for information retrieval: getting the representation of a resource. You must not use it to express other kinds of interactions. To characterize methods, HTTP defines two important properties: *safe* and *idempotent*:

- Safe methods are, from the point of view of the client, for pure information retrieval. If they have side effects, these effects must be such that the client can't be held accountable for them, as it did not request them.
- Idempotent methods are such that (aside from error or expiration issues) the side effects of multiple identical requests are the same as a single request. DELETE is idempotent because a sequence of two or more identical DELETE requests has the same effect on the server as one DELETE request: in both cases, the target resource is deleted. Trying to delete something that has already been deleted doesn't change the outcome.

By these definitions, safe methods have the potential to be invoked automatically by some generic code on the client side (web crawlers make use of this) whereas nonsafe methods require explicit triggering by the end user or the client-side application programmer and awareness of the possible side effects.

Requests with idempotent methods have the potential to be automatically reemitted by the network infrastructure when in doubt over correct reception by the server, without fear of unwanted effects caused by multiple receptions of the request.

In addition to the four methods described in table D.1, HTTP also define an OPTIONS method to describe the communication options available on the target resource. In particular, it allows asking a resource for the list of methods it supports. A given resource might not support all the HTTP methods. For example, you may not want to allow remote clients to DELETE certain resources you expose. HTTP also defines the HEAD and TRACE methods. You'll find a complete description of these methods in the HTTP specification [22].

TIP You shouldn't regard HTTP as a low-level protocol an application developer doesn't have to bother with. On the contrary, HTTP is the high-level protocol with which you're going to directly express the interactions with your applications over the network. REST brings a set of principles that helps you make good use of HTTP, and the Restlet Framework puts all this in motion in your Java programs. Yes, HTTP is your new friend! We advise you to grab the HTTP specification [22], become familiar with it, and keep it at hand. You can also help yourself by reading a good book about HTTP such as *HTTP: The Definitive Guide* by David Gourley and Brian Totty (O'Reilly, 2002). Along with the Restlet documentation, these are going to be some of your main resources for everything related to web development.

When possible, try to use the HTTP methods whose semantics are the most precise. In particular, use the POST method only if other HTTP methods aren't a good fit. As

specified by HTTP, POST has rather unbounded semantics. It means: “Take the data I’m sending to this target resource and process it.” As such, you can use it for any kind of operation, because you have a lot of freedom to define what “process it” means in the context of your application. But if an operation you want to expose on one of your resources fits the more specific semantics of GET, PUT, DELETE, or other HTTP method, you should use that method instead. At run time the HTTP infrastructure (composed of the HTTP clients, caches, and intermediary proxies) has more information about what the remote client is doing with your resources and can better handle the situation. It can automatically resend requests using idempotent methods in case of communication error. It can also cache results of the execution of the GET method and avoid sending some requests over the network, which can greatly improve performance and decrease load on servers.

CONSIDER EACH RESOURCE CLASS

Let’s apply what you’ve learned to our example. In section D.4.4 we defined a complete hierarchy of resource classes with associated URI templates. We now need to define for each resource class what the allowed methods are. Assuming we use the HTTP protocol, the common list of methods is generally: GET, POST, PUT, DELETE, and sometimes OPTIONS. You can also use extension methods in some rare cases, such as WebDAV’s MOVE method—but be careful because this will limit the accessibility of your web API, as many HTTP clients only support basic HTTP methods.

To facilitate the communication of your resource classes design among your team, we recommend creating detailed UML class diagrams containing sets of related resource classes. Figure D.15 refines the class diagram introduced in figure D.13 using an additional compartment below the class name to indicate the relative URI part, as allowed by UML. It also adds the HTTP methods supported.

Note that the Accounts resource class uses a POST method to support the creation of child accounts because you can’t know in advance the URI of those accounts and therefore can’t directly use PUT to create them.

We leave you as an exercise the creation of a class diagram for the remaining example resources, children of Accounts: Contacts, Contact, Mails, Mail, Feeds, and Feed.

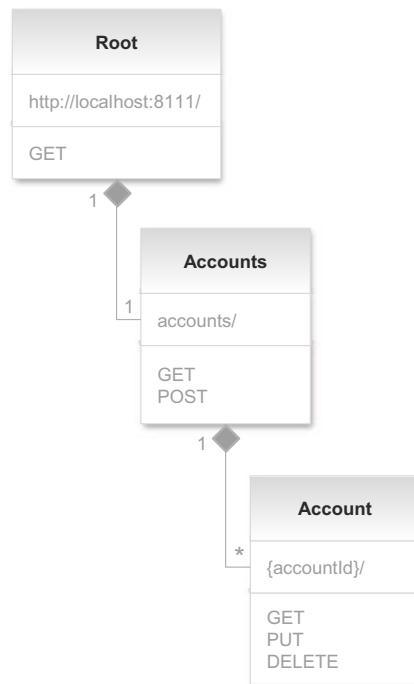


Figure D.15 Example resources hierarchy as a UML class diagram describing allowed methods

D.4.6 Defining response statuses

There's an important bit of information that's found in every HTTP response message: a status code. Using this code your server-side program can communicate crucial information to a client. The well-known code 404 means that the resource target has not been found, and consequently, the server wasn't able to further process the request.

HTTP standardizes several important things here:

- The way to denote the status code in the response message, along with an associated human readable *reason phrase*
- The status categories, also called status classes, such as *Successful*, *Redirection*, *Server error*, and *Client error*
- A number of predefined status codes, with specified semantics
- A way to create new status codes if needed

There are codes to signal various errors conditions, action required from the client to complete its interaction with the server, or other important information about the outcome of the request. A few examples are in table D.2. For a complete list, see table E.2 in appendix E.

Table D.2 Examples of common status codes defined by HTTP

Code	Category	Name	Meaning
200	Successful	Success	The request has succeeded.
201	Successful	Created	The request has been fulfilled and resulted in a new resource being created. The URI of this new resource should be present in the response message, in the Location header field.
202	Successful	Accepted	The request has been accepted for processing, but the processing has not been completed. The response message should include an indication of the request's current status and either a pointer to a status monitor or some estimate of when the client can expect the request to be fulfilled.
301	Redirection	Moved Permanently	The requested resource has been assigned a new permanent URI, and any future references to this resource should use this URI. The new URI should be present in the response message, in the Location header field.
400	Client error	Bad Request	The request could not be understood by the server due to malformed syntax. The client should not repeat the request without modifications.
404	Client error	Not Found	The server has not found anything matching the request-target.
405	Client error	Method Not Allowed	The method specified in the Request-Line isn't allowed for the target resource. The response includes an Allow header containing a list of valid methods for the requested resource.
500	Server error	Internal Server Error	The server encountered an unexpected condition which prevented it from fulfilling the request.

Having a whole set of status codes already defined and ready to use is useful. Rather than reinvent the wheel, you can—and should—use these standardized codes. On the server side, you should strive to return status codes that accurately describe the outcome of the request. On the client side, you should check the status code of each response you receive and act accordingly. The HTTP specification gives you information on how to use status codes correctly, and the Restlet Framework helps you deal with them easily in Java, including automatically acting on some of them (for example, automatically using the redirection information on the client side to reemit a request if the target resource has moved).

D.4.7 *Defining representation classes*

Some HTTP requests or responses contain resource representations. For example, a response to a GET request typically includes the representation of the target resource. As discussed in chapter 1, a representation is some data describing the current or intended state of a resource. Obviously, an important task when designing your resource model is to define the format of these representations (such as an HTML document or an XML document with a given structure).

Each resource class can have multiple types of representation (HTML can be used when interacting with web browsers, and XML when interacting with purely programmatic clients). These types of representation, called representation variants, are used by a powerful mechanism of HTTP called content negotiation (see section 4.5 for more on this).

You should design and document the structure of each variant. For each variant you should indicate the media type, language, character set, and encoding if applicable. For some media types, such as XML, providing an XML Schema is also useful. This should be completed by textual descriptions if necessary.

In our example email system, the Root resource is the main entry point for users. It will be accessed by web browsers and return a welcome HTML page. When the user tries to access this page for the first time, with a GET request, an HTTP Basic authentication mechanism prompts for a login and password. If the credentials are valid, the user will be redirected to their account. If the user is an administrator, they instead see a hyperlink to the Accounts resource.

The Accounts resource returns an HTML page that contains a list of accessible Account resources. If the authenticated user is only an owner, they will only see their account. Administrators will see the list of all accounts. In addition, administrators see an HTML form allowing the creation of new accounts. When a creation is requested, a POST method is invoked, resulting in a redirection to the created Account. The Accounts resource returns an HTML page that contains a list of accessible Account resources. Administrators see the list of all existing accounts. In addition, administrators see an HTML form allowing the creation of new boxes. Simple owner users can't create new boxes by themselves; they have to ask an administrator. The Account resource returns an HTML page with a form containing the properties (first name, last

name, role, and nickname). This form can be used to update or delete the current account. The HTML page also contains hyperlinks to the related Contacts, Mails, and Feeds resources. And the resource can receive remote mails in XML format via its POST method.

The Contacts resource returns an HTML page that contains a list of Contact resources contained in the parent MailBox. An HTML form allows the creation of new contacts. When a creation is requested, a POST method is invoked, resulting in a redirection to the created Contact. The Contact resource returns an HTML page with a form containing all the information related to the contact (first name, last name, and URI address). This form can be used to update or delete the current contact.

The Mails resource returns an HTML page that contains a list of Mail resources contained in the parent MailBox. An HTML form allows the creation of new mails. When a creation is requested, a POST method is invoked, resulting in a redirection to the created Mail. The Mail resource returns an HTML page with a form containing all the information related to the mail (subject, receivers, body text, and tags). This form can be used to update or delete the current mail. This resource is also capable of sending the mail to the receiver by posting it to the receiver's URI address, corresponding to the receiver's account resource.

The Feeds resource returns an HTML page that contains a list of Feed resources contained in the parent MailBox. In addition, an HTML form allows the creation of new feeds. When a creation is requested, a POST method is invoked, resulting in a redirection to the created Feed. The Feed resource can return an HTML page to web browsers with a form containing all the information related to the web feed (name and tags to match) and the list of matched mails. This form can be used to update or delete the current feed. This resource can also return an Atom representation for feed readers, using HTTP content negotiation.

That should give you a good description of the behavior of our mail application from a web browser's point of view. It's not too different for a programmatic client, except that the representation formats use XML or JSON instead of HTML and web forms. If you want to visually describe this important design information, you can use UML state chart diagrams, which are a natural fit for hypermedia, UML sequence, and activity diagrams.

Figure D.16 enriches the previous class diagrams with information related to the supported representations. We use a special `redirect` keyword to indicate that a redirection is issued, including the proper status code and the target URI reference to redirect to (communicated as a special Location HTTP header).

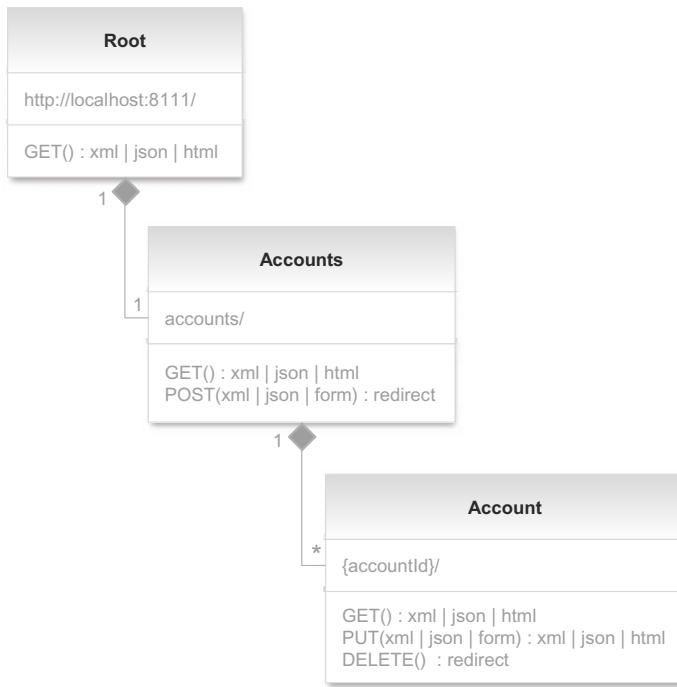


Figure D.16 Example resources hierarchy as a UML class diagram describing representations

appendix E

Mapping REST, HTTP, and the Restlet API

This appendix is meant as a reference guide for the developer of Restlet applications. The first section lists key classes, properties, and constants used by the Restlet API and explains how they map to REST concepts, HTTP concepts, and headers.

Then we list all client and server connectors as well as the automatic representation converters available in Restlet Framework version 2.1. And we discuss the supported security challenge schemes and the authorities specific to the RIAP and CLAP pseudoprotocols.

E.1 Mapping REST concepts to Restlet classes

Table E.1 lists the main concepts of the REST architecture style and indicates the classes or interfaces of the Restlet API that materialize them.

Table E.1 Mapping REST concepts to Restlet classes

REST concept	Restlet class or interface	Description
Client	<code>org.restlet.Client</code>	Connector acting as a generic client. It internally uses one of the available connector helpers registered with the Restlet Engine.
Component	<code>org.restlet.Component</code>	Restlet managing a set of connectors, virtual hosts, services, and applications. Applications are expected to be directly attached to virtual hosts or to the internal router. Components also expose several services: access logging and status setting.
Connector	<code>org.restlet.Connector</code>	Restlet enabling communication between components. It can be either a client or server.

Table E.1 Mapping REST concepts to Restlet classes (continued)

REST concept	Restlet class or interface	Description
Metadata	org.restlet .data.Metadata	Representation metadata for content negotiation such as media type, language, character set, and encoding.
Representation	org.restlet .representation .Representation	Current or intended state of a resource. The content of a representation can be retrieved several times if there's a stable and accessible source, like a local file or a string. When the representation is obtained via a temporary source like a network socket, its content can be retrieved only once.
Resource	org.restlet .resource .Resource	Base resource class exposing the uniform REST interface. Intended conceptual target of a hypertext reference. A uniform resource encapsulates a context, a request, and a response, corresponding to a specific target resource.
Server	org.restlet .Server	Connector acting as a generic server. It internally uses one of the available connector helpers registered with the Restlet Engine.
Uniform interface	org.restlet .Uniform	Uniform REST interface that defines the standard way to communicate between components via connectors. It corresponds to standard HTTP methods and to standard representation media types such as HTML, RDF, and Atom/AtomPub.

E.2 *Mapping HTTP concepts to Restlet classes*

Table E.2 describes the mapping between HTTP concepts and Restlet classes or interfaces.

Table E.2 Mapping HTTP concepts to Restlet classes

HTTP concept	Restlet class or interface	Definition from dissertation
Message	org.restlet .Message	Generic message exchanged between components, either a request or a response.
Request	org.restlet .Request	Generic request sent by client connectors. It's received by server connectors and processed by Restlet. Requests are uniform across all types of connectors, protocols, and components.
Response	org.restlet .Response	Generic response sent by server connectors and received by client connectors. Responses are uniform across all types of connectors, protocols, and components.
Variant	org.restlet .representation .Variant	Descriptor for available representations of a resource. It contains all the important metadata about a representation but isn't able to serve the representation's content itself.

E.3 Mapping HTTP headers to Restlet properties

The HTTP protocol is the main source of inspiration for the Restlet API, which was designed as a high-level abstraction of the HTTP protocol. On the other hand, many developers refer to HTTP headers and need a tool to help them map the Restlet API properties to those HTTP headers. Providing that tool is the aim of table E.3.

By convention, the list of property names refers to Restlet classes such as Request, Response, or Message. The “Restlet property name” column contains values such as `request.clientInfo.acceptedMediaTypes` and `response.age`, referring to Java properties of the message classes. Those properties are accessible using the getter and setter methods, such as `Request.getClientInfo().getAcceptedMediaTypes()` and `Response.getAge()`.

Table E.3 Mapping HTTP headers to Restlet properties

Header	Restlet property name	Restlet property class	Description
Accept	<code>request.clientInfo.acceptedMediaTypes</code>	<code>List<org.restlet.data.Preference<MediaType>></code>	The list of media types accepted by the client.
Accept-Charset	<code>request.clientInfo.acceptedCharacterSets</code>	<code>List<org.restlet.data.Preference<CharacterSet>></code>	The list of character sets accepted by the client.
Accept-Encoding	<code>request.clientInfo.acceptedEncodings</code>	<code>List<org.restlet.data.Preference<Encoding>></code>	The list of encodings accepted by the client.
Accept-Language	<code>request.clientInfo.acceptedLanguages</code>	<code>List<org.restlet.data.Preference<Language>></code>	The list of languages accepted by the client.
Accept-Ranges	<code>response.serverInfo.acceptingRanges</code>	<code>boolean</code>	Allows the server to indicate its support for range requests.
Age	<code>response.age</code>	<code>int</code>	The estimated amount of time since the response was generated or revalidated by the origin server.
Allow	<code>response.allowedMethods</code>	<code>Set<org.restlet.data.Method></code>	Indicates the set of allowed methods. Can be retrieved with an OPTIONS call.

Table E.3 Mapping HTTP headers to Restlet properties (*continued*)

Header	Restlet property name	Restlet property class	Description
Authentication-Info	response.authentication-Info	org.restlet.data.AuthenticationInfo	Authentication information sent by an origin server to a client after a successful authentication attempt.
Authorization	request.challengeResponse	org.restlet.data.ChallengeResponse	Credentials that contain the authentication information of the user agent for the realm of the resource being requested.
Cache-Control	message.cacheDirectives	List<org.restlet.data.CacheDirective>	List of directives that must be obeyed by all caching mechanisms along the request/response chain.
Content-Disposition	message.entity.disposition	org.restlet.data.Disposition	Means for the origin server to suggest a default filename if the user requests that the content be saved to a file.
Content-Encoding	message.entity.encodings	List<org.restlet.data.Encoding>	Indicates what additional content encodings have been applied to the entity-body.
Content-Language	message.entity.languages	List<org.restlet.data.Language>	Describes the natural language(s) of the intended audience for the enclosed entity.
Content-Length	message.entity.size	long	The size of the entity-body, in decimal number of OCTETs.

Table E.3 Mapping HTTP headers to Restlet properties (*continued*)

Header	Restlet property name	Restlet property class	Description
Content-Location	message.entity.locationRef	org.restlet.data.Reference	Indicates the resource location for the entity enclosed in the message.
Content-MD5	message.entity.digest	org.restlet.data.Digest	Value and algorithm name of the digest associated with a representation.
Content-Range	message.entity.range	org.restlet.data.Range	Indicates where in the full entity-body the partial body should be applied.
Content-Type	message.entity.mediaType and characterSet	org.restlet.data.MediaType + CharacterSet	Indicates the media type of the entity-body.
Cookie	request.cookies	Series<org.restlet.data.Cookie>	List of one or more cookies sent by the client to the server.
Date	message.date	Date	The date and time at which the message originated.
ETag	message.entity.tag	org.restlet.data.Tag	The current value of the entity tag for the requested variant.
Expect	request.clientInfo.expectations	List<org.restlet.data.Expectation>	Indicates that particular server behaviors are required by the client.
Expires	message.entity.expirationDate	Date	The date/time after which the response is considered stale.
From	request.clientInfo.from	String	The email address of the human user controlling the user agent.

Table E.3 Mapping HTTP headers to Restlet properties (*continued*)

Header	Restlet property name	Restlet property class	Description
Host	request.hostRef	Reference	Specifies the internet host and port number of the resource being requested.
If-Match	request.conditions.match	List<org.restlet.data.Tag>	Used with a method to make it conditional.
If-Modified-Since	request.conditions.modifiedSince	Date	Used with a method to make it conditional.
If-None-Match	request.conditions.noneMatch	List<org.restlet.data.Tag>	Used with a method to make it conditional.
If-Range	request.conditions.rangeTag and rangeDate	org.restlet.data.Tag + Date	Used to conditionally return a part or the entire resource representation.
If-Unmodified-Since	request.conditions.unmodifiedSince	Date	Used with a method to make it conditional.
Last-Modified	message.entity.modificationDate	Date	Indicates the date and time at which the origin server believes the variant was last modified.
Location	response.locationRef	org.restlet.data.Reference	Used to redirect the recipient to a location other than the Request-URI for completion of the request or identification of a new resource.
Max-Forwards	request.maxForwards	int	Maximum number of proxies or gateways that can forward the request to the next inbound server.

Table E.3 Mapping HTTP headers to Restlet properties (*continued*)

Header	Restlet property name	Restlet property class	Description
Proxy-Authenticate	response.proxyChallengeRequests	List<org.restlet.data.ChallengeRequest>	Indicates the authentication scheme(s) and parameters applicable to the proxy.
Proxy-Authorization	request.proxyChallengeResponse	org.restlet.data.ChallengeResponse	Credentials that contain the authentication information of the user agent for the proxy.
Range	request.ranges	List<org.restlet.data.Range>	List of one or more ranges to return from the entity.
Referer	request.refererRef	Reference	The address (URI) of the resource from which the Request-URI was obtained.
Retry-After	response.retryAfter	Date	Indicates how long the service is expected to be unavailable to the requesting client.
Server	response.serverInfo.agent	String	Information about the software used by the origin server to handle the request.
Set-Cookie	response.cookieSettings	Series<org.restlet.data.CookieSetting>	List of one or more cookies sent by the server to the client.
Set-Cookie2	response.cookieSettings	Series<org.restlet.data.CookieSetting>	List of one or more cookies sent by the server to the client.
User-Agent	request.clientInfo.agent	String	Information about the user agent originating the request.

Table E.3 Mapping HTTP headers to Restlet properties (*continued*)

Header	Restlet property name	Restlet property class	Description
Vary	response .dimensions	Set<org.restlet .data.Dimension>	Indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without revalidation.
Via	message .recipientsInfo	List<org.restlet .data .RecipientInfo>	Used by gateways and proxies to indicate the intermediate protocols and recipients between the user agent and the server on requests, and between the origin server and the client on responses.
Warning	message .warnings	List<org.restlet .data.Warning>	Additional warning information.
WWW-Authenticate	response. challengeRequests	List<org.restlet .data .ChallengeRequest>	Indicates the authentication scheme(s) and parameters applicable to the Request-URI.
X-Forwarded-For	request.clientInfo .forwarded- Addresses	List<String>	The list of client IP addresses, including intermediary proxies.
X-HTTP-Method-Override	tunnelService .methodHeader	org.restlet.data .Method	Overrides the HTTP method for limited clients such as browsers.

E.4 Available connectors

As mentioned in chapter 1, connectors are an essential part of the REST architectural style. They enable communication between components by implementing a particular network protocol.

Because of the distinction between client and server components, this section contains two tables: one for the server connectors and one for client connectors as available in Restlet Framework version 2.1. Table E.4 lists the server connectors ordered by protocol name. Those connectors are available either in the core Restlet module (`org.restlet`) or in extension modules (`org.restlet.ext.<moduleName>`).

Table E.4 Server connectors

Protocol	Module
AJP	jetty
HTTP	core, jetty, simple
HTTPS	ssl, jetty, simple
RIAP	core
SIP	sip
SIPS	sip

Table E.5 lists all the client connectors available.

Table E.5 Client connectors

Protocol	Module
CLAP	core
FILE	core
FTP	net
HTTP	core, httpclient, net
HTTPS	httpclient, net, ssl
JDBC	jdbc
POP (v3)	javamail
POPS (v3)	javamail
RIAP	core
SDC	sdc
SIP	sip

Table E.5 Client connectors (continued)

Protocol	Module
SIPS	sip
SMTP	javamail
SMTPS	javamail
SOLR	lucene
ZIP	core

E.5 Available converters

As discussed elsewhere in the book, the `ConverterService` plays a key role in the content negotiation and automatic conversion feature between beans and raw representations.

Table E.6 lists the modules (either the core module or the extensions) that provide automatic serialization from representation beans to raw representations, including the media types and classes or interfaces supported.

Note that the order of the converters in your classpath might matter and that each converter can express different scores for each media type or class, depending on their affinity with them.

Table E.6 Converters that provide automatic serialization

Module	Target media type	Serializable source class
core	Any kind, the one computed by the content negotiation	<code>java.lang.String</code>
core	<code>application/octet-stream</code>	<code>java.io.File</code>
core	<code>application/octet-stream</code>	<code>java.io.InputStream</code>
core	<code>application/x-java-serialized-object</code> , <code>application/x-java-serialized-object+xml</code> , <code>application/octet-stream</code>	Implements <code>java.io.Serializable</code>
core	Text based media types, or <code>text/plain</code>	<code>java.io.Reader</code> or subclasses
core	<code>application/x-www-form-urlencoded</code>	<code>org.restlet.data.Form</code>
atom	<code>application/atom+xml</code>	<code>org.restlet.ext.atom.Feed</code>
atom	<code>application/atomsvc+xml</code>	<code>org.restlet.ext.atom.Service</code>
emf	<code>application/xmi+xml</code>	Implements <code>org.eclipse.emf.core.EObject</code>

Table E.6 Converters that provide automatic serialization (continued)

Module	Target media type	Serializable source class
emf	application/x-ecore+xmi+xml	Implements org.eclipse.emf.core.EObject
emf	application/xml, text/xml, text_html	Implements org.eclipse.emf.core.EObject
freemarker	Any kind, the one computed by content negotiation	org.freemarker.Template
gwt	application/x-java-serialized-object+gwt	Implements java.lang.Serializable
html	application/x-www-form-urlencoded	org.restlet.ext.html.FormDataSet
html	multipart/form-data	org.restlet.ext.html.FormDataSet
jackson	application/json	Any class
javamail	application/xml	javax.mail.Message
jdbc	text/xml	javax.sql.rowset.WebRowSet
jdbc	text/xml	Java.sql.RowSet
jdbc	text/xml	org.restlet.ext.jdbc.JdbcResultSet
jaxb	According to content negotiation, one of the following: application/xml, text/xml, application/*+xml	Any class annotated with the javax.xml.bind.XmlRootElement
jibx	According to content negotiation, one of the following: application/xml, text/xml, application/*+xml	Any class, as soon as it's been bound
json	application/json	org.json.JSONArray
json	application/json	org.json.JSONObject
json	application/json	org.json.JSONTokener
rdf	application/*+xml	org.restlet.ext.rdf.Graph
rdf	text/x-turtle	org.restlet.ext.rdf.Graph
rdf	text/n-triples	org.restlet.ext.rdf.Graph
rdf	text/n3	org.restlet.ext.rdf.Graph
rome	application/atom+xml	com.sun.syndication.feed.synd.SyndFeed

Table E.6 Converters that provide automatic serialization (continued)

Module	Target media type	Serializable source class
rome	application/rss+xml	com.sun.syndication.feed.synd.SyndFeed
velocity	Any kind of media types	org.apache.velocity.Template
wadl	application/vnd.sun.wadl+xml	org.restlet.ext.wadl.ApplicationInfo
xml	application/*+xml, application/xml, text/xml	org.w3c.dom.Document
xstream	application/json (with jettison library)	Any kind of class
xstream	application/xml, text/xml, application/*+xml	Any kind of class

Table E.7 lists the modules (either the core module or the extensions) that provide automatic deserialization of raw representations into representation beans, including the media types and classes or interfaces supported.

Table E.7 Converters that provides automatic deserialization

Module	Target class	Media type of the request's entity
core	java.lang.String	Any kind
core	org.restlet.representation.StringRepresentation	Any kind
core	java.io.File	Only if the request's entity is an instance of org.restlet.representation.FileRepresentation
core	org.restlet.data.Form	application/x-www-form-urlencoded
core	java.io.InputStream	Any kind
core	org.restlet.representation.InputRepresentation	Any kind
core	java.io.Reader	Any kind
core	org.restlet.representation.ReaderRepresentation	Any kind

Table E.7 Converters that provides automatic deserialization (continued)

Module	Target class	Media type of the request's entity
core	Implements <code>java.io.Serializable</code>	<code>application/x-java-serialized-object</code> , <code>application/x-java-serialized-object+xml</code> , <code>application/octet-stream</code>
core	Java primitive type	<code>application/x-java-serialized-object</code> , <code>application/x-java-serialized-object+xml</code> , <code>application/octet-stream</code>
emf	Implements <code>org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl</code>	<code>application/xmim+xml</code>
emf	Implements <code>org.eclipse.emf.ecore.xmi.impl.EMOFResourceImpl</code>	<code>application/x-ecore+xmim+xml</code>
emf	Implements <code>org.eclipse.emf.ecore.xmi.impl.XMLResourceImpl</code>	<code>application/xml</code> , <code>text/xml</code> , <code>text_html</code>
gwt	Implements <code>java.lang.Serializable</code>	<code>application/x-java-serialized-object+gwt</code>
html	<code>org.restlet.ext.html.FormDataSet</code>	<code>application/x-www-form-urlencoded</code>
jackson	Any class	<code>application/json</code>
jaxb	Any class annotated with the <code>javax.xml.bind.XmlRootElement</code>	According to content negotiation, one of the following: <code>application/xml</code> , <code>text/xml</code> , <code>application/*+xml</code>
jibx	Any class, as soon it's been bound	According to content negotiation, one of the following: <code>application/xml</code> , <code>text/xml</code> , <code>application/*+xml</code>
json	<code>org.json.JSONArray</code>	<code>application/json</code>
json	<code>org.json.JSONObject</code>	<code>application/json</code>
json	<code>org.json.JSONTokener</code>	<code>application/json</code>
rdf	<code>org.restlet.ext.rdf.Graph</code>	<code>application/*+xml</code>
rdf	<code>org.restlet.ext.rdf.Graph</code>	<code>text/x-turtle</code>
rdf	<code>org.restlet.ext.rdf.Graph</code>	<code>text/n-triples</code>
rdf	<code>org.restlet.ext.rdf.Graph</code>	<code>text/n3</code>
rome	<code>com.sun.syndication.feed.synd.SyndFeed</code>	<code>application/atom+xml</code>

Table E.7 Converters that provides automatic deserialization (continued)

Module	Target class	Media type of the request's entity
rome	com.sun.syndication.feed.synd.SyndFeed	application/rss+xml
wadl	org.restlet.ext.wadl.ApplicationInfo	application/vnd.sun.wadl+xml
xml	org.w3c.dom.Document	application/*+xml, application/xml, text/xml
xstream	Any kind of class	application/json (with Jettison library)
xstream	Any kind of class	application/xml, text/xml, application/*+xml

E.6 Supported security challenge schemes

Table E.8 lists all the security schemes supported by the Restlet Framework and the corresponding extensions (see also chapter 5). It also indicates whether the client and/or server sides of the security scheme are implemented.

Table E.8 Challenge schemes and their implementation

Scheme	Module	Client, server class	Description
FTP_PLAIN	net	FtpClientHelper (client only)	Plain FTP authentication
HTTP_AWS_S3	crypto	AwsAuthenticator	Amazon S3 HTTP authentication
HTTP_AWS_QUERY	crypto	HttpAwsQueryHelper (client only)	Amazon Query String authentication
HTTP_AZURE_SHAREDKEY	crypto	HttpAzureSharedKeyHelper (client only)	Microsoft Azure Shared Key authorization (authentication)
HTTP_AZURE_SHAREDKEY_LITE	crypto	HttpAzureSharedKeyLiteHelper (client only)	Microsoft Azure Shared Key lite authorization (authentication)
HTTP_BASIC	core	HttpBasicHelper	Basic HTTP authentication
HTTP_COOKIE	crypto	CookieAuthenticator	Cookie HTTP authentication
HTTP_DIGEST	crypto	DigestAuthenticator	Digest HTTP authentication
HTTP_OAUTH	oauth	OAuthAuthorizer, OAuthHelper	Open protocol for API authentication
POP_BASIC	javamail	JavaMailClientHelper (client only)	Basic POP authentication (USER/PASS commands)

Table E.8 Challenge schemes and their implementation (continued)

Scheme	Module	Client, server class	Description
POP_DIGEST	javamail	JavaMailClientHelper (client only)	Digest POP authentication (APOP command)
SDC	net	GAE edition (client only)	Secure Data Connector authentication (Google)
SDC	sdc	JavaSE and JavaEE editions (client only)	Secure Data Connector authentication (Google)
SMTP_PLAIN	javamail	Core and JavaMail ClientHelper (client only)	Plain SMTP authentication

E.7 Scheme authorities of RIAP and CLAP pseudoprotocols

We've introduced the RIAP and CLAP pseudoprotocols in order to complete the vision of a unified way to access resources, even within a Restlet component, and not only from the outside using a network protocol.

Much as HTTP lets you access resources on the Web, RIAP lets you access resources defined in your own Restlet applications and components, whereas CLAP lets you access resources available via Java class loaders.

Such resources are designated as local resources. A special Reference subclass called `org.restlet.data.LocalReference` facilitates the creation of such references to local resources with several methods, such as `createClapReference()` and `createRiapReference()`.

Table E.9 lists all important constants related to those pseudoprotocols and supported by the `org.restlet.data.LocalReference` class. The table should help you understand which kind of resource can be targeted.

Table E.9 Types defined in the LocalReference class

Authority constant	Value	Description
CLAP_DEFAULT	" " or empty string	The resources will be resolved from the class loader associated with the local class. This is the same as the CLAP_CLASS authority.
CLAP_CLASS	class	The resources will be resolved from the class loader associated with the local class. This is the default CLAP authority.
CLAP_SYSTEM	system	The resources will be resolved from the system's class loader.
CLAP_THREAD	thread	The resources will be resolved from the current thread's class loader.

Table E.9 Types defined in the LocalReference class (continued)

Authority constant	Value	Description
RIAP_APPLICATION	application	The resources will be resolved from the current application's root Restlet.
RIAP_COMPONENT	component	The resources will be resolved from the current component's internal (private) router.
RIAP_HOST	host	The resources will be resolved from the current component's virtual host.

appendix F

Getting additional help

Beyond this book's broad presentation of the Restlet Framework, additional documentation is also available on the web. Also, you can directly interact with the Restlet community by asking questions and contributing in various ways, such as participating in discussion lists and checking out an issue tracker.

Because the Restlet Framework has been created and mainly maintained by Restlet SAS (previously named Noelios Technologies), we'll briefly point you to available professional services in case you need dedicated assistance or have specific needs.

F.1 Accessing online documentation

This section gives you pointers to the best online resources, including Javadocs, tutorials, and user guides to go beyond this book.

JAVADOCS

The Javadocs are available for each Restlet Framework version (such as 2.1) and edition (such as jse). For example, the Javadocs of version 2.1 of the Java SE edition are available at www.restlet.org/documentation/2.1/jse.

Note that each distribution of the Restlet Framework also contains a copy of the Javadocs as well as the whole source code for easier contextual documentation within your favorite IDE or offline reading.

WIKI

Apart from the main www.restlet.org site, a wiki is available that provides structured documentation about the framework. It's available at <http://wiki.restlet.org> and includes several sites. Each site has a table of contents on the left-hand side, a search box, and a set of pages visible on the right, sometimes in several versions (called *variants* by Daisy, the open source CMS used). Here are some interesting pages:

- *Description of the core Restlet API*—[/docs_2.1/27-restlet.html](http://docs_2.1/27-restlet.html)
- *Mapping between Restlet API and HTTP headers*—[/docs_2.1/130-restlet.html](http://docs_2.1/130-restlet.html)

- *Connectors*—[/docs_2.1/37-restlet.html](#)
- *Services*—[/docs_2.1/331-restlet.html](#)
- *Editions*—[/docs_2.1/275-restlet.html](#)
- *Extensions*—[/docs_2.1/28-restlet.html](#)
- *Tutorial*—[/docs_2.1/318-restlet.html](#)

This tutorial illustrates a simple application deployed on the Google App Engine platform and consumed by three kinds of Restlet-based clients: a GWT module, an Android application, and a classical JVM.

Another wiki site lists documentation provided by the Restlet community in other locations such as blog posts, YouTube videos, tutorials, articles, and books: <http://wiki.restlet.org/community/167-restlet.html>.

F.2 Asking questions

You have numerous ways to get in touch with other Restlet users or developers, but first make sure you read the available FAQs at http://wiki.restlet.org/docs_2.1/333-restlet.html.

Then you can search the popular StackOverflow help site for similar questions—or post your own and start helping others: <http://stackoverflow.com/search?q=restlet>.

You can get more directly in touch with Restlet users and developers via the discussion group (mailing list) at www.restlet.org/community/lists.

F.3 Code and issues repository

After using the Tigris.org forge for seven years, the project has recently moved to the more modern and productive GitHub platform for both the code repository (Git server) and the issues tracker. You can now fork the project at <https://github.com/restlet/restlet-framework-java> and contribute enhancements using the powerful pull request feature.

F.4 Professional services

If you need urgent help, can't publicly ask for questions, or want to sponsor the development of new features for the Restlet Framework, you're welcome to get in touch with Restlet SAS, the company behind this open source project. Here are some pointers:

- *Commercial licenses*—www.restlet.com/products/restlet-framework
- *Technical support*—www.restlet.com/products/support
- *Training*—www.restlet.com/services/training
- *Consulting*—www.restlet.com/services/consulting
- *Custom development*—www.restlet.com/services/development
- *Codevelopment*—www.restlet.com/services/development

references

- [1] Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional. 2006.
- [2] McLaughlin, Brett and Justin Edelson. *Java and XML*, 3rd Edition. O'Reilly. 2006.
- [3] "WebID protocol, W3C Incubator." www.w3.org/2005/Incubator/webid/spec/, <http://webid.info/spec>.
- [4] Hadley, Marc. "Web Application Description Language." www.w3.org/Submission/wadl/.
- [5] "W3C – HTML 4.01 Specification – Forms. www.w3.org/TR/html4/interact/forms.html."
- [6] REST – Cookies evaluation. http://roy.gbiv.com/pubs/dissertation/evaluation.htm#sec_6_3_4_2.
- [7] Koelle, David. "The Alphanum Algorithm." www.davekoelle.com/alphanum.html.
- [8] Berners-Lee, Tim. *Weaving the Web*. Harper Paperbacks. 2000. (See www.w3.org/People/Berners-Lee/Weaving/Overview.html)
- [9] Nottingham, Mark. "HTTP caching tutorial." www.mnot.net/cache_docs/.
- [10] "The Original HTTP as defined in 1991." www.w3.org/Protocols/HTTP/AsImplemented.html.
- [11] Wilson, Jesse (from Dalvik team). "Android's HTTP Clients." <http://android-developers.blogspot.fr/2011/09/androids-http-clients.html>.
- [12] Fielding, Roy T. "Representational State Transfer (REST)." http://roy.gbiv.com/pubs/dissertation/rest_arch_style.htm.
- [13] Bush, Vannevar. "As We May Think." www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/3881/.
- [14] Wolf, Gary. "The Curse of Xanadu." www.wired.com/wired/archive/3.06/xanadu.html.
- [15] Fielding, Roy T. "REST APIs must be hypertext driven." <http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>.

- [16] "Microformats community." <http://microformats.org>.
- [17] Berners-Lee, Tim, James Hendler, and Ora Lassila. "The Semantic Web." www.scientificamerican.com/article.cfm?id=the-semantic-web.
- [18] Brickley, Dan and Libby Miller. "Friend-Of-A-Friend specification." <http://xmlns.com/foaf/spec/>.
- [19] Hickson, Ian. "HTML 5—Microdata." <http://dev.w3.org/html5/md/>.
- [20] Cowan, Taylor. "Binding Java Objects to RDF." http://semanticweb.com/binding-java-objects-to-rdf_b10682.
- [21] "Hypertext Transfer Protocol—HTTP/1.0." <http://tools.ietf.org/html/rfc1945>.
- [22] Fielding, R., J. Gettys, J.C. Mogul, H.F. Nielsen, L. Masinter, P. Leach, and T. Berners-Lee. "Hypertext Transfer Protocol—HTTP/1.1." Internet RFC 2616, 1999. <http://tools.ietf.org/html/rfc2616>. (An effort to revise this specification is underway. Drafts of the revised specification can be found at <http://tools.ietf.org/wg/httpbis/>.)
- [23] "BiDirectional or Server-Initiated HTTP (hybi)." <https://datatracker.ietf.org/wg/hybi/charter/>.
- [24] Banon, Shay. "REST and Web Sockets?" www.kimchy.org/rest_and_web_sockets/.
- [25] "W3C. Server-Sent Events working draft." www.w3.org/TR/eventsourcem.
- [26] Wikipedia. "SPDY protocol." <http://en.wikipedia.org/wiki/SPDY>.
- [27] Chromium Project. "SPDY protocol." <http://dev.chromium.org/spdy>.
- [28] Wikipedia. "Waka (protocol)." [http://en.wikipedia.org/wiki/Waka_\(protocol\)](http://en.wikipedia.org/wiki/Waka_(protocol)).
- [29] Nottingham, Mark. "Will HTTP/2.0 Happen After All?" www.mnot.net/blog/2009/11/13/flip.
- [30] Waldo, Jim, Geoff Wyant, Ann Wollrath, and Sam Kendall. Sun Microsystems Laboratories. November, 1994. http://labs.oracle.com/techrep/1994/smlis_tr94-29.pdf.
- [31] Box, Don and Steve Maine. Microsoft, 2007. <http://channel9.msdn.com/Events/MIX/MIX07/DEV03>.
- [32] Larman, C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall. 2005.
- [33] <http://www.idef.com/IDEF1.htm>.
- [34] Internet-Draft. "URI Template." <http://bitworking.org/projects/URI-Templates/>.

For further reading

- "Comet programming." [http://en.wikipedia.org/wiki/Comet_\(programming\)](http://en.wikipedia.org/wiki/Comet_(programming)).
- "HTTP/1.1 bis initiative." <http://tools.ietf.org/wg/httpbis/>.
- Nottingham, Mark. "WADL to HTML transformation with XSLT." http://github.com/mnot/wadl_stylesheets.
- Prescod, P. "Reinventing Email using REST." <http://www.prescod.net/rest/restmail/>.
- Stone, Jonathan and Craig Partridge. "When The CRC and TCP Checksum Disagree, 2000." SIGCOMM '00: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.7611&rep=rep1&type=pdf>.

index

Symbols

- @Delete annotation 38
- @Get annotation 5–6, 38–39, 118
- @Options annotation 38–39
- @Post annotation 38–39
- @Put annotation 38, 95, 118
- \${variableName} 109
- \${variableName} 111

Numerics

- 200 status code 393
- 201 status code 393
- 202 status code 393
- 301 status code 393
- 400 status code 393
- 404 status code 393
- 405 status code 393
- 500 status code 393

A

- abstract data model 223
- Accept-Charset header 114
- Accept-Encoding 186
 - header 114, 221
- Accept-Language header 114
- Access-Control-Allow-Origin header 259
- Account class 287, 381
- account feeds 180
- accountId attribute 28–29, 43
- accountRef property 98, 110, 118

- AccountRepresentation 288
- AccountResource 288, 291
- Accounts resource 388
 - uses POST method 392
- AccountServerResource
 - class 35, 43
- Activator class 323
- Add External JARs button 339
- Add JAR/Folder button 344
- addProduct method 229–230
- afterHandle 25
- agentId parameter 237
- alternatives to HTTP
 - protocol 296–298
 - SPDY protocol 297–298
 - SSE 297
 - WebSocket protocol 296–297
- Amazon Elastic Beanstalk 211–214
 - deploying applications in 212–214
 - overview 211–212
- Amazon S3 (Simple Storage Service) 230–233
 - accessing resources 230–233
 - configuring buckets 230
- Amazon Simple Notification Service 212
- Android 262–272
 - editions for 324
 - overview 263
 - requirements for 263–267
 - Restlet edition for
 - client-side support 268–271
 - extensions for 267–268
 - server-side support 271–272
- Android Hierarchy view 266
- Android TraceView 266
- Android Virtual Device. *See* AVD
- android.permission.INTERNET
 - Uses Permission 268
- AndroidManifest.xml file 268
- annotated property 33
- AOP (aspect-oriented programming) 63, 67
- APISpark platform 306–307
- APOP command 411
- APP (Atom Publishing Protocol) 223
- appcfg.sh script 209
- AppendableRepresentation 86, 198, 277, 318
- appengine-web.xml file 209
- Application class 19, 317
- APPLICATION_JAVA_OBJECT_GWT 256
- APPLICATION_JAVA_OBJECT_XML 256
- Application.getCurrent() method 23
- ApplicationInfo class 155, 157
- applications 13–45
 - common services for 22–24
 - context of 20–22
 - properties for 19–20
 - purpose of 14–15
 - routing system for 24–30
 - filtering for processing 24–27
 - URI-based routing 27–30
 - structure of 15–17
 - subclass for 17–18

- applications (*continued*)

 using resources in 30–45

 ClientResource class 35–38

 in MailServerApplication

 42–45

 Resource class 30–31

 ServerResource class 31–35

 using Java annotations 38–41
- apps-secure-data-

 connector.google.com 237
- architecture of Restlet edition

 for GWT 249
- ArrayList class 256
- aspect-oriented programming.

 See AOP
- AsyncCallback interface 256
- AsyncTask 270
- Atom Publishing Protocol. *See*

 APP
- Atom Syndication Format 223
- AtomPub 222–223
- attachments property 65
- attribute extraction 184
- authenticating users 129–141

 Authenticator class 134–135

 certificate-based

 authentication 139–141

 challenge-based

 authentication 135–136

 credentials in client 130–134

 receiving in response 132–133

 setting in request 130–132

 support for proxy

 authentication 133–134

 verifying credentials 136–139

 DigestAuthenticator

 class 138

 SecretVerifier class 136–137

 using JAAS 138–139
- Authenticator

 class 134–135, 140, 210, 319

 filter 27
- AuthenticatorHelper 321
- Authorization header 231, 234
- Authorizer

 class 143–144, 320

 filter 27
- authorizing user actions 143–147
- Authorizer class 143–144
- for particular resources 145–146
- MethodAuthorizer class 145
- RoleAuthorizer class 144–145
- using Java security

 manager 146–147
- autoCommitting property 33
- autoDescribing property 157, 159
- autodiscovery mechanism 324
- automatic compression 9
- Automatic object 255
- automatic object serialization

 support, in Restlet edition

 for GWT 255–257
- available property 84
- availableSize property 84
- AVD (Android Virtual Device) 263
- AWS Management Console 212, 230–232
- AwsAuthenticator 410
- Azure table 234
-
- B**
- beforeHandle 25
- benefits

 of cloud platforms

 client access to

 services 206–207

 SaaS portability 204–205

 of Restlet framework 9–10
- Berners-Lee, Tim 280
- blob service 233
- Blocker class 26–29
- Boolean property 95
- Box, Don 369
- buckets, for Amazon S3 230
- ByteArrayRepresentation 318
-
- C**
- C2DM (Cloud to Device Messaging) framework 272
- CA (Certification Authority) 123
- CacheService 299
- caching 9, 188–189
- CallbackHandler 138
- Certificate Revocation Lists. *See*

 CRL
- CertificateAuthenticator 134, 140
- certificate-based

 authentication 139–141
- certificates

 generating

 requests for 125–126

 self-signed 125

 importing trusted

 certificates 126

 storing 124–125
- Certification Authority. *See* CA
- challenge schemes 410
- ChallengeAuthenticator 135, 169
- and the JaasVerifier 138
- and verifiers 143
- extending 170
- challenge-based

 authentication 135–136
- ChallengeRequest 130, 132, 318
- ChallengeResponse 100, 130, 232, 318
- ChallengeScheme 130, 135, 318
- ChannelRepresentation 318
- CharacterSet 318
- characterSet property 82
- checkDigest() method 148–150
- cia variable 183
- cig variable 183
- CLAP (ClassLoader Access Protocol) 173

 client 55

 pseudoprotocol 411
- CLAP_CLASS 411
- CLAP_DEFAULT 411
- CLAP_SYSTEM 411
- CLAP_THREAD 411
- Class Libraries type 343
- Class parameter 251
- classes

 for HTTP 398

 for OData protocol 226–228

 for REST architecture 397
- ClassLoader Access Protocol. *See*

 CLAP
- Client class 317
- client connectors 53–57, 365
- clientDispatcher 183, 193
- clientDispatcher property 19, 21, 36
- ClientHelper 321
- ClientInfo 318
- clientInfo property 140–142, 210
- clientInfo.certificates

 property 140
- ClientProxy

 extension 255
- interface 256–257

- ClientProxy interface 52
ClientResource 268, 270, 350, 353, 355–357
and helper methods 116–117
source for calls 35–38
using 7–8
with GWT client side 252
ClientResource.get(Class<T>) method 180
ClientResource.put(Object) method 168
ClientResource.setRequest-EntityBuffering(true) method 208
client-side support, of Restlet edition for Android 268–271
clientTrust.jks 140
cloud platforms 203–241
Amazon Elastic Beanstalk 211–214
deploying applications in 212–214
overview 211–212
Amazon S3 230–233
accessing resources 230–233
benefits of 204–207
client access to services 206–207
SaaS portability 204–205
Google App Engine 207–211
deploying applications in 208–210
Google Accounts authentication for 210–211
overview 207–208
URL fetch service for 221
OData protocol 221–230
calling services for 228–230
generating classes for 226–228
overview 222–226
SDC extension for 235–240
implementing 238–240
in Google App Engine 240
installing agent 236–238
overview 235–236
Windows Azure 214–220, 233–235
configuring storage accounts 233
deploying applications in 215–220
- overview 214–215
using table service 233–235
Cloud to Device Messaging. *See* C2DM
cloudapp.net domain 220
cloud-scale databases 192
CloudWatch monitoring 212
CN (Common Name) 123
code annotations 302–303
com.google.gwt.json.client package 253
com.sun.security.auth.UserPrincipal 139
com.sun.syndication.feed.synd.SyndFeed 407
command line, compiling with 349
command pattern 247
Command-line tools 209
commit() method 33
committed property 33
Common Name. *See* CN
common services for applications 22–24
Comparator object 173
Component class 50–53, 317
declarative XML configuration 62
Component.defaultHost property 59
Component.hosts property 59
Component.internalRouter property 196
Component.main() method 63
Component.services property 61
Component() method 90
components 9, 47–48
in REST architecture style 364
models 368
structure of 48–49
XML configuration for 62–63
compressing representations 186–187
computeDigest() method 148–150
conditional methods 190–191
processing 9
property 33
conf/settings.xml file 329
ConfidentialAuthorizer 144, 320
configuration file 219 per module 244
Configure Variables button 339
Connector class 53, 317
connectors 9, 299, 405
in REST architecture style 364–365
Servlet engine as 68–70
ConnectorService 22
connectTimeout 54
ConnegService 22, 300–301
Console view 342
construction phase 374, 376
consuming feeds 180
linked data 291–293
Contact class 287, 381
ContactRepresentation 252, 254
Contacts class 252
Contacts resource 395
ContactsResourceProxy 256
ContactsServiceImpl 260–261
containers, Servlet engine as 70–71
content negotiation 9
Content-Encoding 186
ContentHandler interface 290
ContentHandler() method 92
Content-Length header 221 problem 185
Content-MD5 header 148–150
Content-Type header 225
Context class 21, 143, 196, 317
context, of applications 20–22
ConverterHelper 321
converters 406–408
ConverterService 22, 39, 259, 290, 299–300
CookieAuthenticator 172, 410
cookie-based authentication 169
cookies 169–172
CORS (Cross-Origin Resource Sharing) 259
Create AVD button 264
create method 251
Create Project from Scratch option 345
Create storage account button 233
create() method 41
create<ENTITY>Query method 228
createChildContext() method 22

createClapReference()
 method 411
createElementNS(String, String) method 96
createInboundRoot method 18, 26, 51, 66, 170, 210, 261
createObjectMapper()
 method 107
createProductQuery
 method 229
createRiapReference()
 method 411
createTransformer() method 89
createmapackage 218
credentials 130–134
 receiving in response 132–133
 setting in request 130–132
 support for proxy
 authentication 133–134
 verifying 136–139
DigestAuthenticator
 class 138
SecretVerifier class 136–137
 using JAAS 138–139
Credentials cookie 171
CRL (Certificate Revocation Lists) 129
cross-domain requests
 in Restlet edition for GWT 257–259
 in server-side extension for GWT 261–262
Cross-Origin Resource Sharing.
See CORS
cscfg file 219
csdef file 219
.cspack.jar 216
CSR (certificate signing request) 125
cURL 355
currentLogger property 22
CUSTOM scheme 131

D

Dalvik Debug Monitor Server.
See DDMS
Dart edition 301
data
 layer 384
 model for RDF 282–283
 objects 244
 package 317–318

database sharding,
 replication 192
Date class 83
DDMS (Dalvik Debug Monitor Server) tool 266
Debian Linux project 326
 debug parameter 237
 debugging 357–358
DecoderService 22, 186
deeplyAccessible property 173
DefaultHandler class 94
defaultMatchQuery property 29
DefaultSslContextFactory 128
defaultVerifier property 22
deferred binding 249, 259
 activated during compilation phase 255
 supported data format 257
DELETE method 38, 224, 232, 235, 389–390
delete() method 8, 34, 37
deleteProduct method 229–230
deploying applications 46–77
 and Restlet components 47–48
 structure of 48–49
 in Amazon Elastic Beanstalk 212–214
 in Google App Engine 208–210
 in Java EE server 67–76
 in OSGi environments 73–76
 Oracle XML DB extension for 71–72
 Restlet framework as library 72–73
 Servlet engine as connector for components 68–70
 Servlet engine as container of applications 70–71
 Servlet extension for 67–68
 in Windows Azure 215–220
 with Java SE 50–62
 Component class 50–53
 configuring services for 61–62
 server and client
 connectors 53–57
 virtual hosting for 57–61
XML configuration for 62–67
 with components 62–63
 with Spring framework 63–67
describeGet(MethodInfo) 157
Descriptor file 217
design implementation 40, 373
developed extensions 325
digest property 84
DigestAuthenticator 136, 138, 410
DigesterRepresentation 148–150, 318
digesting
 representation digesting 148–149
 without losing content 149–150
DigestVerifier 136, 138
Directory class 54, 172–173, 277, 319
discussion groups 414
Disk-based file 177
Display table 234
disposition property 84
distributed objects 366
 systems 368
distributions 328–334
 Eclipse update site 331–334
 Maven repository 328–330
 Windows installer 331
 zip files 330–331
doCatch(Throwable)
 method 175
documenting 151–164
 overview 152
 pitfalls for 152–153
 recommendations for 153–154
WADL
 converting documents to HTML 163–164
 overview 154–155
WadlApplication class 155–156
WadlServerResource
 class 156–163
 describing single resource with 162–163
 improving description of existing server resources with 158–162
 methods for 156–158
doInit() method 31, 34, 51, 115
DOM API, XML representations
 using 89–91
domain parameter 237
DomRepresentation 94–95, 99
 properties and methods 89
doRelease() method 31, 34

download.vbs 217
DynamicContentServer 187

E

Eclipse 336–341
support plug-in 209
update site for 331–334
Eclipse Modeling Framework.
See EMF
editions 322–327
extensions for 324–326
for Android 324
for Google App Engine 323
for Google Web Toolkit 323
for Java EE 323
for Java SE 322
for OSGi Environments 323
logical versions 326–327
versioning scheme for 327
elaboration phase 373–376
EMF (Eclipse Modeling
Framework) 104
EmfRepresentation class 104
EncoderService 22, 186
encodings property 82
enddef marker 302–303
Engine class 321, 358
Enroler interface 141–142, 210,
 319
 default for 143
ephemeralPort 54
error pages 174–176
errorHandler 98
ESB (Enterprise Service
Buses) 308
E-Tag (entity tag) 189
exhaust() method 149
existing property 33
expand property 228
expirationDate property 84
Expires header 188
exposing feeds 178–179
extending variables 341
extensions 321–322
 for editions 324–326
 for Restlet edition for
 Android 267–268
external actors 379
Extractor filter 27

F

Feed class 381
feeds 178–180

Feeds resource 395
feedTableList.getEntries()
 method 234
Fielding, Roy T. 361
file attachment field 176–177
FILE client 55
file uploads 176–177
FileRepresentation 277, 318
Filter class 24, 27, 319
filter property 228
filtering for processing 24–27
Finder class 31
finderClass property 20
FoafBrowser, launch 292
followRedirects property 36,
 182
Form class 167–168, 176, 277,
 320
formattedOutput property 103
FormDataSet class 277
forms 166–168
framework 315–327
 editions 322–327
 extensions for 324–326
 for Android 324
 for Google App Engine 323
 for Google Web Toolkit 323
 for Java EE 323
 for Java SE 322
 for OSGi environments
 323
 logical versions 326–327
 versioning scheme for 327
extensions 321–322
Restlet API 315–320
 data package 317–318
 representation package 318
 resource package 318–319
 root package 316–317
 routing package 319
 security package 319–320
 service package 320
 util package 320
Restlet engine 321
FreeMarker extension, template
 representations using
 109–111
FreeMarker template 166, 175,
 193
FTP_PLAIN scheme 131
future of Restlet 299–307
 APISpark platform 306–307
 connectors 299
 contributing to 310
 Dart edition 301

enhancements 299–301
CacheService 299
ConnegService 300–301
ConverterService 299–300
JAVA 6 support 299
size optimization 301
unified bean converter 300
JavaScript edition 301
Restlet Apps 304–305
Restlet Cloud 305
Restlet forge 303–304
Restlet Studio 305
third-party projects 308–310
 integration efforts 308–309
 stacks 309–310
using code annotations 302–
 303

G

GAE (Google App Engine) 11,
 14, 205, 207–211
deploying applications
 in 208–210
editions for 323
Google Accounts authentication
 for 210–211
overview 207–208
SDC extension in 240
URL fetch service for 221
GaeAuthenticator 210
GaeEnroler class 210
GData (Google Data) 222
Generator class 226
GET method 163–164, 279, 387,
 390
 and @Get 38
 calling remote web
 resource 247
 for information retrieval 391
 retrieve state of resource 190
 updating description 157
get() method 6, 8, 34, 37, 39, 115
getAttributes() method 21
getChallengeRequests()
 method 133
getChannel() method 84
getChild method 251
getClientResource method 52,
 119, 257
getConverterService()
 method 32
getCookies() method 31
getCookieSettings() method 31
getCount property 229

getDefaultHost() method 60
 getDocument method 89, 254
 getElementsByTagName
 method 254
 getEncoderService()
 method 186
 getChild method 254
 getFirstValue(String name)
 method 168
 getFoafProfile() method 288, 290
 getJsonArray() method 253
 getMetadataService()
 method 32
 getName() method 141
 getParent method 251
 getQuery() method 31
 getReader() method 84, 149
 getReference() method 31
 getRegisteredClients
 method 269
 getRegisteredConverters
 method 269
 getRepresentation(Status,
 Request, Response)
 method 174
 getRequest() method 31, 35
 getRequestAttributes()
 method 33, 35
 getResourceInfo() method 156
 getResponse() method 31, 33
 getRoles() method 141
 getStream() method 84, 89, 149
 getText() method 84, 149
 getUser() method 141
 getValuesArray(String name)
 method 168
 getWebRepresentation()
 method 168
 Git server 414
 GitHub platform 414
 global library 344
 Google App Engine. *See* GAE
 Google SDC agent 236
 Google search 389–390
 Google Web Toolkit. *See* GWT
 Gourley, David 391
 Graph class 286
 GraphBuilder class 291
 GraphHandler 290–291, 293
 groups, for user roles 142–143
 GWT (Google Web Toolkit) 11,
 242–246
 and REST 246
 Android 262–272
 overview 263

requirements for 263–267
 Restlet edition for 267–272
 editions for 323
 installing 244–245
 overview 243–244
 Restlet edition for 247–259
 architecture flexibility 249
 automatic object serialization support 255–257
 client-side API 250–252
 concepts of 248–249
 handling cross-domain requests 257–259
 JSON representations in 252–254
 RequestBuilder class 247–248
 XML representations in 252–254
 server-side extension for 259–262
 handling cross-domain requests 261–262
 with GWT-RPC 260–261
 GWT object 256–257, 259, 309
 GWT.create static method 255
 GWT-specific object 256

H

handle method 31, 38, 251, 350
 handle(Request, Response)
 method 316, 358
 HATEOAS (hypermedia as the engine of application state) 274
 principle 275
 HEAD method 391
 head() method 34, 37
 headers, Restlet properties for 399
 health check 212, 214
 helloClientResource.get()
 method 8
 helloServer.start() method 6
 HelloServerResource 6, 182, 187
 HelloWorld class 340, 345, 347
 HelloWorld.java class 349
 HelloWorld.main() method 347
 help
 discussion groups 414
 online documentation 413–414
 Javadocs 413
 wiki 413–414

professional services 414
 repositories 414
 Helper class 321
 homeRef property 174
 Host header 221
 hostDomain property 58
 hostPort property 58
 hostScheme property 58
 HTML redirections 181
 HTML table 166
 HTTP Client 56
 HTTP content negotiation 113–119
 combining annotated interfaces and converter service 117–119
 configuring client preferences 116–117
 declaring resource variants for 115–116
 overview 113–114
 HTTP protocol 8–9, 118, 214, 223–224, 255–256, 388
 alternatives to 296–298
 SPDY protocol 297–298
 SSE 297
 WebSocket protocol 296–297
 and REST architecture style 365–366
 headers, Restlet properties for 399
 history of 295
 HTTP/1.1 bis initiative 296
 main ones 390
 Restlet classes for 398
 using correctly 391–392
 HTTP WWW-Authenticate header 132
 HTTP_AWS_QUERY scheme 131
 HTTP_AWS_S3 scheme 131, 231–232
 HTTP_AZURE_SHAREDKEY scheme 131, 234
 HTTP_AZURE_SHAREDKEY_LITE scheme 131
 HTTP_BASIC scheme 131
 HTTP_COOKIE scheme 131
 HTTP_DIGEST scheme 131
 HTTP_NTLM scheme 131
 HTTP_OAUTH scheme 131
 HTTP/1.1 bis initiative 296
 HTTP/HTTPS client 55
 HttpAwsQueryHelper 410

H
 HttpAzureSharedKeyHelper 410
 HttpAzureSharedKeyLite 410
 HttpBasicHelper 410
 HTTPS, enabling 126–128
 HttpServlet class 316
 HttpServletResponse
 method 72
 HttpURLConnection class 56,
 221, 240, 268, 315
 hyperdata 278–280, 383
 hypermedia 275–280, 395
 defined 276
 HATEOAS principle 275
 hyperdata 278–280, 383
 hypertext
 defined 276
 support for 276–277
 hypermedia as the engine of
 application state. *See*
 HATEOAS
 hypertext
 defined 276
 support for 276–277

I
 IaaS (Infrastructure as a
 Service) 204
 IDE (integrated development
 environment) 334–349
 command line compiling 349
 Eclipse 336–341
 IntelliJ IDEA 345–348
 NetBeans 343–345
 idempotent
 methods 391
 requests 8
 identifier property 83
 IETF (Internet Engineering
 Task Force) 122
 ifdef marker 303
 If-Match header 190
 If-Modified-Since header 190
 ifndef marker 303
 If-None-Match header 190
 If-Unmodified-Since header
 190
 implementation testing 373
 inbound
 root 16, 18, 26
 server redirection 181
 inboundRoot property 19, 66
 inception phase 373–377
 indexName property 173

J
 Infrastructure as a Service. *See*
 IaaS
 init() method 128
 inlineCount property 229
 InputEndpoint 217
 installing 328–358
 distributions 328–334
 Eclipse update site 331–334
 Maven repository 328–330
 Windows installer 331
 zip files 330–331
 GWT 244–245
 IDE 334–349
 command line
 compiling 349
 Eclipse 336–341
 IntelliJ IDEA 345–348
 NetBeans 343–345
 instruction keyword 303
 integrated development envi-
 ronment. *See* IDE
 integration testing 355–356
 ClientResource 355–356
 cURL 355
 RESTClient 355
 IntelliJ IDEA 345–348
 Internet Engineering Task
 Force. *See* IETF
 Inversion of Control. *See* IoC
 IoC (Inversion of Control) 20
 isInRole() method 320
 Issuer DN (Issuer Distinguished
 Name) 123

J
 JAAS, verifying credentials
 using 138–139
 JaasUtils.doAsPrivileged()
 method 146
 JaasVerifier 136, 138–139
 Jackson extension, JSON repre-
 sentations using 107–108
 JacksonRepresentation class 87,
 107
 JAR file 221, 244, 251, 330–331
 Android repackages 268, 324
 JAVA 6 support 299
 Java annotations 38–41
 Java class 208, 226, 244, 248,
 349, 411
 and compiled XML
 Schema 103
 and JAXB extension 102
 JAXB-annotated 103

Java Community Process. *See* JCP
 Java Database Connectivity. *See*
 JDBC
 Java EE server
 deploying applications in 67–
 76
 in OSGi environments 73–
 76
 Oracle XML DB extension
 for 71–72
 Restlet framework as
 library 72–73
 Servlet engine as connector
 for components 68–70
 Servlet engine as container
 of applications 70–71
 Servlet extension for 67–68
 editions for 323

Java Enterprise Edition. *See*
 Java EE

Java interface 40–42
 Java method 5–6, 8, 41, 167
 Java object 246, 256, 259
 Java Runtime Environment. *See*
 JRE

Java SE (Java Standard
 Edition) 11
 deploying applications
 with 50–62
 Component class 50–53
 configuring services for 61–
 62
 server and client
 connectors 53–57
 virtual hosting for 57–61
 editions for 322

Java Secure Socket Extension.
 See JSSE

Java security manager, authoriz-
 ing user actions using 146–
 147

Java Server Pages. *See* JSP

Java Standard Edition. *See*
 Java SE

java.beans.XMLEncoder
 class 256

java.io.File 92, 406, 408

java.io.InputStream 84, 406

java.io.ObjectInputStream
 class 256

java.io.OutputStream 85

java.io.Reader 84, 408

java.io.Serializable 406, 409

java.io.Writer 85

java.lang.Serializable 407

java.lang.String class 84, 406, 408
 java.net.HttpURLConnection 322, 325
 java.nio package 268
 java.nio.ReadableByteChannel 84
 java.nio.WritableByteChannel 85
 java.security.Principal interface 141
 Java.sql.RowSet 407
 java.util.logging API 21
 java.util.logging.config.file 357
 Java-based command 209
 Javadocs 55, 59, 66, 413
 JavaMail 56
 JavaMailClientHelper 410–411
 JavaScript edition 301
 javax.mail.Message 407
 javax.net.ssl.SSLContext 140
 javax.servlet 176
 javax.sql.rowset.WebRowSet 407
 javax.xml.bind.XmlRootElement 407
 javax.xml.parser package 88–89, 92
 javax.xml.validation package 97
 JAXB extension, XML representations using 102–104
 JaxbRepresentation class 87, 103
 JCP (Java Community Process) 38
 JDBC (Java Database Connectivity), extension 197–198
 JRE (Java Runtime Environment) 216
 JSON
 representations 105–108
 in Restlet edition for GWT 252–254
 using Jackson extension 107–108
 using JSON.org extension 106–107
 type 253
 JSON.org extension, JSON representations using 106–107
 JSONArray 252–253
 JSONBoolean 253
 JSONNull 253
 JSONNumber 253
 JSONObject class 106, 253
 JSONParser class 252

JsonpRequestBuilder class 258
 JsonRepresentation class 87, 106–107, 252
 JSONString 253
 JSONValue class 252
 JSP (Java Server Pages) 108
 JSSE (Java Secure Socket Extension) 123
 jSSLUtils 128–129
 JsslutilsSslContextFactory 128
 JUnit 352–354
 JVM parameter 61

K

keyPassword parameter 126
 keys, storing 124–125
 keystore 124
 configuring on a server 127
 importing CA certificates 126
 prompted for password and key 125
 KeyStore class 124
 keystorePassword parameter 126
 keystorePath parameter 126
 keystoreType parameter 126
 keytool 125, 140

L

languages property 83
 Larman, Craig 374, 378
 launcher.XXMaxPermSize property 264
 LdapLoginModule 138
 lib directory 227
 library, Restlet framework as 72–73
 Link class 287
 LINQ (Language-Integrated Query) 228
 lisa.getContacts() method 288
 List interface 320
 ListActivity 269
 listingAllowed property 172–173
 Literal class 287
 Local file 173
 localConfig.xml file 237
 localConfigFile parameter 237
 localhost 271
 LocalReference class 411
 LocalVerifier 138, 320
 location transparency 367

log4jPropertiesFile parameter 237
 logical architecture, for RESTful web APIs 382–384
 logical versions 326–327
 LoginModule 138–139
 logLevel 358
 LogService class 61, 358
 loose coupling 359, 370
 Lucene extension 198

M

m (milestone) 327
 m variable 183
 Mail class 107, 115, 117, 381, 386
 Mail.gwt.xml 251
 Mail.xsd file 103
 MailApiApplication 193, 196
 MailApplication 140
 MailClient class 94, 99, 119
 program 37
 MailComponent 194
 mail-editing form 168, 176–177, 194
 MailResource interface 117–119
 Mails resource 395
 MailServerApplication class 91, 155–156, 170–171, 352
 attach resource class to router 31
 declaring supported roles 144
 JAAS application name 139
 resources in 42–45
 route HTTP calls to 70
 updating 42
 MailServerComponent class 51, 55, 61, 142, 156, 196, 353
 trimmed for Servlet deployment 69
 MailServerResource class 89–90, 94–95, 101, 168, 197
 exchanging JSON representations 106
 MailSiteApplication 193, 196
 MailStatus.ftl template 175
 MailStatusService() method 175
 main() method 6
 major.minor.release 327
 MANIFEST.MF file 73
 MapVerifier 135, 137, 320
 Maven repository 328–330
 maxRedirects property 36
 MediaType class 114, 318

mediaType property 83
MediaType.APPLICATION_JAVA_OBJECT 256
MemoryRealm 142
Message class 100, 317
Message.cacheDirectives 188
MetadataService 22, 39, 117
META-INF/services folder 268
metamodels 281
MethodAuthorizer class 144–145, 320
methodInfo parameter 157
methods
for RESTful web APIs 390–392
consider each resource
class 392
using HTTP methods
properly 391–392
for WadlServerResource
class 156–158
invocation 366
query parameter 167
microformats 278
MIME type 39
MobileMailClientMainActivity 269
mode property 183
MODE_CLIENT_FOUND 183
MODE_CLIENT_SEE_OTHER 183
MODE_CLIENT_TEMPORARY 183
MODE_SERVER_INBOUND 183
MODE_SERVER_OUTBOUND 183
Model View Controller. *See* MVC
modifiable property 169, 173
modificationDate property 83
modularizing applications 192–197
and private applications 196–197
RIAP pseudoprotocol 194–196
server dispatcher for 192–194
MODULE-NAME module 244
<MODULE-NAME>.gwt.xml file 244
MOVE method 392
multithreaded subclasses 24
MVC (Model View Controller) 63
myApplication.getServices()
method 23

MyService class 23
MyService() method 23
MySQL database 197

N

NaiveCookieAuthenticator 169–170
Name property 229
NameCallback 138
NamespaceContext interface 95
namespaces
for XML representations 95–97
property 95
needClientAuthentication
parameter 126, 139
negotiated property 33
negotiatingContent
property 173
Nelson, Ted 276
NetBeans 343–345
next property 36, 193
n-n cardinality 386
NodeList class 94
Noelios Technologies 413
nonrepeating resources 388
NotFoundException class 175
Nottingham, Mark 296

O

OAuth 2.0 131
OAuthAuthorizer 410
OAuthHelper 410
ObjectMapper class 300
Object-Oriented Analysis and Design. *See* OOA/D
object-oriented paradigm 363
ObjectOutputStream class 256
Object-Relational Mapping. *See* ORM
ObjectRepresentation 318
objects layer 384
OData 226–227
calling services for 228–230
generating classes for 226–228
overview 222–226
protocol 221–230
onCreate method 269, 271
online documentation 413–414
Javadocs 413
wiki 413–414
onSuccess method 257

ontologies 281, 283, 286, 291
Ontology Web Language. *See* OWL
OOA/D (Object-Oriented Analysis and Design) 372
Open Specification Promise. *See* OSP
OpenID 2.0 131
OPTIONS method 23, 38–39, 153, 155, 157, 391
options() method 34, 37, 39
Oracle database 72
Oracle XML DB extension, for Java EE server 71–72
orderby property 224, 228
org.apache.commons.io 176
org.apache.velocity.Template 408
org.eclipse.emf.core.EObject 406–407
org.freemarker.Template 407
org.json.JSONArray 407, 409
org.json.JSONObject 407, 409
org.json.JSONTokener 407, 409
org.restlet package 6, 9, 250, 316
org.restlet.<edition> 329
org.restlet.app.<name> modules 305
org.restlet.app.search application 305
org.restlet.Application 17, 19, 155, 164
org.restlet.client 250, 397
org.restlet.client.resource package 256
org.restlet.client.resource.ClientProxy interface 255
org.restlet.Component 50, 397
org.restlet.Connector 397
org.restlet.Context class 21
org.restlet.data package class 83–84, 116, 282
org.restlet.data.Cookie class 169
org.restlet.data.CookieSetting class 169
org.restlet.data.Form class 168, 176, 277, 406, 408
org.restlet.data.LocalReference class 411
org.restlet.data.Metadata 398
org.restlet.data.Parameter 168
org.restlet.data.Status class 37
org.restlet.data.Tag class 83
org.restlet.engine.Engine class 269, 358

org.restlet.engine.security
 .SslContextFactory 140
 org.restlet.ext.<code>
 package 322
 org.restlet.ext.<extension> 324,
 329
 org.restlet.ext.atom
 extension 178, 227, 267
 org.restlet.ext.atom.Feed 406
 org.restlet.ext.atom.Service 406
 org.restlet.ext.crypto
 extension 136, 231, 267
 org.restlet.ext.crypto.CookieAuthenticator class 172
 org.restlet.ext.fileupload
 extension 176
 org.restlet.ext.fileupload
 .RestletFileUpload class 177
 org.restlet.ext.freemarker 110
 extension 227
 org.restlet.ext.gae
 extension 210
 org.restlet.ext.gwt
 extension 256, 259
 org.restlet.ext.html 267
 org.restlet.ext.html
 .FormDataSet 407
 org.restlet.ext.httpclient 267
 org.restlet.ext.jaas 267
 extension 138
 org.restlet.ext.jackson 105, 107,
 119, 267
 org.restlet.ext.jaxrs 42
 org.restlet.ext.jdbc
 extension 197
 org.restlet.ext.jdbc
 .JdbcResultSet 407
 org.restlet.ext.jetty.jar 127
 org.restlet.ext.jibx package 105
 org.restlet.ext.json package 106,
 267
 org.restlet.ext.lucene
 extension 198
 org.restlet.ext.net 267
 org.restlet.ext.net.jar 127
 org.restlet.ext.odata 268
 org.restlet.ext.odata
 extension 226–227
 org.restlet.ext.rdf
 extension 268, 280
 org.restlet.ext.rdf.Graph 407,
 409
 org.restlet.ext.rdf.jar 282, 290
 org.restlet.ext.rome
 extension 178

org.restlet.ext.sdc extension 239
 org.restlet.ext.servlet
 extension 211
 org.restlet.ext.sip 268
 org.restlet.ext.slf4j extension 62
 org.restlet.ext.spring module 64
 org.restlet.ext.ssl package 128
 org.restlet.ext.velocity 112
 org.restlet.ext.wadl
 .ApplicationInfo 408
 org.restlet.ext.wadl.jar file 155
 org.restlet.ext.xml 88–89, 92,
 94, 99, 102, 268
 org.restlet.ext.xstream 87, 105,
 119
 org.restlet.jar 10, 54, 301, 335–
 336, 339, 343, 347, 349
 org.restlet.JSON 251
 org.restlet.lib.<jar identifier>
 329
 org.restlet.Message 398
 org.restlet.representation
 package 81–82
 org.restlet.representation
 .Representation 398
 org.restlet.representation.String
 Representation 34
 org.restlet.representation
 .Variant 398
 org.restlet.Request 398
 org.restlet.resource
 package 118
 org.restlet.resource
 .ClientResource class 21
 org.restlet.resource
 .Directory class 172
 org.restlet.resource.Resource
 398
 org.restlet.resource
 .ServerResource 31
 org.restlet.Response 398
 org.restlet.Restlet 19, 24, 30,
 196, 251, 255
 org.restlet.routing.Redirector
 class 261
 org.restlet.routing.Router
 class 27
 org.restlet.security package 130,
 141
 org.restlet.security.Authorizer 25
 org.restlet.security.Enroler
 interface 141
 org.restlet.security.Group 142
 org.restlet.security.Realm
 class 142

org.restlet.security.Role 147
 org.restlet.Server 398
 org.restlet.service package 24
 org.restlet.service.Service
 class 23
 org.restlet.service.StatusService
 174
 org.restlet.Uniform
 interface 19, 398
 org.restlet.util.Resolver
 interface 110
 org.restlet.util.Resolver<T>
 class 183
 org.restlet.XML 251
 org.xml.sax package 92
 origin server 365
 ORM (Object-Relational
 Mapping) 197, 384
 OSGi environments 73–76, 205,
 323
 OSP (Open Specification
 Promise) 222
 outbound
 root 16–17
 server redirection 181
 outboundRoot property 19
 Output view 345–347, 351
 OUTPUT_DIRECTORY
 parameter 227
 OutputRepresentation 318
 OWL (Ontology Web
 Language) 281

P

PaaS (Platform as a Service)
 204
 Package location field 219
 package.xml file 218
 packagetype attribute 218
 parent property 36
 partial representations 9, 187
 PasswordCallback 138
 Pattern class 59
 performance 185–192
 caching information 188–189
 compressing
 representations 186–187
 partial representations 187
 removing server-side session
 state 191–192
 streaming
 representations 185–186
 using conditional
 methods 190–191

persistence 197–199
 JDBC extension for 197–198
 Lucene extension for 198
 tips for 199
PKI (Public Key Infrastructure) 123
PkixSslContextFactory 128
Plain Old Java Object. *See POJO*
Platform as a Service. *See PaaS*
POJO (Plain Old Java Object) 384
POP_BASIC scheme 100, 131
POP_DIGEST scheme 100, 131
Popper, Karl 369
 portability, of SaaS 204–205
POST method 38, 56, 197, 224, 234, 256, 369, 390
 when invoked 395
 when to use 391
 preauthenticating calls 19
 Principal interface 141
 priori 123
 procedure call 366
 professional services 414
profileRef property 287
<profiles> section 329
 properties
 for applications 19–20
 for HTTP headers 399
Proxy-Authorization HTTP header 133
proxyChallengeResponse property 133
pseudoconnector 198
pseudoprotocols 9, 192, 194–195, 197
Public Key Cryptography 122
Public Key Infrastructure. *See PKI*
 purpose, of applications 14–15
PUT method 38, 94–95, 107, 224, 390
 and Amazon S3 service 232
 conditional methods 190
 not allowed by HTML 4 167

Q

Query class 229
queue service 234

R

range property 84
RangeService 23, 187

RBAC (Role-Based Access Control) 141
rc (release candidate) 327
RDF 281–293
 consuming linked data 291–293
 data model for 282–283
 exposing RDF resources 286–291
 representation variants 284–286
rdf:Description element 284
RdfClientResource 291
RdfConverter 290
RdfRepresentation 284, 290–291, 293
ReadableRepresentation 318
ReaderRepresentation 318
Realm class 48, 142
 reason phrase 393
 redirect keyword 395
RedirectedClient class 182
 redirections 180–184
 manual 181–182
Redirector class 182–184
Redirector class 181–184, 199, 249, 261–262, 319
Reference class 83, 282
reia 208, 218, 220, 234
reia.table.core.windows.net 233
reiabucket 230–231
 release candidate. *See rc*
release() method 31
Remote Procedure Call. *See RPC*
 remote service interfaces 244
RemoteServiceServlet 260–261
 repositories 414
<repositories> section 329
represent() method 5–6, 39, 41
Representation
 class 37, 82–87
 method 97
 parameter 293
Representation.expirationDate 188
Representation.size
 property 185
RepresentationInfo class 82–83, 318
RepresentationInfo.
 .modificationDate 188
RepresentationInfo.tag 188
 representations 9, 81–120
 classes for RESTful web APIs 394–395
digesting 148–149
HTTP content
 negotiation 113–119
 combining annotated interfaces and converter service 117–119
 configuring client preferences 116–117
 declaring resource variants for 115–116
 overview 113–114
JSON representations 105–108
 using Jackson extension 107–108
 using JSON.org extension 106–107
package 318
Representation class 83–87
RepresentationInfo class 82–83
template
 representations 108–112
 using FreeMarker extension 109–111
 using Velocity extension 111–112
Variant class 82–83
XML representations 87–105
 and XPath expressions 94–95
 applying XSLT transformations to 99–102
 namespaces for 95–97
 using DOM API 89–91
 using JAXB extension 102–104
 using SAX API 92–94
 validating against schemas 97–99
XmlRepresentation class
 for 88–89
Request class 193, 317, 350
request.clientInfo.accepted-CharacterSets 399
request.clientInfo.accepted-Encodings 399
request.clientInfo.accepted-Languages 399
request.clientInfo.accepted-MediaTypes 399
Request.getClientInfo()
 method 399
Request.hostRef.hostDomain
 property 58

- Request.hostRef.hostPort
 property 58
Request.hostRef.scheme
 property 58
Request.loggable property 358
Request.resourceRef
 property 28
Request.resourceRef
 .hostDomain property 58
Request.resourceRef.hostPort
 property 58
Request.resourceRef.scheme
 property 58
RequestBuilder class 246–248
requestEntityBuffering
 property 36
RequestInfo class 155
requirements
 analysis 373, 384
 for Android 263–267
 for RESTful web APIs
 classifying by priority 377–378
 collecting from sources 376
 describing system
 sequences 381
 describing usage
 scenarios 379
 gathering 373
Resolver interface 110
Resource class 30–31, 47, 318
resource model, for RESTful
 web APIs 384–385
resource package 318–319
resource variants, for HTTP con-
 tent negotiation 115–116
resource.get() method 234
resourceDomain property 58
ResourceException 37
ResourceInfo class 155–156
Resource-Object Mapping. *See*
 ROM
Resource-Oriented Analysis &
 Design. *See* ROA/D
resource-oriented paradigm 364
resourcePort property 58
resourceRules.xml file 237
resources 30–45, 362
 authorizing user actions
 for 145–146
 ClientResource class 35–38
 for RESTful web APIs 385–387
 in MailServerApplication 42–45
 in REST architecture
 style 362–364
 layer 383–384
 Resource class 30–31
 ServerResource class 31–35
 using Java annotations 38–41
resourceScheme property 58
ResourcesInfo class 155
Response
 class 317
 method 110, 164
 statuses for RESTful web
 APIs 393–394
response.age 399
Response.serverInfo.address
 property 58
Response.serverInfo.port
 property 58
Response.status property 181
Response#dimensions
 property 300
responseEntityBuffering
 property 36
REST
 and GWT 246
 and Semantic Web 280–281
 architecture style 365
 architecture, Restlet classes
 for 397
 interface 30, 398
REST architecture style 359–370
 and HTTP 365–366
 components in 364
 connectors in 364–365
 example of 365
 forms of web using 360–361
 resources in 362–364
 vs. RPC 366–370
RESTClient 355
RESTful method 82
RESTful web APIs
 analyzing requirements 378–381
 defining domain model 381
 describing system
 sequences 381
 describing usage
 scenarios 379
 designing solution 381–395
 defining allowed
 methods 390–392
 defining logical
 architecture 382–384
 defining representation
 classes 394–395
 defining response
 statuses 393–394
 defining URI space 387–390
 deriving resource
 model 384–385
 identifying and classifying
 resources 385–387
 gathering requirements 376–378
 classifying by priority 377–378
 collecting from sources 376
ROA/D methodology 372–376
 constructing solution 375
 elaboration phase 375
 inception phase 375
 overview 372–375
 transitioning project 376
Restlet
 Cloud 305
 contributing to future 310
 engine 321
 forge 303–304
 property 399
 Studio 305
Restlet API 315–320
 data package 317–318
 representation package 318
 resource package 318–319
 root package 316–317
 routing package 319
 security package 319–320
 service package 320
 util package 320
Restlet Apps 304–305
Restlet class 7, 14, 19, 30, 193,
 316, 397–399
 advantage 317
Restlet edition for GWT
 concepts 248–249
Restlet framework 8–12
 as library 72–73
 benefits of 9–10
 design 10
 design of 10
 installing 328
 platforms supported by 11–12
Restlet Internal Access Protocol.
 See RIAP
RESTLET_HOME variable 339–340
Restlet.class file 55
Restlet-annotated method 256

RestletFileUpload class 177
 RestletFrameworkServlet class 73
 restletLogLevel 358
 Result class 256
 retryAttempts property 36
 retryDelay property 36
 retryOnError property 36
 reverse proxying 319
 rewrite(*Representation*)
 method 184
 RIA (Rich Internet
 Applications) 242
 RIAP (Restlet Internal Access
 Protocol) 49
 client 55
 pseudoprotocol 194–196, 411
 RIAP_APPLICATION 412
 RIAP_COMPONENT 412
 RIAP_HOST 412
 Rich Internet Application. *See*
 RIA
 RNG (Relax NG) 97
 ROA/D (Resource-Oriented
 Analysis & Design)
 methodology 372–376
 constructing solution 375
 elaboration phase 375
 inception phase 375
 overview 372–375
 transitioning project 376
 Role object 210
 RoleAuthorizer class 144–145,
 320
 Role-Based Access Control. *See*
 RBAC
 roles, user 141–143
 Enroler interface 142–143
 groups for 142–143
 Principal interface 141
 ROM (Resource-Object
 Mapping) 385
 root package 316–317
 rootRef property 172–173
 RootServerResource class 33,
 37, 39–40, 159, 352
 refactor 40
 Router class 27, 29, 57–58,
 319
 routing package 319
 routing system 24–30
 filtering for processing 24–27
 URI-based routing 27–30
 routingMode property 29
 RowSetRepresentation class
 198

RPC (Remote Procedure Call)
 366
 vs. REST architecture
 style 366–370
 rulesFile parameter 237
 runOnUiThread method 270

S

SaaS (Software as a Service) 204
 portability of 204–205
 safe methods 391
 SAX API
 XML representations
 using 92–94
 SaxRepresentation class 87, 92,
 94, 99
 Schema class 97
 SDC (Secure Data Connector)
 206, 221, 235–240
 implementing 238–240
 in Google App Engine 240
 installing agent 236–238
 overview 235–236
 scheme 131
 sdcServerHost 237
 SDK (Software Development
 Kit) 244
 command 215
 SecretVerifier 136–138, 170, 320
 Secure Data Connector. *See* SDC
 Secure Socket Layer. *See* SSL
 security 121–150
 assigning roles to users 141–
 143
 Enroler interface 142–143
 groups for 142–143
 Principal interface 141
 authenticating users 129–141
 Authenticator class 134–135
 certificate-based
 authentication 139–141
 challenge-based
 authentication 135–136
 credentials in client 130–
 134
 verifying credentials 136–
 139
 authorizing user actions 143–
 147
 Authorizer class 143–144
 for particular
 resources 145–146
 MethodAuthorizer
 class 145

RoleAuthorizer class 144–
 145
 using Java security
 manager 146–147
 ensuring end-to-end integrity
 of data 147–150
 digesting without losing
 content 149–150
 representation
 digesting 148–149
 using Content-MD5
 header 148
 for communications 122–129
 enabling HTTPS 126–128
 generating certificate
 requests 125–126
 generating self-signed
 certificates 125
 importing trusted
 certificates 126
 SSL 122–124, 128–129
 storing keys and

serverKey.jks 125, 127
 serverPort property 58
 ServerResource class 5–6, 30–35
 ServerResource#get-
 Application() method 199
 Server-Sent Events. *See* SSE
 ServerServlet 260–261
 server-side extension, for
 GWT 259–262
 handling cross-domain
 requests 261–262
 with GWT-RPC 260–261
 server-side redirections 319
 server-side session state,
 removing 191–192
 server-side support, of Restlet edi-
 tion for Android 271–272
 Service class 61, 226, 320
 service package 320
 SERVICE_CLASS_NAME
 parameter 227
 SERVICE_URI parameter 227
 ServiceConfiguration.cscfg 216,
 219
 services, configuring 61–62
 Servlet engine
 as connector for
 components 68–70
 as container of
 applications 70–71
 Servlet extension for Java EE
 server 67–68
 ServletAdapter class 72
 ServletContext class 21
 servlet-mapping element 71
 servlet-name values 71
 session affinity 191
 Session Initiation Protocol. *See*
 SIP
 Set<String> interface 25
 setAttributes() 21
 setChallengeResponse
 method 232
 setOnResponse method 252
 setUserPrincipalClassName()
 method 139
 setVerifier() method 136
 setWrappedVerifier()
 method 136
 Simple Storage Service, Amazon.
 See Amazon S3
 SimpleWebMailActivity class 269
 Single Sign-On. *See* SSO
 Single-Entry Module Library
 option 347

SIP (Session Initiation
 Protocol) 268
 size
 optimization of editions 301
 property 84
 skip property 228
 SMTP_PLAIN scheme 100, 131
 Software Development Kit. *See*
 SDK
 solution design 373, 381
 sourceRepresentation
 property 99
 SPDY protocol 297–298
 split browsing 297
 Spring framework, XML config-
 uration for 63–67
 SSE (Server-Sent Events) 297
 SSL (Secure Socket Layer) 122–
 124
 custom settings for 128–129
 SSLContext 128
 SslContextFactory 126, 128,
 140
 sslContextFactory
 parameter 126
 SSO (Single Sign-On) 169
 stacks for Restlet 309–310
 Start after successful deployment
 option 219
 Start Menu folder 331
 start() method 7, 19, 51
 Status class 181
 Status.getThrowable()
 method 176
 StatusService class 23, 61, 174
 statusService property 175
 StatusService.getStatus
 (Throwables, Uniform-
 Resource) method 175
 stop() method 7
 streaming representations 185–
 186
 StreamRepresentation class 318
 String class 82, 84
 String method 144
 StringRepresentation 82, 318
 Subject DN (Subject Disting-
 guished Name) 123
 super.describeGet(methodInfo)
 157
 System.getSystemClassLoader()
 method 55

T
 table service for Windows
 Azure 233–235
 Tag class 386
 Target class 408
 targetTemplate property 183
 TaskService 23
 template representations 108–
 112
 using FreeMarker
 extension 109–111
 using Velocity extension 111–
 112
 TemplateRepresentation
 class 110
 TemplateRoute class 27–29
 testing 350–358
 debugging 357–358
 integration testing 355–356
 ClientResource 355–356
 cURL 355
 RESTClient 355
 unit testing 350–354
 JUnit 352–354
 TestNG 354
 TestNG 354
 testRestlet project 336
 third-party projects 308–310
 integration efforts 308–
 309
 stacks 309–310
 Thread.getContextClass-
 Loader() method 55
 TikaRepresentation class 198
 TLS (Transport Level
 Security) 122–124
 top property 229
 toString() method 39
 toText() method 39
 Totty, Brian 391
 TRACE method 391
 Tracer class 26, 28–29
 Transformer class 102
 TransformRepresentation
 class 99–100
 transformSheet property 99
 transient property 84
 transition phase 374
 trust models 123, 140
 trusted certificates,
 importing 126
 TrustManagers 140
 truststore 123–124, 126–128,
 140

truststorePassword
 parameter 126
truststorePath parameter 126
truststoreType parameter 126
TunnelService 167
 browser workarounds 23

U

UML (Unified Modeling Language) 19, 156, 283, 374
UML class 380, 388, 392
uncomment keyword 302–303
unified bean converter 300
Unified Modeling Language. *See* UML
Unified Process. *See* UP
uniform interface 9, 25, 316
UniformResource 318
unit testing 350–354
 JUnit 352–354
 TestNG 354
unzip.vbs 217
UP (Unified Process) 373
updateProduct method 229–230
URI (Uniform Resource Identifier) 251, 362
 fetch service for Google App Engine 221
 space for RESTful web APIs 387–390
URI-based routing 27–30
URLConnection class 221, 267
useAlphaComparator()
 method 172
useAlphaNumComparator()
 method 172
user agents 364
User-Agent header 114
util package 320

V

validating XML
 representations 97–99
validatingDtd property 97
Validator filter 27
Variant class 82–83
variants 413
 property 33
Vary header 221

Velocity extension 111–112
Verifier interface 134–136, 138, 320
versioning, for editions 327
Via header 221
virtual hosting 49, 58
 for deploying applications 57–61
VirtualHost 319

W

WADL (Web Application Description Language)
 converting documents to HTML 163–164
 overview 154–155
WadlApplication class 155–156
WadlServerResource class 156–163
 describing single resource with 162–163
 improving description of existing server resources with 158–162
 methods for 156–158
WadlApplication class 156–157, 159, 163–164
WadlDescribable interface 156
wadlRepresent(Request, Response) method 164
WadlRepresentation class 155
WadlServerResource class 156–163
 describing single resource with 162–163
 improving description of existing server resources with 158–162
 methods for 156–158
wantClientAuthentication parameter 126, 139
WAR file 14, 209, 212, 217
Web Application Description Language. *See* WADL
web applications 7, 9
web elements 166–177
 cookies 169–172
 error pages 174–176
 feeds 178–180
 consuming 180
 exposing 178–179
 file uploads 176–177

forms 166–168
serving file directories 172–174

Web resources 244
web.xml file 217, 260–261
webapps directory 217–218
WEB-INF/web.xml file 69
WebRowSet interface 198
WebSocket protocol 296–297
wiki 413–414
Windows Azure 214–220, 233–235
 configuring storage accounts 233
 deploying applications in 215–220
 overview 214–215
 using table service 233–235
Windows, installer for 331
windowsazurepackage 216, 218
WindowsAzurePackage.cspkg file 219
wrap method 251
WritableRepresentation 318
write() method 149–150
write(Writer) method 186, 277
writer.endDocument()
 method 93
WriterRepresentation 186, 277, 318

X

XdbServerServlet class 72
X-Forwarded-For header 221
XML configuration, for deploying applications 62–67
 with components 62–63
 with Spring framework 63–67
XML representations 87–105
 and XPath expressions 94–95
 applying XSLT transformations to 99–102
 in Restlet edition for GWT 252–254
namespaces for 95–97
using DOM API 89–91
using JAXB extension 102–104
using SAX API 92–94
validating against schemas 97–99
XmlRepresentation class for 88–89

XMLConstants class 95
XMLDecoder class 256
XMLHttpRequest object
 258
xmlns attribute 96
XmlRepresentation class 88–89,
 94–95, 97
-Xmx argument 264

XP (eXtreme
 Programming) 373
XPath expressions, and XML
 representations 94–95
XSD (W3C XML Schema)
 97
XSLT
 stylesheets 300

transformations to
 applying 99–102

Z

zip files 330–331
ZIP/JAR client 55

Restlet IN ACTION

Louvel • Templier • Boileau

In a RESTful architecture any component can act, if needed, as both client and server—this is flexible and powerful, but tricky to implement. The Restlet project is a reference implementation with a Java-based API and everything you need to build servers and web clients that integrate with most web and enterprise technologies.

Restlet in Action introduces the Restlet Framework and RESTful web APIs. You'll see how to easily create and deploy your own web API while learning to consume other web APIs effectively. You'll learn about designing, securing, versioning, documentation, optimizing, and more on both the server and client side, as well as about cloud computing, mobile Android devices, and Semantic Web applications.

What's Inside

- Written by the creators of Restlet!
- How to create your own web API
- How to deploy on cloud and mobile platforms
- Focus on Android, Google App Engine, Google Web Toolkit, and OSGi technologies

The book requires a basic knowledge of Java and the web, but no prior exposure to REST or Restlet.

The authors are founders and technical leads of Restlet Inc. and Restlet SAS. **Jérôme Louvel** is the creator of the Restlet Framework.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/RestletinAction



SEE INSERT

“Broad, deep, and example-driven.”

—From the Foreword by
Brian Sletten
Bosatsu Consulting

“Accurate, informative, and extremely useful.”

—Dustin Jenkins, National Research Council of Canada

“A must-have for RESTful web services Java developers.”

—Fabián Mandelbaum
NeoDoc SARL

“A broad reach for a perfectly minimalist framework.”

—Tal Liron, Three Crickets

“Thoroughly recommended ... helps the reader come to grips with REST.”

—Dave Pawson
Pawson Software Services, Ltd

ISBN 13: 978-1-935182-34-4
ISBN 10: 1-935182-34-X



9 7 8 1 9 3 5 1 8 2 3 4 4



MANNING

\$49.99 / Can \$52.99 [INCLUDING eBOOK]