Creating Evolvable Hypermedia Applications

Building

Hypermedia APIs with HTML5 & Node

O'REILLY®

Mike Amundsen

Building Hypermedia APIs with HTML5 & Node

With this concise book, you'll learn the art of building hypermedia APIs that don't simply run *on* the Web, but that actually exist *in* the Web. You'll start with the general principles and technologies behind this architectural approach, and then dive hands-on into three fully functional API examples.

Too many APIs rely on concepts rooted in desktop and local area network patterns that don't scale well—costly solutions that are difficult to maintain over time. This book shows system architects and web developers how to design and implement human- and machine-readable web services that remain stable and flexible as they scale.

- Learn the H-Factors for representing application metadata across all media types and formats
- Understand the four basic design elements for authoring hypermedia types
- Convert a simple read-only XML-based media type into a successful API design
- Examine the challenges and advantages of designing a hypermedia type with JSON
- Use HTML5's rich set of hypermedia controls in the API design process
- Learn the details of documenting, publishing, and registering media type designs and link-relation types

Purchase the ebook edition of this O'Reilly title at oreilly.com and get free updates for the life of the edition. Our ebooks are optimized for several electronic formats, including PDF, EPUB, Mobi, APK, and DAISY—all DRM-free. Purchase the ebook edition of this O'Reilly title at oreilly.com and get free updates for the life of the edition. Our ebooks are optimized for several electronic formats, including PDF, EPUB, Mobi, APK, and DAISY—all DRM-free.

Twitter: @oreillymedia
facebook.com/oreilly

US $24.99 CAN $26.99
ISBN: 978-1-449-30657-1

O'REILLY®
oreilly.com

Building Hypermedia APIs with HTML5 and Node

Mike Amundsen

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Building Hypermedia APIs with HTML5 and Node
by Mike Amundsen

Copyright © 2012 amundsen.com, inc.. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Simon St. Laurent
Production Editor: Melanie Yarbrough

Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Revision History for the First Edition:
2011-11-21 First Release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449306571* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Building Hypermedia APIs with HTML5 and Node*, the image of a rough-legged buzzard, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-30657-1

[LSI]

1321983647

"The human mind ... operates by association. With one item in its grasp, it snaps instantly to the next that is suggested by the association of thoughts, in accordance with some intricate web of trails carried by the cells of the brain."

—Vannevar Bush, 1945

"*If computers are the wave of the future, displays are the surfboards".*

—Ted Nelson, 1974

"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."

—Tim Berners-Lee, 1992

"Hypermedia is defined by the presence of application control information embedded within, or as a layer above, the presentation of information."

—Roy T. Fielding, 2001

"The WWW is fundamentally a distributed hypermedia application."

—Richard Taylor, 2010

Table of Contents

Foreword

You can't talk about something if you don't have the words.

The World Wide Web is driven by hypermedia: the ability of a document to describe its possible states, and its relationship to other documents. Hypermedia is not just a way of making websites that average people can use; it's a new style for distributed computing, powerful and flexible.

There's nothing new about the web technologies or the hypermedia concept: in another world, we could have been using hypermedia for distributed computing since the mid-1990s. Instead, we've been slow to adopt hypermedia for anything but consumer use. It's an easy concept to grasp intuitively—we all use the Web—but it's difficult to understand in a context of development.

Our problems stem from conceptual blocks. The Web invaded our everyday lives years before its architecture was formally described. We've spent the twenty-first century making gradual progress, coming up with new vocabulary to help developers come to terms with the power of the Web—power that was there all along.

The description of hypermedia you'll read in this book is, in my opinion, one of the biggest conceptual advances since Roy Fielding first defined the REST architectural style. Mike Amundsen has taken the blanket term "hypermedia" and taken it apart to see exactly what it can mean and how it works.

What makes a data format useful for some applications and not others? Why is HTML so versatile, even for nonconsumer applications, and where does it fall short? Under Mike's view of hypermedia, these questions have precise answers—answers that I hope will drive the next generation of web services and web-based technologies.

Mike has not only found the words to describe hypermedia, he's given voice to our intuitions about how it works.

—Leonard Richardson, November 2011

Preface

When you set out on your journey to Ithaca,
pray that the road is long, full of adventure,
full of knowledge.

- Constantine P. Cavafy

Hypermedia API Design

This book's primary focus is on designing hypermedia APIs. That may seem a bit strange to some readers. There are many books on programming languages, data storage systems, web frameworks, etc. This is not one of those books. Instead, this book covers the nature of the messages passed between client and server, and how to improve the content and value of those messages. I, personally, find this to be an exciting and fascinating area.

As of this writing anecdotal trends seem to indicate an ever-increasing reliance on APIs in web development. In general, this is a good thing. It means more and more developers are catching on to the notion that the World Wide Web is a great place to share not only data, but also services, a goal of those who championed the web in its early days.

However, I believe that this explosion of web APIs may lead us down a troublesome path. In my experience over the last few years, I have seen too many examples of implementations that rely on concepts of APIs rooted in desktop and local area network patterns that will not scale well at the WWW level, solutions still exhibiting brittleness that can lead to costly and frustrating maintenance issues as time goes by. In short, I don't see enough hypermedia in API offerings for the web.

This book is an attempt to improve the chances that new APIs added to the WWW will be easier to use and maintain over time, and that they will take their cue from those who were responsible for the discovery of the value of hypermedia linking; the codification of the HTTP protocol; and the implementation of HTML, Atom/AtomPub, and other native hypermedia formats that still drive the growth of the web today.

Intended Audience

The primary goal of this book is to increase both the quantity and quality of hypermedia content in use on the web. To that end, the audience for this text is two-fold.

First, this book is offered as a guide to system architects. Hopefully the text can be a valuable guide for those responsible for designing systems that rely on hypermedia to improve the evolvability and stability of long-lived implementations. When viewed as an integral part of system architecture, hypermedia provides a wealth of possibilities to architects. Hopefully this book will illustrate that, by treating hypermedia data as a key architectural component (rather than merely a payload to be pushed about by clients and servers), architects can increase future stability and flexibility of their systems.

Second, readers tasked with implementing clients and servers will find valuable advice and examples on how to deal with hypermedia messages themselves. Up to now, most books on web implementations have focused too often on the role of servers in dealing hypermedia. It is the author's view that this oversight too often results in improper client implementations that not only ignore, but often negate the value of hypermedia messages on the web. One of the key advantages of hypermedia as an architectural pillar is that hypermedia encourages clients to "code for the media type" instead of writing applications that treat messages as simple data. Writing hypermedia-aware clients is a skill that takes time to master. And while this book does not focus solely on writing hypermedia clients, the author hopes that it will show enough examples and advantages as to spur other, more talented individuals to establish new practices and techniques aimed at taking direct advantage of hypermedia.

What Is Not Covered

While the examples in this book use HTML5, Node.js, and CouchDB, this book should not be used as a source for learning these technologies. Astute readers may find the author's use of these tools—HTML5, Node.js, CouchDB—somewhat stilted and possibly, to some, blasphemous. The author makes no claims at expertise in these technologies. Instead, in the context of this book, they are used as tools for illustrating points about hypermedia design and implementation. The appendices list several good books on the technologies used in the writing of this text that the reader is encouraged to refer to for a more authoritative voice on these matters.

This book does not cover the details of the HTTP protocol and associated web standards. There is a wealth of writing available and the appendices reference important RFCs and other standards documents used while preparing this text. The reader will also find several book recommendations in the appendices well worth the time to read and become acquainted.

Finally, while the subject of the Representation State Transfer (REST) architectural style comes up occasionally, this book does not explore the topic at all. It is true that REST identifies hypermedia as an important aspect of the style, but this is not the case for the inverse. Increasing attention to hypermedia designs can improve the quality and functionality of many styles of distributed network architecture including REST. Readers who want to learn more about Fielding's style will find helpful recommendations in the appendices.

Contents of This Book

The book is designed to allow readers to jump around to sections they find interesting; you do not need to read it cover-to-cover in sequential order. There are a number of links within the chapters to point the reader to related material that may have been missed when skipping around in the text. Hopefully this format will also make the text more useful as a reference when the reader wants to refer back to content at a later date.

The general layout of the book is as follows:

Chapter 1: Understanding Hypermedia
: This is the conceptual chapter of the book. It provides some historical references for hypermedia, HTTP, and HTML, and then goes on to lay out the basic premise of the text including making a case for more hypermedia, offering an analysis of existing hypermedia content, and a suggested methodology for creating new hypermedia designs.

Chapters 2, 3, and 4: Implementations
: The middle chapters contain complete walk-throughs of fully functional hypermedia examples. These chapters are meant to lead the reader through the process of assessing an application scenario, selecting design elements, creating sample data, and implementing complete server and client solutions that meet the use case requirements. While the examples are kept relatively basic, they are still meant to convey most of the details the reader is expected to encounter when creating real-life production-ready solutions.

Chapter 5: Documenting Hypermedia
: This is the housekeeping chapter of the book. It provides tips on documenting media type designs and registering those designs with standards bodies such as the IANA, IEFT, and WC3. There is a section covering the concepts of Versioning and Extending hypermedia types as well as some general tips on good API and hypermedia designs.

Appendices
: This book contains a number of appendices. These are included as pointers to quoted and referenced materials as well as to hold additional content that did not fit well into the flow of the chapters. The information here may also be valuable for future reference after the reader has already completed the body of the book.

Coding Style for This Book

One of the reasons Node.js and CouchDB were selected for this book is that, from the beginning, these products are HTTP-aware. That means the software works well using the existing HTTP application protocol and in a state-less environment like the World Wide Web. As a result, there is very little friction between the components I created using Node and CouchDB, and the protocol used to communicate with those components.

It is also an advantage that these software systems all use the same front-facing programming language for scripting (Javascript). While not all readers will be proficient in Javascript, hopefully this single language format can reduce the need for mental context-switching when moving between client code, server code, and data storage implementation.

The important point, though, is that the *software* is not the focus for this book; it is merely the medium for the hypermedia message. You will likely find that many of the examples contain code that is either too brief or too fragile to run in a production environment. This is mostly a matter of expediency; I'm anxious to illustrate the details of the hypermedia, not the code used to implement that design. These designs will work well when implemented for any platform, using any language, running on any operating system. I suspect many readers will find better ways to implement these media type designs using their own languages and platforms, and that's all the better.

Conventions Used in This Book

The following typographical conventions are used in this book:

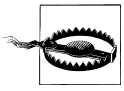
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Building Hypermedia APIs with HTML5 and Node* by Mike Amundsen (O'Reilly). Copyright 2012 O'Reilly Media, Inc., 978-1-449-30657-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://shop.oreilly.com/product/0636920020530.do

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgements

There are quite a number of people who deserve acknowledgement for the completion of this book.

Several people volunteered to review early drafts of this book and provided excellent feedback and suggestions. Thanks to Leonard Richardson, Erik Wilde, Ian Robinson, Jan Algermissen, Mike Kelly, Will Hartung, William Martinez Pomares, Erlend Hamnaberg, Darrel Miller, Glenn Block, David Zulke, Erik Morgensen, Kevin Burns Jr., Jonathan Moore, and Subbu Allamaraju. If the book is accurate, clear, and concise, that is likely due to the contributions of these individuals. Any remaining shortcomings are solely the responsibility of the author.

Julian Reschke, Mykyta Yevstifeyev, Frank Ellermann, and others were very helpful and generous as I worked to learn the details of IETF and IANA procedures and processes.

I'd like to thank Benjamin Young for introducing me to CouchDB and for all his efforts to teach me how to improve my understanding and coding of CouchDB. His willingness to spend one-on-one time to help me get past some hurdles was invaluable. If there are shortcomings or errors in the CouchDB code, it's most likely because I failed to grasp what Benjamin tried to teach me; my apologies to the reader and to Benjamin Young.

Simon St. Laurent, my editor, has been a great champion of this book. Without his tireless efforts, this book would never have seen the light of day. Thanks also to Melanie Yarbrough for all her proofreading and editing work.

I want to thank the organizers of CodeStock, Stir Trek, JAOO (aka Gotocon), and OSCON. My presentations at these and other events over the last few years has given me a chance to explore, refine, and correct initial concepts and methodologies.

I owe special thanks to all of the attendees of REST Fest 2010 and 2011. It was during very productive and enjoyable weekends in lovely Greenville, South Carolina, that I was able to first publicly describe and refine my ideas about analyzing and designing hypermedia.

I've benefited quite a bit from the conversations on the REST IRC channel on freenode.org. My thanks to all who hang out there for all of the great feedback and ideas.

Thanks, finally, to the moderators and members of the REST-Discuss list. Over the years I've posted many questions, assertions, and comments to that list in an effort to learn more about Fielding's style and HTTP implementation in general. In most cases, many of the correct things I learned about REST and HTTP were a result of my REST-Discuss experience.

CHAPTER 1
Understanding Hypermedia

There is no end to the adventures that we can have if only we seek them with our eyes open.

- Nehru

Designing scalable, flexible implementations that live on the web relies on many concepts, technologies, and implementation details. Understanding the history behind the way the World Wide Web (WWW) works today and the various standards and practices created to support it is an essential part of developing skills as a web architect and hypermedia designer.

In addition to understanding the technologies behind the Web, designers also need to be aware of the differences between implementing applications on a distributed network that leverages coarse-grained message protocols like HTTP that span multiple platforms, programming languages, and storage systems, and more traditional local networked applications where most components in the network share similar storage, programming, and operating details. Programming for distributed hypermedia environments usually means that message transfers must carry more than just data; they must carry additional information including metadata and higher-level application flow control options. The Web thrives on this style of rich hypermedia, and it is important to design APIs that support this method of sharing understanding of the data sent between network participants.

The way in which hypermedia information is shared varies between data formats, but the actual hypermedia concepts—links, templated queries, idempotent updates, etc.—are the same across the Web. Having a clear understanding of the various hypermedia factors that can be expressed within a message is an essential part of developing the skills needed to implement successful hypermedia APIs.

Finally, the implementation details of creating hypermedia types include selecting an appropriate data format and state transfer style, expressing the application domain details properly within the format, and determining the way in which application flow

control options are exposed in responses. These basics of hypermedia design are similar regardless of data to be shared or the application domain involved.

This chapter covers some key technologies that make the Web possible, the importance of hypermedia as an architectural approach, and the concepts and details of hypermedia designs. Armed with this understanding the reader will have the tools necessary to build scalable, flexible implementations that do not simply run *on* the Web but that actually exist *in* the web in a way that acknowledges, as stated by Richard Taylor, that the "WWW is fundamentally a distributed hypermedia application."

HTTP, MIME, and Hypermedia

Designing hypermedia applications for the Web is just that: a design process. This process depends on a handful of important standards and technologies. Chief among these are the protocols, message standards, and message content that allows participants to drive the system forward, often toward some designated goal (completing a search, purchasing an item online, etc.).

The most commonly used protocol for transferring content on the Web today is HTTP. Initially designed as a read-only protocol for exchanging HTML documents, HTTP quickly expanded from a file-based transfer protocol into a more generalized protocol that supports both read and write operations across multiple intermediaries that allow real-time negotiation of representation formats for a wide range of information stored on servers.

The ability to represent data instead of just mirror files derives from HTTP's use of the Multipurpose Internet Mail Extensions (MIME) media type system. Originally created for supporting electronic mail transfers, the MIME typing standard allows HTTP transfers to support a wide range of data formats including ones that are designed specifically for transferring application-related requests such as search parameters and data storage operations. Also important is the built-in ability to support new media types over time in order to support new, unanticipated uses for data transfer on the Web.

The creation of the Web was heavily influenced by the notion of hypermedia and the ability to link related material and easily follow these links in real time. To this end, media types have been designed to natively support hypermedia controls as a way to enable client applications to make selections found within responses and drive the state of the application to the desired outcome. This allows client applications to discover the specific controls within the media type message with which to modify the state of the Web (or at least that client's view of the Web). It is hypermedia, and the design and implementation of it, that makes the Web a unique and powerful environment for building distributed network applications.

HTTP Is the Transfer Protocol

The first version of HTTP (documented in 1991 as HTTP/0.9) was a simple read-only protocol. It allowed clients to send a request string made up of the letters GET followed by a space and a document address (what we know as a URI today) and servers could then respond by returning the associated document. HTTP/0.9 had no metadata headers and the response was always assumed to be in HTML form.

By 1992, a Basic HTTP (considered a full specification) was documented. This version included several new methods including HEAD, PUT, POST, and DELETE. Other methods such as CHECKIN, CHECKOUT, LINK, UNLINK, SEARCH, and several others are no longer in the specification today. This document included the concept of response codes (1xx, 2xx, 3xx, 4xx, 5xx), an early form of content negotiation, and meta-information (later known as HTTP Headers).

Although the 1992 document was never officially released through a governing body (IETF, etc.), over the next few years, browser clients and web servers extended the read-only features of HTTP/0.9 to include many of the headers and additional request methods identified as Basic HTTP. The results of these additions were documented in 1996 in RFC1945 and named HTTP/1.0. Not long after RFC1945 was released, in January of 1997, HTTP/1.1 was documented and released as RFC2068. This refined many of the implementation details of HTTP as we know it today. An additional update was released in June of 1999 as RFC2616. This is the version of HTTP that is most commonly deployed today.

Throughout the history of specifying HTTP, the protocol was always designed to work as a client-initiated, stateless protocol for transferring messages between parties running over TCP/IP. There are other transfer protocols including FTP, BitTorrent, and rsync, but HTTP has remained the dominant transfer protocol for the Web. This focus on stateless transfer as well as HTTP's support for negotiating the format used to represent server data (via the Accept and Content-Type headers) are key to understanding how to best utilize HTTP and design efficient and effective applications for the Web. Architects and developers who create applications that run counter to these design principles not only ignore the strengths of HTTP but also add unnecessary complexity and complications to their implementations.

MIME Is the Media Type Standard

HTTP's standard for defining the body of the message sent between parties is based on the MIME standard (RFC2046). Although this standard was designed to support exchanging messages via email, MIME was adopted by HTTP in order to support the notion of resource representations and to allow for future extensibility.

One of HTTP's important concepts is the idea that responses are only *representations* of the actual data. For example, under the FTP (File Transfer Protocol) specifications, the goal is to send an exact copy of the data between parties. However in HTTP, servers

are free to represent the data in various ways and clients are encouraged to inform servers which representation formats are preferred. In order to support this additional feature, the MIME standard is used to indicate the current representation format.

MIME Type, Media Type, and Content Type

The terms MIME Type, Media Type, and Content Type as often used in similar ways. The term MIME comes from the initial RFCs describing media type handling for SMTP. Part two of that document collection regarding MIME (RFC2046) carries the sub-title of "Media Type." Subsequent RFC documents (e.g. RFC4288) refer to Media Type as the object of public registration for use in MIME and other Internet protocols. "Content Type" is the name of the HTTP Header that carries the media type information for the response message. Usually, when people use any one of these three phrases, they are referring to the media type registration string (e.g. application/xml, text/plain, etc.). Throughout this book the phrase "Media Type" is most often used unless there is a reference directly to the HTTP Header (Content-Type) or an historical reference to the original standards (MIME).

HTTP's support for varying response representations via a media type indicator opens the door to using the message as a key component in web architecture. Messages no longer need to be relegated to simply carrying raw data. Instead, designers and architects can leverage this opportunity to create new message formats and standards that can allow responses and requests to convey not just the raw data, but also metadata about the content. It also means that formats can be created to support very specific purposes independent of the application in use or the data that is transferred between parties.

For example, the same set of data points could be represented for use in a spreadsheet (text/csv), for display in a tabular view (text/html), or as a graphical pie chart display (image/png). Data sent from client to server can be represented as simple name-value pairs (application/x-www-formurlencoded), as plain text (text/plain), or even as part of a multiple-format collection (multipart/form-data).

There are more than a dozen multipart media types registered with the Internet Assigned Numbers Authority (IANA). These types are designed to support multiple unique media types within the same message body, each separated by a boundary. The HTTP/1.1 spec also defines message/http and application/http as similar container media types. These container media types are not covered in this book, but their very existence and use is evidence of flexibility and extensibility of the MIME media type standard.

The reliance on MIME media types also points to an important aspect of HTTP's message model: it was designed to send coarse-grained messages. Unlike some transfer

protocols whose aim is to send the smallest data packets possible, HTTP is concerned with including as much descriptive information as possible with each message, even if this means the message is longer than it needs to be. While there are some efforts underway to reduce message sizes (mostly by shortening or compressing HTTP Headers), designers and architects are free to use the body of the message to carry whatever is deemed important.

This freedom to design new message bodies for HTTP via the MIME standard leads to another unique aspect of the use of HTTP on the Web: in-message hypermedia. HTTP and MIME together make it not only possible, but common to include hypermedia information such as links and forms directly in the body of the response message. It is this ability to include hypermedia controls within messages that makes HTTP so well suited for use in distributed networks like the World Wide Web.

Hypermedia Is the Engine

The concept of hypermedia has been with us for quite a while. Vannevar Bush's 1945 article "As We May Think" envisioned a device (the "Memex") that allowed researchers to discover and follow links between topics and phrases in projected documents. Doug Engelbart's 1968 NLS (oN-Line System), one of the first graphical computer systems, used a new mechanical pointer device dubbed "the mouse" and allowed users to click on links to display related data on the screen. Similar examples of enabling links via computer display cropped up in the 1970s and 1980s. Ted Nelson coined the terms "hypertext" and "hypermedia" in the mid-1960s, and his work "Computer Lib/Deam Machines," published in 1974, is considered by many to be the first to establish the notion of "surfing the 'net" and cyberculture in general.

From links to controls

The initial concept for hypermedia was as a read-only link between related items and to this day, this is the most common way hypermedia is used on the Web. For example, many media types only support read-only links between elements. However, with the introduction of graphical user interfaces based on the use of Englebart's mouse as a way to activate elements of the interface (including buttons), the idea of using links as a way to perform other actions (sending a message, saving data, etc.) became accepted.

The development of HTTP mirrored this development from read-only (HTTP/0.9) to read/write linking (HTTP/1.0 and 1.1). Along the way, the de facto media type for HTTP (HTML) developed to include controls within messages that allowed users to supply arguments and send this data to remote servers for processing. These hypermedia controls included the `FORM` and `INPUT` elements among others. This ability to support not only navigational links (HTML anchor tags) and in-place rendering of related content (e.g. the `IMG` tag) but also parameterized queries and write operations helped HTML be become the *lingua franca* of the Web.

Hypermedia types

HTML's success as a media type on the web is due in large part to its unique status as a media type that supports a wide range of hypermedia controls. Unlike plain text, XML, JSON, and other common formats, HTML has native support for hypermedia. HTML is, in effect, a Hypermedia Type. There are other media types that share this distinction; ones that contain native support for some form of hypermedia linking. Other well-known types include SVG (read-only), VoiceXML, and Atom.

A Hypermedia Type is a media type that contains native hyperlinking elements that can be used to control application flow.

Media types that exhibit native hypermedia controls can be used by client applications to control the flow of the application itself by activating one or more of these hypermedia elements. In this way, hypermedia types become, as Fielding stated, "the engine of application state." This use of MIME media types to define how data is transferred and also how application flow control is communicated, is an essential aspect to building scalable, flexible applications using HTTP on the Web.

Programming the Web with Hypermedia APIs

Knowing about HTTP, MIME, and hypermedia is the easy part. Using them to design and implement stable, flexible applications on the Web is something else. The Web poses unique challenges to building long-lived applications that can safely evolve over time. Many web developers and architects have experienced these challenges firsthand:

- Updating server-side web APIs only to learn that client applications no longer work as expected without undergoing code updates.
- Moving long-lived server applications to a new DNS name (e.g. from www.example.org to www.new-example.org) and having to completely rewrite all of the API documentation as well as update all existing client code.
- Implementing new or modified process flow within the server-side application and discovering that existing clients break when encountering the new rules, ignore the rules, or, worse, continue to execute their own code in a way that creates invalid results on the server.

These challenges are sometimes mistakenly attributed to the nature of HTTP (the most common transfer protocol in use on the Web to date). However, implementations that rely on SOAP-based messages sent directly over TCP/IP are just as likely to experience the failures listed here. Instead these difficulties, and others like them, stem from the way information is shared between parties on the network.

The most common pattern for crafting and shipping messages is to serialize existing programming objects (e.g. classes and structures) into a common format (XML, JSON, etc.) and send the content to another party. This results in an architecture based on simple data and smart applications. Applications that must be constantly kept in sync with each other to make sure their understanding of the data is always compatible.

Type-Marshaling vs. Object Serialization

Throughout this section of the text, you will see the phrases "Type-Marshaling" and "Object Serialization" used interchangeably to mean converting an object's state into a byte-stream and shipping that byte-stream to another party that can then reconstitute the bytes into a copy of the original object. For some programming environments (e.g. Python), these phrases are considered equivalent. However, for others (e.g. Java) they carry slightly different meaning. In Java, "Type-Marshaling" means not just converting the state of the object, but also its basic coding.

For the purposes of this discussion, the details of what is shipped across the wire are not as important as the pattern of converting an internal object into a message and recreating a copy of that object upon receipt of the message.

This method of marshaling internal types over HTTP can easily lead to brittle and inflexible implementations on the Web. Often, introduction of new arguments for requesting data, new addressees to which requests must be targeted, and/or new data elements in the response messages will cause a mismatch between parties that can only be resolved by reprogramming one or more participants on the network. This is the antithesis of how the Web was designed to work. Reprogramming participants on the Web may be possible when there are only a few parties involved, but it does not scale up to the thousands and millions of participants that interact on the Web today.

In this section, the drawbacks of various forms of type-marshaling are explored and an alternative approach to message design based on hypermedia is identified.

The Type-Marshaling Dilemma

Probably the most common model for web programming today is to serialize internal object types ("customer," "order," "product," etc.) in a common data format (XML, JSON, HTML, etc.) and pass them back and forth between client and server. Most of the available web frameworks encourage this style of programming with built-in serializers and other helpers that make it easy to publish internal objects. Examples of these type-marshaling helpers include schema documents that can be generated from existing source code; run-time services that automatically respond to URIs that include type names; response bodies that contain attributes or elements containing type names to aid in automated serialization selection; and routing rules that use the HTTP Accept and Content-Type headers as object type indicators.

While these patterns are convenient, they had a number of drawbacks. For example, they are very server-centric; they meet the needs of server developers but usually leave client programmers to fend for themselves. Also, by focusing on internal types kept on the server, these patterns introduce risks to client code that is built independent of the server code. Changes to any internal objects can easily introduce changes to public interfaces (shared schema, URIs, payload details, and/or media type definitions).

Below is a short review of each of these popular approaches to sharing data over HTTP along with some commentary on the appeal and shortcomings of these techniques.

Shared schema

Probably the best understood method is to publish a detailed schema document that lists arguments and interaction details. This is the way SOAP was designed to work. The advantage of this approach is that it provides clear description for all parties. The downside is that it most often is used as a way to express the details of private objects (almost always on the server). When these objects change, all previously deployed clients can become broken and must be rebuilt using the newly published schema document. While it is true that this compile-time binding is not a requirement of the SOAP model, to date no major web libraries exist for clients and servers that treat this schema information in a dynamic way that scales for the Internet.

```
<?xml version="1.0"?>
<definitions name="StockQuote"
  targetNamespace="http://example.com/stockquote.wsdl"
  xmlns:tns="http://example.com/stockquote.wsdl"
  xmlns:xsd1="http://example.com/stockquote.xsd"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <types>
    <schema
      targetNamespace="http://example.com/stockquote.xsd"
      xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
.....
</definitions>
```

URI construction

Another common solution is to try to make the URI carry type information. This is a popular solution since most frameworks can easily generate these URIs from metadata within the source code. This method also makes things relatively easy for frameworks that use templating and other processing based on the type. Of course the biggest

drawback is that you lose control of your public URI space. Typing via URIs means private source code decides what your URIs look like and any change to source code threatens to invalidate previously published (and possibly cached) URIs.

```
http://www.example.org/orders/123

http://www.example.org/customer/big-co/orders/123

http://www.example.org/users/mike/address/home
```

Payload decoration

Developers who want to remain in control of their URIs can opt for the another common typing option: decorating the payload. This is usually done by adding a type element to the JSON or XML body or, in rare cases, adding a `<meta>` tag or `profile` attribute to HTML documents. This approach has the distinct advantage of allowing frameworks to sniff the contents of the payload in order to find the type hint and process the data accordingly. The primary drawback of this method, however, is that the payload is now tightly bound to source code and, again, changes to the source code on the server will modify response payloads and may invalidate incoming request payloads from clients who have no idea the server has changed the details of the typed object.

```
{
  "__type":"Circle:#MyApp.Shapes",
  "x":50,
  "y":70,
  "radius":10
}
```

Narrow media types

Another solution is to create a custom public media type for each private object on the server. The advantage here is that you have full control over both the public URI and the payload (although most frameworks link a private class to these custom media types). The downfall of this approach is that it merely pushes the problem along; in this scenario a single application could generate dozens of new media types. It would be an incredible feat for an independent client application to try to keep up with recognizing and parsing the thousands of media types that might be produced in a single year.

```
...
@PUT
@Consumes("application/stockquote+xml")
public void createStock(Stock stock ) {
...
}
...
@Provider
@Produces("application/stockquote+xml")
Public class StockProvider implements MessageBodyWriter<Stock> {
```

```
...
}
```

As is probably evident to the reader, type-marshaling is a difficult challenge for distributed networks where client and server applications are built, modified, and maintained independently, and where public-facing connectors may be expected to be maintained in production for a decade or more. The solutions illustrated above all fall short of supporting this kind of environment because they all suffer from the same flaw: they all attempt to solve the wrong problem.

The question architects and designers of distributed network applications need to ask themselves is not, "How can a server successfully export its private objects in a way that clients can see and use them?" Instead the question that should be asked is, "How can a server and client share a common understanding of the payloads passed between them?"

The short answer is to stop trying to keep clients and servers in sync by working out ways to share private types. Instead what is needed is a technique for describing data in a way that is not bound to any internal type, programming language, web framework, or operating system.

And one answer is to look beyond payloads based on marshaled types and toward payloads based on the principles of hypermedia.

The Hypermedia Solution

Relying on payloads based on hypermedia designs avoids the common pitfalls of the type-marshaling dilemma. That's because hypermedia payloads carry more information than just the data stored on the server. Hypermedia payloads carry two types of vital metadata: metadata about the data itself and metadata about the possible options for modifying the state of the application at that moment. Both are important to enabling stable and flexible distributed network applications.

By committing to message designs without the constraints of a particular internal type system, designers are free to craft payloads to more directly address the needs of the problem domain. By creating messages that contain not just data points but also information about the data (e.g. shared labels, hierarchies and relationships), both client and server can increase the shared understanding of the information passed between them.

Finally, by crafting responses that include data (and its metadata); information about the current state of the application itself; and possible transitions available at the present moment, distributed network applications can offer interfaces—interfaces that take advantage of new state transitions and new information flows that may appear over the lifetime of the application—to both human and machine-driven agents. This type of message design places stability and flexibility at the forefront of the architecture.

Metadata about the data

Hypermedia messages contain not only the data requested but also metadata. To make this point clear, consider a server that offers data on users defined for a network application. The server might return the data in the following form:

```
darrel,admin,active
mike,manager,suspended
...
whartung,user,pending
```

Note that the data contains no descriptive information, or no metadata. This works if both the client and server have a complete shared understanding over all the possible requests and responses ahead of time. It works if a human is assumed to be able to infer the missing details. However, in a distributed network where independent individuals might be able to create new applications to access this same data—individuals that might not know all of the details of every shared request and response—this raw data can be a problem.

Hypermedia designs can send additional metadata that describes the raw data. Here is the same data marked up using the HTML hypermedia format:

```
...
<ul class="users">
  <li class="user">
    <a href="..." rel="user">darrel</a>
    <span class="role">admin</span>
    <span class="status">active</span>
  </li>
  <li class="user">
    <a href="..." rel="user">mike</a>
    <span class="role">manager</span>
    <span class="suspended">suspended</span>
  </li>
  ...
  <li class="user">
    <a href="..." rel="user">whartung</a>
    <span class="role">user</span>
    <span class="suspended">pending</span>
  </li>
</ul>
...
```

The example here is simplistic (this is just one possible way to add metadata to the reponse), but it conveys the idea. Hypermedia designs include more than raw data; they include metadata in a form easily consumed by both humans and machines. The details of selecting an appropriate data format to carry the metadata is part of the design process that will be explored later in this chapter (see "Hypermedia Design Elements" on page 20).

Metadata about the application

Marking up the raw data is only part of the process of designing hypermedia. Another important task is to include metadata about the state of the application itself. It is the availability of application metadata in the message that turns an ordinary media type into a hypermedia type. This metadata provides information to clients on what possible actions can be taken ("Can I create a new record?", "Can I filter or sort the data?", etc.). This kind of application metadata allows the client to modify the state of the application, and to drive the application forward in a way that gives the client the power to add, edit, and delete content, compute results, and submit filters and sorts on the available data. This is hypermedia. This is the engine of application state to which Fielding refers in his dissertation.

Below are some examples of application metadata in an HTML message. You can see a list of users (as in the previous example) along with navigation, filtering, and search options that are appropriate for this response (and the identified user making the request).

```
...
<ul class="users">
  <li class="navigation">
    <a href="..." rel="next-page">next-page</a>
  </li>
  <li class="navigation">
    <a href="..." rel="last-page">last-page</a>
  </li>
  <li class="user">
    <a href="..." rel="user">darrel</a>
    <span class="role">admin</span>
    <span class="status">active</span>
  </li>
  <li class="user">
    <a href="..." rel="user">mike</a>
    <span class="role">manager</span>
    <span class="suspended">suspended</span>
  </li>
  <li class="user">
    <a href="..." rel="user">whartung</a>
    <span class="role">user</span>
    <span class="suspended">pending</span>
  </li>
  ...
</ul>
<!-- defined filters -->
<ul class="queries">
  <li class="query"><a href="..." rel="admins">admins</a></li>
  <li class="query"><a href="..." rel="pending">pending</a></li>
  <li class="query"><a href="..." rel="suspended">suspended</a></li>
  ...
</ul>
<!-- user search -->
<form name="user-search" action="...">
  <input name="search-name" value="" />
```

```
  <input type="submit" />
</form>
...
```

As can be seen in this example, a set of HTML anchor tags and an HTML form have been included in the server's response. This information lets the client application know that there are ways to navigate through the list, filter the data, and search it by username. Note that the navigation links only include "next" and "last" options since "previous" and "first" are not appropriate at the start of the list. Good hypermedia designs make sure the application metadata is context-sensitive. Just as human users can become confused or frustrated when offered too many options, quality hypermedia design includes making decisions on the most appropriate metadata to provide with each response.

Most of the examples in this section use the HTML media type, a well-known format. But HTML is not the only possible way to provide hypermedia to clients. In fact, HTML is lacking in some key hypermedia elements that might be important in your use cases. There are a handful of hypermedia elements that you can use to express application metadata. HTML has many; others appear in different media types. In order to allow clients to drive the application state, these elements must appear within the server responses. Fielding calls these "affordances." In this book they are called "Hypermedia Factors." These factors are the building blocks for hypermedia APIs.

Summary

This section identifies one of the common pitfalls of implementing applications on the Web: the type-marshaling dilemma. Many web applications today suffer from this problem, often because the programming languages, frameworks, and editors in use today encourage it. However, a more stable and flexible approach is available using hypermedia-rich messages as the primary way to share understanding between clients and servers. Unlike data-only messages based on type-marshaling, hypermedia messages contain the raw data, metadata about that data, and metadata about the state of the application.

The next section describes a set of abstract factors that make up a set of building blocks for designing hypermedia messages. These building blocks are called H-Factors.

Identifying Hypermedia : H-Factors

Designing hypermedia messages involves deciding how to represent the requested data (including metadata *about* the requested data) as well as deciding how to represent application metadata such as possible filters, searches, and options for sending data to the server for processing and storage. The details of representing the application metadata is handled by hypermedia elements within the message. The actual message el-

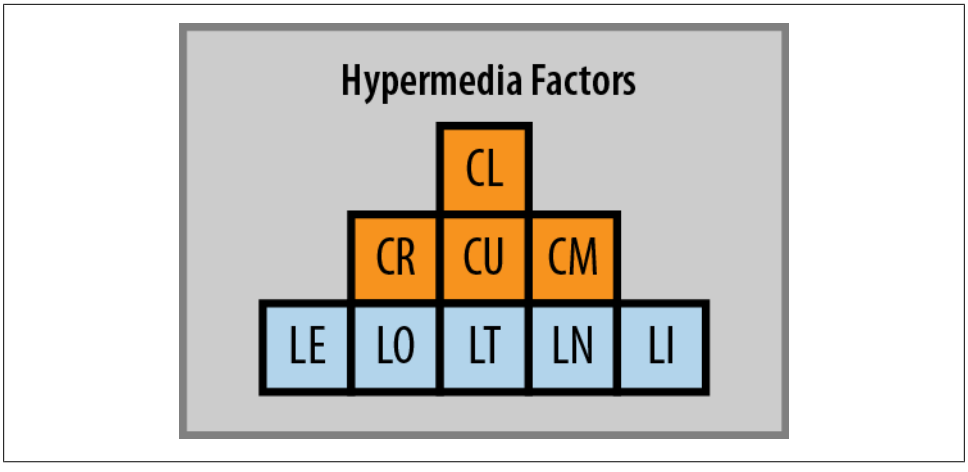


Figure 1-1. H-Factor Diagram

ements, attributes, etc. used to represent these options vary depending on the base format and the registered media type in use. However, despite these variances, the abstract options represented are the same across all media types and format. These are referred to in this book as "Hypermedia Factors" or "H-Factors."

Table 1-1. H-Factor Table

Links	LE	Embed Links
	LO	Outbound Links
	LT	Templated Links
	LI	Idempotent Links
	LN	Non-Idempotent Links
Control Data	CR	Read Controls
	CU	Update Controls
	CM	Method Controls
	CL	Link Annotation Controls

There are nine H-Factors, and they can be placed into two groups: link factors (represented as LO, LE, LT, LI, and LN) and control factors (represented as CR, CU, CM, and CL). The five link factors denote specific linking interactions between client and server: Outbound, Embedded, Templated, Idempotent, and Non-Idempotent. The remaining four control factors provide support for customizing metadata details (e.g. HTTP Headers) of the hypermedia interaction: Reads, Updates, Method, and Link Annotations.

Each of the H-Factors provide unique support for hypermedia designs. In this way, H-Factors are the building blocks for adding application hypermedia to messages. It is

not important that your design use every one of these factors. In fact, to date, there is no widely used hypermedia design that incorporates all of these factors. What is important, however, is knowing the role each factor plays and how it can be used to meet the needs of your particular implementation.

Below are brief descriptions of each of these factors along with examples from existing media types and formats.

Link Factors

Link factors represent opportunities for a client to advance the state of the application. This is done by activating a link. Some links are designed to update a portion of the display content with new material (LE) while other links are used to navigate to a new location (LO). Some links allow for additional inputs for read-only operations (LT), and some links are designed to support sending data to the server for storage (LI and LN).

Below are examples of each of the link factors identified here.

Embedding Links (LE)

The LE factor indicates to the client application that the accompanying URI should be de-referenced using the application level protocol's read operation (e.g. HTTP GET), and the resulting response should be displayed within the current output window. In effect, this results in merging the current content display with that of the content at the other end of the resolved URI. This is sometimes called transclusion. A typical implementation of the LE factor is the IMG markup tag in HTML:

```
<img src="..." />
```

In the above example, the URI in the `src` attribute is used as the read target and the resulting response is rendered inline on the web page.

In XML, the same LE factor can be expressed using the `x:include` element:

```
<x:include href"..." />
```

Outbound Links (LO)

The LO factor indicates to the client application that the accompanying URI should be de-referenced using the application level protocol's read operation, and the resulting response should be treated as a complete display. Depending on additional control information, this may result in replacing the current display with the response or it may result in displaying an entirely new viewport/window for the response. This is also known as a traversal or navigational link.

An example of the LO factor in HTML is the A markup tag:

```
<a href="...">...</a>
```

In a common web browser, activating this control would result in replacing the current contents of the viewport with the response. If the intent is to indicate to the client application to create a new viewport/window in which to render the response, the following HTML markup (or a similar variation) can be used:

```
<a href="..." target="_blank">...</a>
```

Templated Links (LT)

The LT factor provides a way to indicate one or more parameters that can be supplied when executing a read operation. Like the LE and LO factors, LT factors are read-only. However, LT factors offer additional information in the message to instruct clients on accepting additional inputs and including those inputs as part of the request.

The LT element is, in effect, a link template. Below is an example LT factor expressed in HTML using the `FORM` markup tag:

```
<form method="get" action="http://www.example.org/">
   <input type="text" name="search" value="" />
   <input type="submit" />
</form>
```

HTML clients understand that this LT requires the client to perform URI construction based on the provided inputs. In the example above, if the user typed "hypermedia" into the first input element, the resulting constructed URI would look like this:

```
http://www.example.org/?search=hypermedia
```

The details on how link templates (LT) are expressed and the rules for constructing URIs depends on the documentation provided within the media type itself.

Templated links can also be expressed directly using tokens within the link itself. Below is an example of a templated link using specifications from the URI Template Internet Draft:

```
<link href="http://www.example.org/?search={search}" />
```

Idempotent Links (LI)

The LI factor provides a way for media types to define support for idempotent submissions to the server. These types of requests in the HTTP protocol are supported using the PUT and DELETE methods. While HTML does not have direct support for idempotent submits within markup (e.g. `FORM method="PUT"`), it is possible to execute idempotent submits within an HTML client using downloaded code-on-demand.

```
<script type="text/javascript">
function delete(id)
{
  var client = new XMLHttpRequest();
  client.open("DELETE", "http://example.org/comments/"+id);
}
</script>
```

The Atom media type implements the LI factor using a link element with a relation attribute set to "edit" (`rel="edit"`):

```
<link rel="edit" href="http://example.org/edit/1"/>
```

Clients that understand the Atom specifications know that any link decorated in this way can be used for sending idempotent requests (HTTP PUT, HTTP DELETE) to the server.

Non-Idempotent Links (LN)

The LN factor offers a way to send data to the server using a non-idempotent "submit." This type of request is implemented in the HTTP protocol using the POST method. Like the LT factor, LN can offer the client a template that contains one or more elements that act as a hint for clients. These data elements can be used to construct a message body using rules defined within the media type documentation.

The HTML `FORM` element is an example of a non-idempotent (LN) factor:

```
<form method="post" action="http://example.org/comments/">
  <textarea name="comment"></textarea>
  <input type="submit" />
</form>
```

In the above example, clients that understand and support the HTML media type can construct the following request and submit it to the server:

```
POST /comments/ HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Length: XX

comment=this+is+my+comment
```

It should be noted that the details of how clients compose valid payloads can vary between media types. The important point is that the media type identifies and defines support for non-idempotent operations.

Control Factors

Control factors provide support for additional metadata when executing link operations. The possible metadata elements (and their values) can vary between supported protocols (FTP, HTTP, etc.) as does the details for communicating this link metadata. For example, in HTTP, this is accomplished through HTTP Headers. Regardless of the mechanism, control factors fall into four categories: Read, Update, Method, and Link Annotation.

What follows is a brief discussion of Control Factors along with examples.

Read Controls (CR)

One way in which media types can expose control information to clients is to support manipulation of control data for read operations (CR). The HTTP protocol identifies a number of HTTP Headers for controlling read operations. One example is the `Accept-Language` header. Below is an example of XInclude markup that contains a custom `accept-language` attribute:

```
<x:include href="http://www.exmaple.org/newsfeed" accept-language="da, en-gb;q=0.8,
en;q=0.7" />
```

In the example above, the hypermedia type adopted a direct mapping between a control factor (the `accept-language` XML attribute) and the HTTP protocol header (`Accept-Language`). There does not need to be a direct correlation in names as long as the documentation of the hypermedia design provides details on how the message element and the protocol element are associated.

Update Controls (CU)

Support for control data during send/update operations (CU) is also possible. For example, in HTML, the `FORM` can be decorated with the `enctype` attribute. The value for this attribute is used to populate the `Content-Type` header when sending the request to the server.

```
<form method="post"
  action="http://example.org/comments/"
  enctype="text/plain">
  <textarea name="comment"></textarea>
  <input type="submit" />
</form>
```

In the above example, clients that understand and support the HTML media type can construct the following request and submit it to the server:

```
POST /comments/ HTTP/1.1
Host: example.org
Content-Type: text/plain
Length: XX

this+is+my+comment
```

Method Controls (CM)

Media types may also support the ability to change the control data for the protocol method used for the request. HTML exposes this CM factor via the `method` attribute of the `FORM` element.

In the first part of the example below, the markup indicates a send operation (using the POST method). The second part uses the same markup with the exception that the GET method is indicated. This second example results in a read operation:

```
<!-- update operation -->
<form method="post" action="..." />
  <input name="keywords" type="text" value="" />
  <input type="submit" />
</form>

<!-- read operation -->
<form method="get" action="..." />
  <input name="keywords" type="text" value="" />
  <input type="submit" />
</form>
```

Link Annotation Controls (CL)

In addition to the ability to directly modify control data for read and submit operations, media types can define CL factors that provide inline metadata for the links themselves. Link control data allows client applications to locate and understand the meaning of selected link elements with the document. These CL factors provide a way for servers to "decorate" links with additional metadata using an agreed-upon set of keywords.

For example, Atom documentation identifies a list of registered Link Relation Values that clients may encounter within responses. Clients can use these link relation values as explanatory remarks on the meaning and possible uses of the provided link. In the example below, the Atom entry element has a link child element with a link relation attribute set to "edit" (`rel="edit"`):

```
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Atom-Powered Robots Run Amok</title>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
  <updated>2003-12-13T18:30:02Z</updated>
  <author><name>John Doe</name></author>
  <content>Some text.</content>
  <link rel="edit" href="http://example.org/edit/1"/>
</entry>
```

Clients that understand the Atom and AtomPub specifications know (based on the documentation) that the URI value of links decorated in this way can be used when executing idempotent submits (HTTP PUT, HTTP DELETE) to the server.

Another example of using CL factors is HTML's use of the `rel="stylesheet"` directive:

```
<link rel="stylesheet" href="..." />
```

In the above example, the client application (web browser) can use the URI supplied in the `href` attribute as the source of style rendering directives for the HTML document.

Summary

This section has identified a limited set of elements (H-Factors) that describe well-known protocol-related operations. These operations make up a complete set of hypermedia factors. These factors can be found, to some degree, in media types designed

to support hypermedia on the Web. H-Factors are the building blocks for designing your own hypermedia APIs.

Hypermedia Design Elements

Along with knowing the set of hypermedia factors that can be expressed within a message, there are a number of basic design elements that need to be addressed when authoring hypermedia types. These elements are:

Base Format
: Every hypermedia design relies on a base-level message format; it's the format that is used to express the hypermedia information. Typical base formats for hypermedia messages sent over HTTP are XML, JSON, and HTML. They each have their advantages and limitations, which will be covered in this section. It is also possible to design hypermedia types using other base formats (CSV, YAML, Markdown, Binary formats, etc.), but this book does not cover these additional base formats.

State Transfer
: Many hypermedia types allow client-initiated state transfer (sending data from the client to the server). In "Identifying Hypermedia : H-Factors" on page 13, several hypermedia factors were identified as supporting state transfer. Hypermedia designs typically have three styles of state transfer: None (i.e. read-only), Predefined (via external documentation), and Ad-Hoc (via in-message hypermedia controls).

Domain Style
: Hypermedia designs usually express some level of domain affinity. In this context, "domain" refers to the application domain, or the problem space. Domain styles can be categorized as Specific, General, or Agnostic.

Application Flow
: Hypermedia designs may also contain elements that express possible application flow options. This allows client applications to recognize and respond to possible options that allow for advancing the application through state transitions at the appropriate times. Application Flow styles for hypermedia can be identified as None, Intrinsic, or Applied.

These hypermedia design elements (minus the Base Format element) can be viewed as a matrix. (See Figure 1-2.)

All hypermedia types can be inspected for these design elements (Format, State Transfer, Domain Style, and Application Flow) and their values identified. In this way, you can analyze existing hypermedia types and use the same information in making selections for your own hypermedia type designs.

Hypermedia Design Elements			
State Transfer	Read-Only	Predefined	Ad-Hoc
Domain Style	Specific	General	Agnostic
Application Flow	None	Intrinsic	Applied

Figure 1-2. Hypermedia Design Matrix

Just as the previous section ("Identifying Hypermedia : H-Factors" on page 13) identified a set of factors or building blocks for hypermedia, this section describes a set of design elements or techniques for applying those factors to a hypermedia design.

The following sections explore each of these design elements, provide examples, and offer some guidance on their use in your hypermedia designs.

Base Format

A critical element in any hypermedia design is the base-level message format. As of this writing, the most often used formats over HTTP are XML, JSON, and HTML. This section explores the advantages and limitations of these common formats.

XML

The XML media type is a common base format for designing hypermedia. There are several advantages to using XML as your base format. First, XML is a mature format and there are a number of supporting technologies including transformation (XSLT), querying (XPath, XQuery), validation (XSD, Schematron), and even transclusion (XPointer, XInclude). XML data types are also standardized. Even better, almost all programming environments support these technologies in a consistent way. It's safe to bet that you can count on XPath to work the same way across platforms and languages.

Another nice aspect of XML is its element + attribute design. Designers can take advantage of this pattern in many ways, including using elements to define top-level data descriptors in the media type, and attributes as additional or metadata items.

One of the drawbacks of XML is that the original media type contains no native H-Factors: no predefined links, forms, etc. There are some related XML standards (XLink, XForms) that can be applied, but these may or may not be exactly what the use case requires and these additional standards may not be widely supported on all target platforms.

However, if your use cases require strong standardized support for your hypermedia type across a wide range of platforms and environments, XML can be an excellent choice as a base format.

JSON

The rise of the HTTP web as a platform has brought with it an increased use of JavaScript. Today it is possible to find JavaScript as the default programming language for clients (web browsers), servers (Node.js), data storage (CouchDB) and more. JavaScript's language model supports a very simple and portable data structure based on name-value pairs and lists called JSON. This data structure has been standardized, and parsers are available for a wide range of languages outside JavaScript, too. Another advantage of JSON is that it is a very terse format. For example, unlike XML, whose design can end up using more bytes for element names than for the data these elements describe, JSON has relatively low overhead and this can make messages very small and easy to move about.

While JSON is a standard, it is still relatively new. There are no RFC-level standards for querying and validating JSON although there are some commonly used approaches. For example, JSONPath is a query pattern similar to XPath and JSON Schema provides an XSD-like validator service to JSON. Finding implementations of JSONPath and JSON Schema may be difficult for environments where JavaScript is not the chosen programming language.

Another downside for using JSON is that, like XML, JSON has no native H-Factors and there are no established standards to rely upon for expressing links and forms. Designing a hypermedia type with JSON requires the definition of all the H-Factor elements from scratch, including figuring out the best way to express control data information such as language preferences and protocol method names, etc.

Despite these drawbacks, if your target audience is made up of web browsers and other environments where JavaScript is common, JSON may be a good fit as the base format for your hypermedia type design.

HTML

HTML can be an excellent choice as a base format for your hypermedia design for a number of reasons. First, it is a mature and stable technology that is more than twenty years old. Second, HTML has one of the most ubiquitous client applications (common web browsers) available on almost any platform. Third, browsers support code-on-demand via JavaScript, which adds a power dimension to delivering hypermedia via HTML.

However, HTML (and its cousins, XHTML and HTML5) is often overlooked when designing hypermedia types for a use case. This is probably because HTML suffers from an unfair assumption that it is an old-school, bloated technology appropriate only for cases where a human is driving a web browser client.

HTML does have some drawbacks. As of this writing, HTML still only supports a subset of the HTTP method set (GET, HEAD, and POST). If your use case relies on using HTTP PUT and/or DELETE, plain HTML will not be a good fit. One of the biggest downsides to using HTML as your base format is that it is domain-agnostic (see "Agnostic" on page 28). That means it can be a bit more work to define elements and attributes that match closely to your application domain, but it is certainly possible, as you will see in Chapter 4.

But these limitations are usually outweighed by the advantages of HTML. HTML is the only base format considered here that has a rich set of native hypermedia controls to support LO H-Factors (via links), and LT and LN H-Factors (via forms). Since web browsers can easily render HTML links and forms, defining your hypermedia API using HTML usually means that humans can easily surf your API* by stepping through the HTML as it is rendered in a browser. There are now a number of libraries capable of parsing HTML, XHTML, and HTML5 that are available for several platforms and environments. That means it is relatively easy to use HTML for use cases that do not require web browsers (e.g. command-line tools, desktop applications, etc.).

Since HTML can be used in wide number of client environments including web browsers, HTML can be a very good choice as a base format for your hypermedia designs.

Others

As mentioned at the start of the section, XML, JSON, and HTML are not the only possible base formats for hypermedia designs. Markdown, YAML, CSV, even binary formats (i.e. Protocol Buffers) can be used in hypermedia designs. However, many of these formats lack not only native hypermedia controls, but the document structure needed to define them. For example, XML offers element names and attributes to hold your application-specific metadata. JSON has hash tables and arrays that can be given meaningful names to match your application domain. HTML has a number of attributes especially designed to hold domain-specific information. Most of the alternate base formats mentioned here do not have these types of allowances. For the purposes of this book, attention will remain on the three most commonly used base formats today. If your use case calls for supporting hypermedia using one of these other formats, you may have some additional challenges, but you should still be able to apply the same design ideas shown here.

What about RDF?

One well-known format that is not covered in this book is RDF (Resource Description Framework). Technically, RDF is not format or media type. It is a data interchange standard that leverages the tuples pattern, relies heavily in URIs, and uses well-defined ontologies. While there have been a number of attempts to define hypermedia controls

* I first heard the term "surfing your API" from Jonathan Moore in his presentation at Oredev in 2010, where he demonstrated using browsers to explore hypermedia types based on XHTML.

for RDF (most notably RDF Forms), to date no ontology that supports a wide range of H-Factors has emerged as a clear leader for RDF. There is an XML serialization for RDF (RDF-XML) and a number of JSON variations, but none of those serializations of RDF include strong support for H-Factors either. For those who want to use the RDF interchange standard and still need hypermedia support, the RDFa specification for expressing structured data using attributes (when applied to HTML) is probably the best choice since it offers all the H-Factors of HTML (LO, LE, LT, LN) as well as the ability to model RDF's subject-predicate-object expressions.

State Transfer

Another key aspect of hypermedia design is supporting the transfer of information (i.e. state) from the client to the server. This client-initiated state transfer is really the heart of hypermedia messaging. There are many media type designs focused on efficient transfer of data from servers to clients, but not many do a good job of defining how clients can send data to servers. In some cases, read-only designs are appropriate for the task. For example, bots that search for and index specific data on the Web usually have no reason to send data to other servers. However, in cases where client applications are expected to collect inputs (e.g. from human users or by other means) and store them on a remote server; these same client applications will need to know how to locate and use hypermedia-enabled link controls (the LT, LI, and LN H-Factors).

For the purposes of this book, we can divide the work of expressing client-initiated state transfer for hypermedia types into three types: read-only, predefined, and ad-hoc.

Read-only

As was already mentioned, there are a number of scenarios where hypermedia types do not need to support client-initiated state transfers. In these cases, the media types are, essentially, read-only. That does not mean that the messages are devoid of hypermedia controls. For example, the SVG [SVG11] media type is a read-only design that uses outbound links (LO) and embedding links (LE). The CSS media type supports the LE H-Factor.

If your use case does not require clients to transfer data directly to servers, using a media type design that supports no client-initiated state transfer is a perfectly valid design choice.

Predefined

Another common approach to handling client-initiated state transfer is to rely on predefined transfer bodies that clients learn and then use when indicated. Media types that rely on this design pattern usually provide documentation detailing the required and/or optional elements for valid transfers, encoding rules, etc. The AtomPub (RFC5023)

protocol relies on predefined state transfers to support creating and updating resources via the Atom (RFC4287) media type.

The AtomPub/Atom RFC pair is an interesting example of one RFC defining the format (RFC4287) and another, related RFC defining the state-transfer rules (RFC5023).

One of the advantages of using predefined state transfers is that client-coding can be relatively straightforward. Client applications can be pre-loaded with the complete set of valid state transfer bodies along with rules regarding required elements, supported data types, message encoding, etc. The only remaining task is to teach clients to recognize state transfer opportunities with response messages, and to know which transfer body is appropriate.

In the case of the AtomPub protocol, there are two basic client-initiated state transfers defined in the specification:

Entry resources
: These transfers represent entries in an Atom feed document and can be treated as a stand-alone resource.

Media resources
: These transfers support binary uploads (images, etc.) to the server that are automatically associated with an Entry Resource.

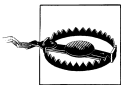
The details on how clients can recognize when state transfers are supported (e.g. identifying predefined `rel` attributes on `link` elements), how clients should compose valid state transfer requests (e.g. which protocol method to use, etc.), and how servers should respond to these requests are outlined in the AtomPub RFC.

In cases where your media type only needs to support a limited set of possible state transfers from the client, it can be a good design choice to define these state transfer bodies within documentation and encourage client applications to embed the rules for handling this limited set directly in the client code.

Ad-Hoc

A very familiar method for handling client-initiated state transfers is to use an ad-hoc design pattern. In this pattern, the details about what elements are valid for a particular state transfer are sent within the hypermedia message itself. Since each message can have one or more of these control sets, clients must not only know how to recognize the hypermedia controls but also how to interpret the rules for composing valid transfers as they appear. The HTML media type relies on this ad-hoc design pattern using the `form`, `input`, and other elements to support LT and LN link H-Factors.

The primary advantage of adopting the ad-hoc style is flexibility. Document authors are free to include any number of transfer elements (inputs) needed to fulfill the immediate requirements. This also means that client applications must be prepared to recognize and support the state transfer rules as they appear in each response.



For human-driven clients, ad-hoc state transfers can be handled by rendering the inputs and waiting for activation. However, for clients that have no human to intervene, the ad-hoc style can be an added challenge. If your primary use case is for automated client applications, the ad-hoc state transfer style may not be the best design choice.

The HTML documentation identifies the hypermedia elements (e.g. `form`, `input`, `select`, `textarea`, etc.) that client applications should support along with encoding rules on how to convert values associated with these controls into valid state transfer bodies. Once client applications know how to handle the designated elements, they will be prepared to handle a wide range of hypermedia messages.

If your use case requires the power and flexibility of ad-hoc state transfers, this is probably the best choice for your media type design.

Domain Style

Hypermedia designs usually express some level of domain affinity. In this context, "domain" refers to the application domain, or the problem space. The process of selecting element and attribute names, deciding where hierarchies should exist, and so on, is the essence of translating the problem domain into message form. Modeling the problem domain focuses on the information that needs to be shared, stored, computed, etc. There are many ways to accomplish this task, and the common approach is to model domain data by declaring meaningful elements and/or attributes. These elements are then passed between parties where they can be processed, computed, stored, and passed again when requested. Achieving a desirable domain model for messaging is the art of hypermedia design.

The closer a design is to modeling the problem space, the easier it is to use the design to accurately express important activities over a distributed network. However, the more closely tied a message design is to a single problem domain, the less likely it is that the design can be applied to a wide range of problems. This balance between specificity and generality is at the core of hypermedia design.

It can be helpful to view this issue (like others covered here) in three broad categories: specific, general, and agnostic.

Specific

It is very common to use a very domain-specific design when creating a custom message. Domain-specific designs usually incorporate name and collection patterns that exist

within the problem space itself. For example, consider a typical design for an Order in a message:

```
<!-- domain-specific design -->
<order>
  <id>...</id>
  <shipping-address>...</shipping-address>
  <billing-address>...</billing-address>
  ...
</order>
```

In the example shown above, it is very easy to identify the domain-specific elements (order, order-id, customer-name, shipping-address, etc.). The primary advantage of a domain-specific design is that it is easy for humans working with the design to determine the meaning and infer the use of the various elements. This is one the many reasons XML is a popular format for custom message implementations: XML does an excellent job of supporting domain-specific designs.

There are drawbacks to this style of message design, too. The more specific your design, the more closely tied it is to a single problem domain. If your problem domain is quite large, your message design becomes very large, too. If your domain space changes frequently over time, your message design must do the same. Finally, if your problem domain is rather small and very specific, it's not likely that your design can be applied to many other use cases.

If your domain space is well-established and stable (not likely to change over time) or if your use case is relatively short-lived, domain-specific style designs can be a good choice.

General

An alternate approach to domain-specific designs is to adopt a domain-general style. In this style, elements are given generally understood names. Optionally, elements are decorated with attributes that qualify the general name with something more domain-specific. In this way, general style designs strike a balance between specificity and generality.

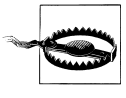
Here is one possible domain-general style design for the Order message shown earlier:

```
/* domain-general design */
{
  "order":
  {
    "id" : "...",
    "address" : {"type" : "shipping", "street-address" : "..."},
    "address" : {"type" : "billing",  "street-address" : "..."}
  }
}
```

You can see that the domain-specific address elements from the first example have been replaced by general "address" elements carry a "type" indicator to provide additional

domain-specific information. The advantage of this design style is that the "address" elements could be applied to many other use cases within the problem domain (customer address, user address, supplier address, etc.) without having to modify the actual message design itself. In this way, your design takes on a level of modularity and reuse that makes supporting new domain-specific elements easier over time. In addition, client applications can create code that: 1) supports reusing these modular elements of your message design, and 2) is able to adjust to evolving use cases in the domain space more easily.

There are still downsides to this approach. First, creating domain-general designs adds a level of indirection to implementations. It usually takes more coding logic to parse both the element ("address") and the domain-specific indicator ("type"). This can make finding the right address element in a particular message a bit more complicated. On the other hand, domain-general messages can still suffer from enough specificity to limit their wide use and adoption.



Care should be taken when employing a domain-general style since it is possible to design a message that takes on the limitation and frustrations of both domain-specific and domain-agnostic styles.

In cases where your domain has a core set of reusable elements and relatively simple messages that are not likely to make for complicated parsing due to a high level of reuse in the same message, a domain-general approach can be the best selection.

Agnostic

The most flexible and evolveable domain style is domain-agnostic. In this style, all the element names are generic (i.e. "data" or "item," etc.) and there is a strong reliance on context-setting values (usually attributes) to establish the meaning of these generic elements. The following example of a domain-agnostic design should look familiar to the reader:

```
<!-- domain-agnostic design -->
<ul class="order">
  <li class="id">...</li>
  <ul class="shipping-address">
    <li class="street-address">...</li>
    ...
  </ul>
  <ul class="billing-address">
    <li class="street-address">...</li>
    ...
  </ul>
  ...
</ul>
```

The primary advantage of the domain-agnostic style is that you can usually express a wide range of domain-specific information using every few elements. A domain-ag-

nostic message design can usually get by with just a few key elements used to express collections, individual items in a collection, and one or more properties of a single item. The trick to designing domain-agnostic messages is to employ a rich set of decorators (attributes) that can be applied to almost any element in the design. HTML, for example, supports the `id` and `class` attributes on almost all elements. HTML also uses the `name` and `rel` attributes on key state transition elements.

An important aspect of HTML design is that some decorators support only a single value (`id="this-element"`) and others support multiple values (`class="important order pending"`). This distinction in decorators can be very handy when designing domain-agnostic hypermedia types.

Of course, the domain-agnostic style has its limitations. Chief among these is that overly generic markup can pose a challenge to clients attempting to parse the message. This includes humans, too. We naturally desire clear statements of meaning in messages and using an agnostic approach that relies on one or more levels of indirection via attribute decorators can be confusing and frustrating at times. There is also the possibility that a design can be too agnostic, where there is almost no meaning in the elements themselves and all interesting information is tucked out of the way in decorators. This can lead to overly complex coding for applications creating the messages as well as those receiving, parsing, and rendering the results.

Ultimately, domain-agnostic messages require a slightly different approach to hypermedia design; one that separates the semantics of the data with the message markup itself. This notion of a dual-level design (the message markup and the domain-specific design) will be explored more fully in Chapter 4.

For use cases where a single message design needs to be used across a wide range of problem domains and/or where the problem domain is likely to change and evolve over time, a domain-agnostic style can be very effective.

Application Flow

Hypermedia designs may also contain elements that express possible application flow options. This allows client applications to recognize and respond to possible options that allow for advancing the application through state transitions at the appropriate times. Designing support for application flow in hypermedia types is more than just including links and forms in responses. Application flows require identifications for the various possible options for changing the state of the application.

In many existing media types, this is done using decorators on links (`rel="edit"` in AtomPub) and forms (`name="payment"` in HTML). However, in VoiceXML, a domain-specific design for telephony services, there are a number of application flow elements such as `goto`, `exit`, `return`, `help`, `log`, and others. Other logical applications flow iden-

tifiers could be “write”, “update”, “remove”, “add”, “save”, and a whole host of nouns and activities that are domain-specific (e.g. payment, order, customer, etc.).

Some media types define application flow identifiers as native to the media type (e.g. HTML’s link relation values). Other designs rely on a public registry of values such as the IANA Link Relation Registry, Microformats Existing Rel Values, and Dublin Core Terms.

Whether the application flow is identified using unique elements, attributes, or unique values for existing attributes, application flow styles for hypermedia can be categorized into three general groups:

None
: This hypermedia type contains no identifiers for application flow. Designs that are read-only and/or cover a very limited domain may not support any application flow identifiers.

Intrinsic
: The application flow identifiers are defined within the hypermedia type itself. Atom and AtomPub define a small set of link relations to indicate application flow (“edit”, “edit-media”, and “self”).

Applied
: The application flow identifiers are not part of the hypermedia type, but the type designs contain allowances (usually element decorators or attributes) for applying external values to indicate application flow. HTML supports a number of attributes that can be used to add application flow information including the `profile`, `id`, `name`, `rel`, and `class` attributes.

None

There are several cases where your hypermedia design does not require any application flow identifiers. Usually this is when the problem space covered is rather limited and/or read-only in nature. Hypermedia type designs for automated services (bots, machine-to-machine interactions, etc.) are a good candidate for designs that do not contain application flow identifiers.

For example, the text/uri-list media type is designed to hold a list of URIs only. It is a read-only media type that supports the LO H-Factor and can be used for presenting lists of URIs to bots and other automated services. There is no need for application flow since the only work is to resolve the list of presented URIs.

```
# urn:isbn:0-201-08372-8
http://www.huh.org/books/foo.html
http://www.huh.org/books/foo.pdf
ftp://ftp.foo.org/books/foo.txt
```

Another hypermedia type that has no need for application flow is the OpenSearch specification. Designed to support searching web indexes, the OpenSearch specification uses the XML base format and supports only the LO H-Factor in responses.

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
  <ShortName>Web Search</ShortName>
  <Description>Use Example.com to search the Web.</Description>
  <Tags>example web</Tags>
  <Contact>admin@example.com</Contact>
  <Url type="application/rss+xml"
       template="http://example.com/?
q={searchTerms}&amp;pw={startPage?}&amp;format=rss"/>
</OpenSearchDescription>
```

If your use case covers a relatively narrow problem domain and/or your hypermedia will be used primarily by automated systems, a design that has no application flow can be a good option.

Intrinsic

If you want to design a hypermedia type that supports application flow, one way to do this is to define the application flow identifiers as part of your media type directly. This can be done in two primary ways: 1) by identifying specific elements or attributes in your design that represent options for application flow (e.g. `<update>...</update>` or `<store type="update">...</store>`); and 2) by identifying unique decorator values that can be applied to existing elements and attributes (e.g. `<link rel="update" ... />`). In these examples, the rules for application flow are intrinsic to the media type definition itself.

The AtomPub media type uses this intrinsic style of application flow. The specification identifies a few elements and link relations that clients can use to activate application flow. For example, the AtomPub spec identifies the `href` attribute of the `app:collection` element as the URI to use to create new entries:

```
<?xml version="1.0" encoding='utf-8'?>
<service xmlns="http://www.w3.org/2007/app"
         xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title>Main Site</atom:title>
    <collection
        href="http://example.org/blog/main" >
      <atom:title>My Blog Entries</atom:title>
      <categories
         href="http://example.com/cats/forMain.cats" />
    </collection>
  </workspace>
  ...
</service>
```

It also states that the `href` attribute of the `atom:link` element marked with `rel="edit"` contains the URI for use when updating or deleting individual entries:

```
<?xml version="1.0"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>Atom-Powered Robots Run Amok</title>
  <id>urn:uuid:1225c695-cfb8-4ebb-aaaa-80da344efa6a</id>
```

```
  <updated>2003-12-13T18:30:02Z</updated>
  <author><name>John Doe</name></author>
  <content>Some text.</content>
  <link rel="edit" href="http://example.org/edit/first-post.atom"/>
</entry>
```

Intrinsic application flow works well when you want your design to stand alone and not depend on any external specifications or definitions, and when your application flow options can be expressed with a limited set of elements, attributes, and/or values. This works well for hypermedia types that cover a relatively limited set of use cases where the options are not likely to change over time, or types that provide support for a general use case (e.g. writing blog entries, etc.). Intrinsic application flow is also very helpful if you are working to define a hypermedia type that might be implemented by a wide range of servers all attempting to support the same use cases.

Applied

When your hypermedia design needs to support application flow that can change over time, or has a wide range of possible use cases, you may need to consider a design that relies on external identifiers that can be applied consistently to specific elements and attributes of your hypermedia type.

The HTML media type uses this approach to support application flow. There are a handful of HTML attributes that can be used to define application flow options including `rel`, `class`, `name`, and `id`. By supplying predefined values to these attributes, designers can apply details about application flow options that may appear in a message.

The key to making the applied style work is to publish a set of predefined values along with their meaning and purpose at a stable URI. This external application flow specification can then be accessed and used by client and server implementors to guide the creation, parsing, and interpretation of hypermedia messages using the specified rules. A pointer to this specification can be shared as a link header and/or as a part of the entity body. This can be done in HTML using the `profile` attribute or the `meta` element.

In the following HTML example, the response indicates the application flow specification in use (see the `meta` tag). You can see several places in the document where elements are decorated with `rel`, and `name` attributes indicating application flow options:

```
<html>
  <head>
    <title>Payment Options</title>
    <meta name="profile" content="http://www.example.com/profiles/payment.html" />
  </head>
  <body>
    <h1>Payment Options</h1>
    <p>
      <a href="..." rel="cancel-order">Cancel Order</a>
    </p>
```

```
    <form href="..." name="credit-card" method="post">
      <input name="card-number" value="" />
      ...
      <input type="submit" />
    </form>
    <form href="..." name="purchase-order" method="post">
      <input name="po-number" value="" />
      ...
      <input type="submit" />
    </form>
    <form href="..." name="bank-draft" method="post">
      <input name="routing-number" value="" />
      ...
      <input type="submit" />
    </form>
    ...
</html>
```

The applied style of application flow offers the most flexibility since it allows designers an almost unlimited number of possibilities. This also allows designers to define application flow specifications that are independent of a particular media type; the rules can be applied to any base format (XML, JSON, etc.) or existing hypermedia type that supports it (similar to the way CSS and XSLT work for HTML today).

There are also a number of drawbacks to the applied style. First, by creating an external document that holds the application flow details, clients and servers will be required to support not only the primary media type but also a second rule document. This can put an added burden on client and server implementors. Second, clients and servers may ignore the rule specifications completely, which can cause problems, especially for server implementors who may have to deal with clients who are not following the rules laid out in the application profile. Additionally, when the profile document is updated over time with new application flow options, there is no guarantee that clients using that profile will recognize and honor the changes without additional coding updates. There are ways to mitigate these problems, but there are no standards for doing so.

Despite these drawbacks, in cases where your design must support a wide-ranging, flexible set of applications flow options that can change over time, the applied style may be your best option.

Summary

In this chapter, four key topics were discussed: 1) the underlying technologies behind the Web, 2) the importance of adopting hypermedia as the basis for sharing data on the Web, 3) the identification of nine H-Factors used in all hypermedia designs, and 4) four basic design elements used in implementing a functioning hypermedia type.

Today the Web relies on HTTP as the transfer protocol for sending messages. These messages are used to represent data using MIME media types. The representation

model for the Web allows clients and servers to negotiate for preferred data formats including support for future extensibility through the design of new data formats. The development of the Web follows the introduction of hypermedia links as a method of not only navigating between documents but also through the use of hypermedia controls that support sending parameterized queries and write instructions to remote servers.

The essence of programming the Web is designing hypermedia-rich messages that can be understood and passed between parties on the network. Unlike object-serialization patterns that simply convert internal private data structures into bytes that can be passed between client and server, the Web encourages the use of coarse-grained messages that include metadata that describes not only the data being passed but also the state of the application at time of the request. The set of hypermedia elements that can be used to communicate application state changes (H-Factors) is the same regardless of the data format used to transfer the message. The design of these hypermedia messages depends on four key elements including data format, state transfer style, application domain style, and flow control.

What's Next?

The following three chapters illustrate the process of designing and implementing hypermedia APIs. Each chapter is devoted to a particular application domain and shows how choices are made in the process of the hypermedia design (which H-Factors are needed, what data format is selected, etc.) in order to achieve the desired outcome. The hypermedia designs that follow are all based upon the concepts and techniques described in this chapter.

Let's design some hypermedia APIs!

CHAPTER 2
XML Hypermedia

They that mistake life's accessories for life itself are like them that go too fast in a maze: their very haste confuses them.

- Seneca

In this first hands-on design chapter, a simple read-only, XML-based media type will be used to show how to convert a use case scenario into a successful design. Once the design is completed, example server and client implementations will be explored including one client example that shows how an autonomous web bot can successfully choose from the available hypermedia links in order to reach a final goal.

Scenario

For the first project, what is needed is a web-enabled "Maze Game." In this game, clients should be able to request a maze to play and then be able to navigate from a predetermined starting point through the maze to the exit. Clients should be able to use the cardinal directions (north, south, east, and west) to move through a two-dimensional space. Mazes could be of any size or shape; use any configuration of rooms and doorways.

What is needed is a media type that can express the state of a given client's traversal through the maze from start to finish. A minimum set of requirements for this media type design should allow client applications (given an initial starting URI) to:

- Discover a list of available mazes
- Select one of the available mazes to play
- Recognize the existing doorways in each cell of the maze
- Navigate through a selected doorway into the next cell
- Recognize and navigate through the exit when it appears

Note that the requirements are expressed in terms of what *clients* can do. Hypermedia designs almost always start with a focus on what clients will be doing. One of the reasons for this is that in client-server transfer protocols like HTTP, all requests are initiated by the client. Client-centered design also reinforces the notion that the representation (message) received by the client must contain the next possible steps available. For example, in the current project (the Maze Game), each response from the server must include indications of where the doors are in the current room. The doors represent the next possible steps for the client application.

For the sake of the exercise, the server will generate what are called perfect mazes. These are mazes that have rooms of equal sizes (four equal-length walls) with doors possible on one or more sides. The mazes used in this example will always have one entrance and one exit, too. However, it should be noted that these server limitations are immaterial to the media type design used to express mazes and that client's current location within a given maze. These restrictions are only to reduce the complexity of the code used on the server to generate the maze, and to make it easier to write client code that can solve the maze.

With these basic requirements in mind, a hypermedia design can be created.

Designing the Maze XML Media Type

The process of coming up with the actual design of a hypermedia type is just as much art as science; there is no single way to approach the problem. However, the methodology outlined in this book is based on the notion that server responses will always represent the state of the application from the client's point of view. In other words, each server response is a state representation. With this in mind, designers can start by identifying a state of the application and crafting a representation of that state using the elements, attributes, properties, values, etc. appropriate to the selected base format (XML, JSON, HTML, etc.).

In this Maze Game example, the possible states of the application are relatively easy to identify and model. In other, more extensive designs, it may be difficult to model all of the possible states ahead of time. The reader will see other ways to approach the hypermedia design process in later chapters.

Identifying the State Transitions

For this project, there are only a few states of the application that need to be represented:

- A list of available mazes
- A single maze (selected from the previous list)
- A single room in the maze (including the available doorways and/or the entrance or exit)
- The details of an error, if encountered along the way

You'll notice that the list of states includes error details. All hypermedia designs should include the ability to render details about a particular error. While HTTP supports returning response codes to indicate possible error states (4xx and 5xx codes), these are protocol-level indicators and are usually inadequate for expressing the details of the application-level error itself or any possible suggestions for remediation of the error, retry options, etc.

Along with identifying the possible states for the application, hypermedia designs rely on information for state transitions. Below is a modified list of the application states along with the appropriate transitions for each identified state.

Collection State
: In this state, the response represents a list of mazes. The possible transitions are: 1) select a maze (*Item State*), or 2) reload the list (*Collection State*).

Item State
: In this state, the response represents details on a single maze. The possible transitions are: 1) start into the maze, 2) return to the maze list (*Collection State*), or 3) reload the maze (*Item State*).

Cell State
: In this state, the response represents the details of a single room or cell in the maze. The possible transitions are: 1) continue through one of the doorways to the next room (*Cell State*), 2) return to the maze detail (*Item State*), 3) return to the maze list (*Collection State*), or 4) reload the cell (*Cell State*).

Error State
: In this state, the response represents the details on an error condition. For this design there are no transitions supported.

Using the set of application states and possible valid transitions for each is a very good way to approach hypermedia design. This is especially true when using a Domain-Specific style of design.

Selecting the Basic Design Elements

One of the first things to do when creating a media type design is to review each of the four basic design elements outlined in Chapter 1 (see "Hypermedia Design Elements" on page 20):

- Base Format
- State Transfer
- Domain Style
- Application Flow

By settling these design questions early, the remaining process will go more smoothly.

For this project, the base format selected will be XML. XML is a widely supported format on almost any platform and programming environment. That means we can expect a wide range of client applications to be able to parse and process our Maze Game design. In truth, almost any format will work. For example, there are libraries for the JSON format in a wide range of programming languages, too. However, for this project XML will be used anyway.

Since the requirements listed for this design do not include clients sending any data to the server (i.e. no opportunities to write data), this design can adopt the Read-Only state transfer style. By implementing this as a read-only design, client applications will not need to understand how to parse parametrized queries or input form elements. That means the media type need only support LO (Outbound Links) and possibly LE (Embed Links), but does not need to support LT, LN, or LI H-Factors (see "Identifying Hypermedia : H-Factors" on page 13). This will simplify both the design and implementation details, especially for the client.

The Domain Style for this design will be Specific. That means the design will contain elements and attributes that apply directly to the problem domain: mazes. For example, the root element in the XML document could be a `<maze/>` element, and each room could be a `<cell/>`, etc. This will limit the applicability of this hypermedia design to other problem domains (it will not make sense to try to use this design to handle card games, for example), but it will also present a rather logical document model to anyone attempting to render mazes for clients or servers.

Finally, there is the matter of Application Flow. For this design, the application flow style will be Intrinsic, one that is built right into the media type itself. The design requires only a small set of application flow options (get a list of mazes, select a maze, start in the first room, navigate to the next room, pass through the exit). Like the Domain Style decision, this design will use elements and attributes that communicate the possible actions in domain-specific language (north, south, east, west, start, exit, etc.).

With these design elements out of the way, it is time to work up the actual XML document to represent mazes.

The Maze+XML Document

Designing the actual XML document to represent mazes requires deciding on the exact elements and attributes needed to support the requirements of the domain space. XML Elements names will be used to represent the various states of the application. These XML elements will usually contain child elements to express the details of that particular state, too. Below are the high-level elements in the design:

`<maze />`
: This element is the root element and will appear in every response, even the error responses.

`<collection />`
: This element will represent the list of mazes available to the client. It will have one or more child elements, each representing a selectable maze.

`<item />`
: This element will represent a single maze, usually one selected by the client application. It will have at least one child element that represents the starting point of the maze.

`<cell />`
: This element will represent a single cell or room within the maze, usually the cell to which the client has navigated on its traverse through the maze. It will have one or more child elements, each representing a possible doorway through which the client can pass.

`<error />`
: This element will represent the details of any error that has been encountered (e.g. the client has attempted to pass through a wall, the server is unable to accept any more requests, etc.). It will contain one or more elements to communicate various details including possible ways to fix the problem and repeat the request.

With these high-level elements defined, it is possible to create a basic map of the hypermedia design at this point:

```
<maze>
  <collection />
  <item />
  <cell />
  <error />
</maze>
```

The next step is to complete the details for each of the top-level elements in the hypermedia design. These details include data elements and attributes as well as H-Factors (links) to express the state transitions.

The design walk-through here covers only the highlights. A complete documentation of the Maze+XML media type can be found at the back of this book (See Appendix C).

The `collection` element

The `collection` element will be used to represent the *Collection State* of the application. This state supports a transition to a single maze representation (*Item State*) or to simply reload the list. This design will represent both transitions using LO H-Factors.

For the *Item State* transitions, this design will use a `<link/>` element with `href` and `rel` attributes to hold the URI and transition identifier respectively:

```
<link href="..." rel="maze" />
```

The transition that supports reloading the list is actually a transition to the *Collection State*. To support that, the design will include an `href` attribute on the `<collection/>` element itself:

```
<collection href="...>...</collection>
```

With these added features, it is possible to fully represent that *Collection State*:

```
<maze>
  <collection href="...">
    <link href="..." rel="maze" />
    <link href="..." rel="maze" />
    ...
  </collection>
</maze>
```

Required, Optional, and Hypermedia Design

You may have noticed that, to this point, no elements or attributes have been indicated as required or optional. In general, it is good practice to allow all design elements to be treated as optional for representations. This provides for possible future changes in the design without breaking existing clients.

There is a section in Chapter 5 (see "The RFC 2119 Keywords" on page 136) that covers additional details on how to use a special set of standardized keywords (from RFC 2119) to indicate the status of design elements.

The `item` element

The *Item State* of the application will be represented using the `item` element. This state can be used to supply details about a single maze. For this project, the only detail will be a link that points to the entrance of the maze. Other details that might be interesting could be the name of the maze, its dimensions, etc.

The *Item State* supports starting into the maze (*Item State*), returning to the maze list (*Collection State*), or reloading the maze details (*Item State*). Again, LO H-Factors will be used to indicate these transitions. Using the previous description for the *Collection State* as a guide, the following shows the design of the `item` element:

```
<maze>
  <item href="..." >
    <link href="..." rel="collection">
    <link href="..." rel="start" />
  </item>
</maze>
```

Note the placement of the `href` attribute on the `<item/>` element to support the *Item State* transition (reloading the maze details). Also note the `<link/>` element with the `rel="collection"` attribute to support the *Collection State* transition as well as the `<link/>` element with the `rel="start"` attribute to support the first transition to the `Cell State`.

The `cell` element

Representation of the *Cell State* is the task of the `cell` element in this design. All possible transitions such as moving through doorways (*Cell State*), returning to the maze details (*Item State*), returning to the maze list (*Collection State*), or reloading the cell (*Cell State*) will all be represented using LO H-Factors.

Below is a map of all the possible transitions for this element:

```
<maze>
  <cell href="...">
    <link href="..." rel="north" />
    <link href="..." rel="south" />
    <link href="..." rel="east" />
    <link href="..." rel="west" />
    <link href="..." rel="exit" />
    <link href="..." rel="maze" />
    <link href="..." rel="collection" />
  </cell>
</maze>
```

It is not likely that all the transitions listed above would be valid at the same time. Some cell walls will not have doors and only one cell is likely to be the exit. However, the example `<cell/>` element above shows all the possible transitions that clients must be ready to handle.

You may have noticed that none of the design examples here contain values for the `href` attribute. In fact, *none of the hypermedia designs in this book will contain static URIs or suggested resource names.* Good hypermedia design focuses on properly representing application states and possible transitions that should be available to the client. This allows each server implementation to determine their own URI scheme, resource model, and any other details needed to support an application that uses the target hypermedia type. The rationale behind this constraint on hypermedia designs was outlined in 2008 by Roy Fielding in his blog post "REST APIs must be hypertext driven" (see "Roy Fielding's Hypertext API Guidelines" on page 163).

The `error` element

Finally, the `error` element will be used to represent the *Error State* of the application. For this design, this element will contain only the data elements `<title/>`, `<code/>`, and `<message/>`. These elements can be used by the server to pass any additional information that is appropriate:

```
<maze>
  <error>
    <title>...</title>
    <code>...</code>
    <message>...</message>
```

```
  </error>
</maze>
```

Sample Data

In order to validate the hypermedia design, you need a running server implementation. Before you can implement the server, you need valid sample data. For this example, the maze data is stored in CouchDB as a simple JSON object composed of 25 cells. Each cell in the maze represents a room and each room as four walls, some of which have doors. The walls with doors are represented by "0"s and the other walls are represented by "1"s. Below is the 5x5 maze used for validating this hypermedia design.

Below is the maze document stored in CouchDB that will be used for testing:

```
{
  "_id" : "five-by-five",
  "cells" : {
    "cell0":[1,1,1,0],
    "cell1":[1,1,1,0],
    "cell2":[1,1,0,0],
    "cell3":[0,1,0,1],
    "cell4":[0,1,1,0],
    "cell5":[1,0,1,0],
    "cell6":[1,0,0,1],
    "cell7":[0,0,1,0],
    "cell8":[1,1,0,0],
    "cell9":[0,0,1,1],
    "cell10":[1,0,0,1],
    "cell11":[0,1,1,0],
    "cell12":[1,0,1,1],
    "cell13":[1,0,0,1],
    "cell14":[0,1,1,0],
    "cell15":[1,1,1,0],
    "cell16":[1,0,1,0],
    "cell17":[1,1,0,0],
    "cell18":[0,1,0,1],
    "cell19":[0,0,1,0],
    "cell20":[1,0,0,1],
    "cell21":[0,0,0,1],
    "cell22":[0,0,1,1],
    "cell23":[1,1,0,1],
    "cell24":[0,0,1,1]
  }
}
```

See "Source Code" on page 217 for details on how to download the book's source code including the script for creating this CouchDB database.

The Server Code

With the hypermedia design sketched out and sample data available, it's time to implement a server that emits responses for our new media type. This step makes it possible to validate the hypermedia design and make any adjustments that might be needed.

For the examples in this book, the servers will be implemented using the Express framework for Node.js. See "Source Code" on page 217 for details on how to obtain the full source code for this book. For brevity, and to focus on key portions of the implementation, only selected portions of the code appear in this section of the chapter. In this case, the server implementation has four code blocks, each corresponding to the four top-level elements of the design (see "The Maze+XML Document" on page 38). The code blocks represent the four possible states of the maze game. Along with the code block, the Express framework uses an associated EJS template to render the actual Maze+XML representation.

The Collection State Response

The Collection State lists the available mazes the server can share with the client application. Below is the code block along with the rendering template:

```
// handle collection
app.get('/maze/', function(req, res){
  res.header('content-type',contentType);
  res.render('collection', {
    title : 'Maze+XML Hypermedia Example',
    site  : 'http://localhost:3000/maze',
  });
});
```

```
<collection href="<%=site%>/maze/">
  <link rel="maze" href="<%=site%>/maze/five-by-five/" />
</collection>
```

For this example, just one maze is returned for the list. Here is an example runtime representation of the collection state:

```
<maze version="1.0">
  <collection href="http://localhost:3000/maze/maze/">
    <link rel="maze" href="http://localhost:3000/maze/maze/five-by-five/"/>
  </collection>
</maze>
```

The Item State Response

The Item State represents details on a single (selected) maze. Below is the Node.js code and EJS template for representing a single maze:

```
// handle item
app.get('/maze/:m', function (req, res) {
```

```
  var mz;

  mz = (req.params.m || 'none');

  db.get(mz, function (err, doc) {
    res.header('content-type',contentType);
    res.render('item', {
      site  : 'http://localhost:3000/maze',
      maze  : mz,
      debug : doc
    });
  });
});
```

```
<item href="<%=site%>/<%=maze%>/">
  <link rel="start" href="<%=site%>/<%=maze%>/0:north" />
  <link rel="collection" href="<%=site%>/" />
  <debug>
    <%=debug%>
  </debug>
</item>
```

Note that the Item State returns three possible links: 1) the Item's URI, 2) the start URI, and the collection URI. At runtime, a valid Item response looks like this:

```
<maze version="1.0">
  <item href="http://localhost:3000/maze/five-by-five/">
    <link rel="start" href="http://localhost:3000/maze/five-by-five/0:north"/>
    <link rel="collection" href="http://localhost:3000/maze/"/>
  </item>
</maze>
```

The Cell State Response

The Cell State represents any of the standard cells within the maze. Below is Node.js code and the EJS template for rendering standard cells:

```
// handle cell
app.get('/maze/:m/:c', function (req, res) {
  var mz, cz, x, ex, i, tot, sq;

  mz = (req.params.m || 'none');
  cz = (req.params.c || '0');

  db.get(mz, function (err, doc) {
    i = parseInt(cz.split(':')[0], 10);
    x = 'cell' + i;

    tot = Object.keys(doc.cells).length;
    ex = (i === tot-1 ? '1' : '0');
    sq = Math.sqrt(tot);

    res.header('content-type', contentType);
    res.render('cell', {
      site  : 'http://localhost:3000/maze',
```

```
        maze  : mz,
        cell  : cz,
        total : tot,
        side  : sq,
        ix    : [i-1, i + (sq*-1), i+1, i+sq],
        debug : doc.cells[x],
        exit  : ex
      });
    });
  });
```

This code is where the only computation for the server occurs. The line that computes the next possible moves is the array that initializes the `ix` template variable. Below is the template to match. You can see that this template uses `<% if %>` macros to see whether a doorway should be rendered in the response. In this way, only the valid next step links are presented to the client application:

```
<cell href="<%=site%>/<%=maze%>/<%=cell%>" rel="current">
  <% if(debug[0]=='0') { %>
    <link href="<%=site%>/<%=maze%>/<%=ix[0]%>:north" rel="north"/>
  <% } %>
  <% if(debug[1]=='0') { %>
    <link href="<%=site%>/<%=maze%>/<%=ix[1]%>:west" rel="west"/>
  <% } %>
  <% if(debug[2]=='0') { %>
    <link href="<%=site%>/<%=maze%>/<%=ix[2]%>:south" rel="south"/>
  <% } %>
  <% if(debug[3]=='0') { %>
    <link href="<%=site%>/<%=maze%>/<%=ix[3]%>:east" rel="east"/>
  <% } %>
  <% if(exit=='1') { %>
    <link href="<%=site%>/<%=maze%>/999" rel="exit" />
  <% } %>
  <link href="<%=site%>/" rel="collection" />
  <link href="<%=site%>/<%=maze%>/" rel="maze" />
</cell>
```

Here is an example of runtime representation of a cell in the maze:

```
<maze version="1.0">
  <cell href="http://localhost:3000/maze/five-by-five/5:east" rel="current">
    <link href="http://localhost:3000/maze/five-by-five/0:west" rel="west"/>
    <link href="http://localhost:3000/maze/five-by-five/10:east" rel="east"/>
    <link href="http://localhost:3000/maze/" rel="collection"/>
    <link href="http://localhost:3000/maze/five-by-five/" rel="maze"/>
  </cell>
</maze>
```

The Exit State Response

Although it is a `cell` representation, the Exit State response is treated as a special case in this server implementation. The exit cell has different links (including one to allow the client application to start the same maze over again). For this reason, there are separate code and template blocks for the exit state:

```
// handle exit
app.get('/maze/:m/999', function (req, res) {
  var mz, cz;

  mz = (req.params.m || 'none');
  cz = (req.params.c || '0');

  res.header('content-type', contentType);
  res.render('exit', {
    site  : 'http://localhost:3000/maze',
    maze  : mz,
    cell  : cz,
    total : 0,
    side  : 0,
    debug : '999',
    exit  : '0'
  });
});
```

```
<cell href="<%=site%>/<%=maze%>/<%=cell%>" rel="exit">
  <link href="<%=site%>/<%=maze%>/0:north" rel="start" />
  <link href="<%=site%>/<%=maze%>/" rel="maze" />
  <link href="<%=site%>/" rel="collection" />
</cell>
```

And a valid runtime response that renders the exit state looks like this:

```
<maze version="1.0">
  <cell href="http://localhost:3000/maze/five-by-five/0" rel="exit">
    <link href="http://localhost:3000/maze/five-by-five/0:north" rel="start"/>
    <link href="http://localhost:3000/maze/five-by-five/" rel="maze"/>
    <link href="http://localhost:3000/maze/" rel="collection"/>
  </cell>
</maze>
```

The Client Code

Now that the server implementation is completed, it's time to build some hypermedia client applications that understand the Maze media type. For this example, two clients will be developed: 1) a simple user interface modeled on classic adventure games of the 1990s, and 2) a web bot that automatically navigates and completes the maze without any human intervention.

Both client applications will be implemented using HTML5, CSS, and JavaScript.

Maze Game Example

This example interface will prompt the user to navigate the maze, step-by-step, and record their moves along the way. Attempts to make illegal or unknown moves will result in a short message of confusion from the application. When the user has reached the exit, the application will send out a congratulatory greeting.

Figure 2-1. Maze Game Screenshot

A screenshot of the Maze Game in action can be seen in Figure 2-1.

HTML5 Markup

The markup for this example is rather simple. It consists of a handful of `div` tags, a few `span` elements, and one `form`. All the details of populating the UI via the server responses are handled by the JavaScript. Below is the complete HTML5 markup for this example (`maze-game.html`):

```
<!DOCTYPE html>
<html>
  <head>
    <title>Maze Game</title>
    <meta http-equiv="Content-type" content="text/html;charset=UTF-8">
    <link rel="stylesheet" href="/stylesheets/maze-game.css">
    <script type="text/javascript" src="/javascripts/ajax.js"></script>
    <script type="text/javascript" src="/javascripts/mozxpath.js"></script>
    <script type="text/javascript" src="/javascripts/maze-game.js"></script>
  </head>
  <body>
    <h1>Maze Game</h1>
    <div id="play-space">
      <div id="history"></div>
      <div id="display">
        <span class="prompt">You have the following options:</span>
        <span class="options"></span>
      </div>
      <form name="interface" action="#" method="post">
        <fieldset>
```

```
          <legend>What would you like to do?</legend>
          <input type="text" name="move" value="">
          <input type="submit" value="Go">
        </fieldset>
      </form>
    </div>
  </body>
</html>
```

JavaScript

The client-side scripting for this example needs to handle only a few details including:

- Initializing the UI and loading the first Maze+XML response
- Displaying the possible options to the user based on the hypermedia links sent by the server
- Accepting (and validating) the user's next move
- Processing the valid move request and retrieving the next set of options from the server
- Repeat these steps until the user reachers the exit (or gives up trying)

When the page first loads, the application handles some initialization work (registering the event sink), sends a request for a maze representation to the server, and processes the response. Below is the relevant JavaScript code:

```
var g = {};
g.moves = 0;
g.links = [];
g.mediaType = "application/vnd.amundsen.maze+xml";
g.startLink = "http://localhost:3000/maze/five-by-five/";
g.sorryMsg = 'Sorry, I don\'t understand what you want to do.';
g.successMsg = 'Congratulations! you\'ve made it out of the maze!';

function init() {
  attachEvents();
  getDocument(g.startLink);
  setFocus();
}

function attachEvents() {
  var elm;

  elm = document.getElementsByName('interface')[0];
  if(elm) {
    elm.onsubmit = function(){return move();};
  }
}

function getDocument(url) {
  ajax.httpGet(url,null,processLinks,true,'processLinks',{'accept':g.mediaType});
}
```

```
function setFocus() {
  var elm;

  elm = document.getElementsByName('move')[0];
  if(elm) {
    elm.value = '';
    elm.focus();
  }
}
```

You should notice that the script for this example contains only one hardcoded URI: the one needed to start the game. All the other links come directly from the server responses. There is no code in the client to construct new URIs or call pages directly. This is one important aspect of writing hypermedia clients: limit the number of URIs coded directly into the application to, if at all possible, just one.

The details of processing the response from the client and displaying the options is the next step. In this example, the application finds all the `link` elements in the response and displays the `rel` values from each of those links to the user. These `rel` values represent the list of valid commands available to the user as possible next steps in the game.

The `rel` Attribute and Space-Separated Values

The HTML 4.01 spec defines the value of the `rel` attributes as "a space-separated list of link types." That means `rel="very important invoice"` is a syntactically correct `rel` attribute. You'll notice the client-side code in this example correctly parses the `rel` value as a collection of space-separated values when it populates the link collection for each server response. This is an example of good hypermedia client coding. Even though the hypermedia design described earlier in this chapter did not indicate the `rel` values could contain more than one word, the client is coded to match the generally understood specification of a `rel` attribute.

```
function processLinks(response) {
  var xml, link, i, x, i, j, rels, href;

  g.links = [];
  xml = response.selectNodes('//link');
  for(i = 0, x = xml.length; i < x; i++) {
    href = xml[i].getAttribute('href');
    rels = xml[i].getAttribute('rel').split(' ');
    for(j = 0, y = rels.length; j < y; j++) {
      link = {'rel' : rels[j], 'href' : href};
      g.links[g.links.length] = link;
    }
  }
  showOptions();
}
```

```
function showOptions() {
  var elm, i, x, txt;

  txt = '';
  elm = document.getElementsByClassName('options')[0];
  if(elm) {
    for(i = 0, x = g.links.length; i < x; i++) {
      if(i>0){
        txt += ', ';
      }
      if(g.links[i].rel === 'collection') {
        txt += 'clear';
      }
      else {
        txt += g.links[i].rel;
      }
    }
    elm.innerHTML = txt;
  }
}
```

Once the options are displayed, the UI simply waits for the user to submit a command. Each command is validated against the current list (the ones for this representation) and, if valid, is processed and executed. Command processing includes validating the command, incrementing the number of moves executed by the user, updating the user's history, and then executing the command by selecting the associated `href` of the selected link to the server:

```
function move() {
  var elm, mv, href;

  elm = document.getElementsByName('move')[0];
  if(elm) {
    mv = elm.value;
    if(mv === 'clear') {
      reload();
    }
    else {
      href = getLinkElement(mv);
      if(href) {
        updateHistory(mv);
        getDocument(href);
      }
      else {
        alert(g.sorryMsg);
      }
    }
    setFocus();
  }
  return false;
}

function reload() {
  history.go(0);
}
```

```
function getLinkElement(key) {
  var i, x, rtn;

  for(i = 0, x = g.links.length; i < x; i++) {
    if(g.links[i].rel === key) {
      rtn = g.links[i].href;
      break;
    }
  }
  return rtn || '';
}

function updateHistory(mv) {
  var elm, txt;

  elm = document.getElementById('history');
  if(elm) {
    txt = elm.innerHTML;
    g.moves++;
    if(mv==='exit') {
      txt = g.moves +': ' + g.successMsg + '<br />' + txt;
    }
    else {
      txt = g.moves + ':' + mv + '<br />' + txt;
    }
    elm.innerHTML = txt;
  }
}
```

That is the complete set of markup and coding for this example application. You should notice that the application is a simple state machine that retrieves a document from the server, scans the document for links, updates the UI to reflect the returned links, and waits for the user to initiate the process again. This is the essence of hypermedia client implementation.

The author was able to reach the exit of this maze in just twelve steps.

Maze Bot Example

Since the maze application domain has a clear start and exit point, it is relatively easy to write an autonomous web bot to traverse the maze. It also helps that this hypermedia design is read-only and will not require the client to maintain any state to send back to the server. All the client needs to do is select links and make requests until the exit is reached.

Figure 2-2 shows the results of one run of the web bot.

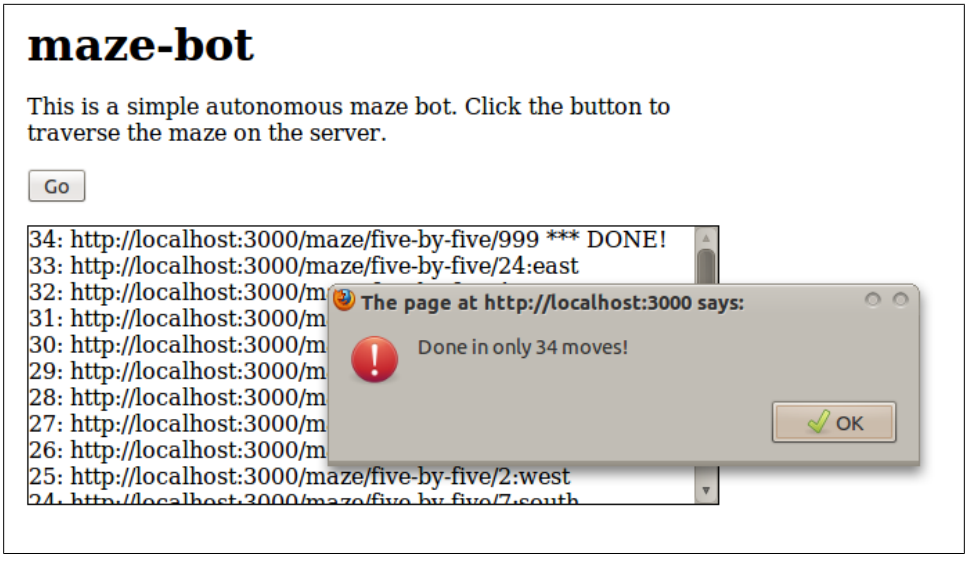


Figure 2-2. Maze Bot Screenshot

HTML5 Markup

The HTML5 markup for the web bot is very simple: just some text, a button, and a `DIV` to display the bot's progress through the maze. Below is the complete HTML5 for the `maze-bot.html` page:

```
<!DOCTYPE html>
<html>
  <head>
    <title>maze-bot</title>
    <meta http-equiv="Content-type" content="text/html;charset=UTF-8">
    <link rel="stylesheet" href="/stylesheets/maze-bot.css">
    <script type="text/javascript" src="/javascripts/ajax.js"></script>
    <script type="text/javascript" src="/javascripts/mozxpath.js"></script>
    <script type="text/javascript" src="/javascripts/maze-bot.js"></script>
  </head>
  <body>
    <h1>maze-bot</h1>
    <div id="display-space">
      <div id="notes">
        <p>
          This is a simple autonomous maze bot.
          Click the button to traverse the maze on the server.
        </p>
        <p>
          <button id="go">Go</button>
        </p>
      </div>
      <div id="game-play"></div>
    </div>
  </body>
</html>
```

JavaScript

The client-side script for the web bot is also relatively simple. The most interesting part of the script is the part that makes the decision of which link to follow for each move through the maze.

The first code block shows the initialization, including an array to hold the rules for selecting links, and the setup routine to ready the bot for traveling through the maze:

```
var g = {};
g.idx = 1;
g.links = [];
g.facing = '';
g.done = false;
g.start = false;
g.mediaType = "application/vnd.amundsen.maze+xml";
g.startLink = "http://localhost:3000/maze/five-by-five/";

// simple right-hand wall-following rules:
// if door-right, face right
// else-if door-forward, face forward
// else-if door-left, face left
// else face back
g.rules = {
  'east'  : ['south', 'east', 'north', 'west'],
  'south' : ['west', 'south', 'east', 'north'],
  'west'  : ['north', 'west', 'south', 'east'],
  'north' : ['east', 'north', 'west', 'south']
};

function init() {
  attachEvents();
  setup();
}

function attachEvents() {
  var elm;

  elm = document.getElementById('go');
  if(elm) {
    elm.onclick = firstMove;
  }
}

function setup() {
  var elm;

  g.done = false;
  g.start = false;

  elm = document.getElementById('game-play');
  if(elm) {
    elm.innerHTML = '';
  }
}
```

The second code block shows the first call to the server (to retrieve the item state of a maze) along with some support routines for locating a link in the current collection and displaying the bot's progress on the screen:

```
function firstMove() {
  if(g.done === true) {
    setup();
    firstMove();
  }
  else {
    g.idx = 1;
    getDocument(g.startLink);
  }
}

function getDocument(url) {
  ajax.httpGet(url, null, processLinks, true, 'processLinks', {'accept' :
g.mediaType});
}

function getLinkElement(key) {
  var i, x, rtn;

  for(i = 0, x = g.links.length; i < x; i++) {
    if(g.links[i].rel === key) {
      rtn = g.links[i].href;
      break;
    }
  }
  return rtn || '';
}

function printLine(msg) {
  var elm, txt, x;

  elm = document.getElementById('game-play');
  if(elm) {
    txt = elm.innerHTML;
    txt = g.idx++ + ': ' + msg + '<br>' + txt;
    elm.innerHTML = txt;
  }
}
```

The last code block is the one that shows the bot's decision tree for selecting the preferred link from those returned in the representation by the server. You can see that, after collecting all the links from the representation, the bot first looks to see if there is an `exit` link (indicating the end of processing). If there is no `exit`, the bot checks to see if there is a `start` link (indicating the start of a maze traversal). Finally, if there is no `exit` or `start` link, the bot uses the right-hand wall, following rules to select the most advantageous link (based on the direction the bot is facing) as the next step in the maze.

Note that the bot never checks the value of the URIs in each link, just the `rel` values associated with the link. This is another key to implementing successful hypermedia clients. Servers may change the actual URIs for links at any time. But, in good hypermedia designs, the `rel` values are predefined and static. Client applications that target `rel` values for processing will be successful even when servers need to change the actual URIs associated with those static `rel` values.

```
function processLinks(response) {
  var xml, i, x, j, y, rels, href, link, flg, rules;

  flg = false;
  rules = [];
  g.links = [];

  // get all the links in the response
  xml = response.selectNodes('//link');
  for(i = 0, x = xml.length; i < x; i++) {
    href = xml[i].getAttribute('href');
    rels = xml[i].getAttribute('rel').split(' ');
    for(j = 0, y = rels.length; j < y; j++) {
      link = {'rel' : rels[j], 'href' : href};
      g.links[g.links.length] = link;
    }
  }

  // is there an exit?
  href = getLinkElement('exit');
  if(href !== '') {
    g.done = true;
    printLine(href + ' *** DONE!');
    alert('Done in only ' + --g.idx + ' moves!');
    return;
  }

  // is there an entrance?
  if(flg === false && g.start === false) {
    href = getLinkElement('start');
    if(href !== '') {
      flg = true;
      g.start = true;
      g.facing = 'north';
      printLine(href);
    }
  }

  // ok, let's "wall-follow"
  if(flg === false) {
    rules = g.rules[g.facing];
    for(i = 0, x = rules.length; i < x; i++) {
      href = getLinkElement(rules[i]);
      if(href !== '') {
        flg = true;
```

```
        g.facing = rules[i];
        printLine(href);
        break;
      }
    }
  }

  // update pointer, handle next move
  if(href !== '') {
    getDocument(href);
  }
}
```

Wall-Following and Mazes

The algorithm used in the web bot example is called the *wall-follower* or *right-hand* or *left-hand* rule. It works for this test data because the maze described in the test data is a two-dimensional, simply connected maze (each wall in the maze is connected to at least one other wall). The solution is based on the principle that, in these types of mazes, if you place your hand on one wall as you travel through the maze you will eventually find an exit. Of course, there are also ways to construct mazes such that the wall-follower rule does not guarantee success.

Also, if you compare the results of the web bot's progress through the maze (the author's bot took 34 steps before reaching the exit), you will find that the wall-follower rule, while effective, is not very efficient.

Finally, there are ways to improve this bot's performance. That task is left to the reader.

Summary

This chapter used the scenario of representing a two-dimensional maze to illustrate a read-only, domain-specific hypermedia design with intrinsic application flow using the XML data format. The process of identifying top-level elements in the design as well as appropriate link relations for each state was outlined.

Some test data was generated and a simple server was implemented using the top-level design elements as guides to representing the set of possible states for the application.

Lastly, two example client implementations were shown. One based on humans making selections and traversing the maze, and one based on an autonomous web bot using a simple algorithm to select the appropriate links in order to reach the maze exit.

In the next chapter, a JSON-based hypermedia design that supports both read and write operations will be explored.

CHAPTER 3
JSON Hypermedia

I have the world's largest collection of seashells. I keep it on all the beaches of the world; perhaps you've seen it.

- Stephen Wright

The second example chapter explores the challenges and advantages of using JSON as a base for designing a hypermedia type. The scenario includes requirements for basic read/write semantics for managing simple lists (friends, tasks, blog entries, etc.). This scenario relies on generic design that supports the ability to create customized implementations that reflect specific application domains not considered during the initial design. After designing the type and documenting the application domain, a sample server is built using test data and two clients are to be implemented: one that employs a single-page interface (SPI) model for HTML browsers and one that offers a quick command-line interface built using only Node.js.

Scenario

Unlike the Maze+XML media type, which was designed to support a very specific scenario, this example needs to support a range of similar implementations, that of managing lists such as contacts, tasks, bookmarks, blog entries, comments, etc. Also, unlike the last example, this one requires support for full read/write access to the data. The hypermedia design should contain enough application control information to allow clients to know how (and when) to add, edit, and delete data from the list. Finally, users should be able to discover and execute predefined queries to filter the server responses.

As the reader may recall from the last example, the basic scenario is expressed from the point of view of the user or client-side application. This allows the hypermedia design to adopt the client-centered view and creates a design that will support the use case regardless of the available server-side components (file systems, databases, etc.). Ideally, the resulting design should be workable no matter what technologies, languages, or frameworks are available to server or client implementors.

As in all the examples in this book, the data will be stored in CouchDB and the server will be hosted in Node.js. The client applications will be written using HTML and JavaScript (for a browser sample) and using Node.js (for a command line sample).

Designing the Collection+JSON Media-Type

Since the clients will be coded using JavaScript, the data format for this hypermedia design will be JSON. Along with settling the other basic element decisions, it is also important to identify the state transitions for this scenario and then proceed to map out the solution.

Identifying the State Transitions

Since the use case calls for simple list management, there are just a few possible states that need to be represented:

- The collection of items
- A single item in the list
- A list of possible queries that can be executed against the item list
- A template or blueprint for adding or editing items
- Details of an error, if encountered along the way

For the states listed above, there are really only three interesting cases:

Collection State
: In this state, the response represents a set of items from the list. The response may also include details on the available queries and the template to use for writing items to the list. The possible transitions are: 1) select a single item (*Item State*), 2) add a new item (*Item State*), 3) execute one of the available queries (*Collection State*), or 4) reload the list (*Collection State*).

Item State
: In this state, the response represents a single item from the list. The response may include the list of available queries and/or the write template for editing this item. The possible transitions are: 1) update this item (*Item State*), 2) delete this item (*Collection State*), reload this item (*Item State*), or return to the list (*Collection State*).

Error State
: In this state, the response represents details on the most recent error encountered on the server. For this design there are no transitions supported.

Most all work can be done using only the two primary states (Collection and Item). However, as an additional design feature for this example, two more possible states will be added. Since it is possible that the list of queries could be long or not always

applicable (some users may not be able to see all possible queries), a new state will be added;

Query State
: In this state, the response represents the list of possible queries that can be executed against the collection. The possible transitions are: 1) execute a query (*Collection State*), 2) return to the list (*Collection State*), or 3) reload the list of queries (*Query State*).

It might also be necessary to allow clients to request a representation of only the template used to add/edit items. This can be handy since some users may not be allowed to add/edit the items or the template may not be available at all times (there may be times when no new items are allowed in the list, or the list cannot be edited, etc.). So there will be one more state for this design:

Template State
: In this state, the response represents the template that should be used to add or edit items. The possible transitions are: 1) Add a new item (*Item State*), 2) edit an existing item (*Item State*), return to the list (*Collection State*), or reload the template (*Template State*).

Selecting the Basic Design Elements

Just as in Chapter 2, the initial process of mapping out the design involves selecting the basic design elements.

For this example, since both clients will be written using JavaScript (the HTML and Node.js applications), the JSON data format is a good fit. It's also important to point out that the JSON data format is supported on many other platforms and languages. Even if we were implementing clients that did not rely on JavaScript, JSON could still be a viable choice. In this case, since the scenario calls for a design based on lists of links and data elements, the JSON data format's strong support for hash tables and arrays makes it a good fit for our design.

The scenario calls for the ability to support CRUD-style operations along with predefined filters or queries. This limited style of state transfer means the use of the Predefined State Transfer pattern would be a good fit. By establishing the mapping of HTTP methods against the collection URI (for POST[create]) and the item URI (for PUT[update] and DELETE), clients can easily support the full range of editing for the collection. Clients can also easily map HTTP GET to the collection, item, and query URIs to support the filters.

Since the design calls for creating a hypermedia type that can be applied to a range of related solutions (lists of friends, tasks, shopping items, etc.) adopting a Generic Domain style is a logical choice. That means the names for the objects, arrays, and properties can remain rather general (e.g. `collection`, `item`, `data`, `link`, etc.). Applying ad-

ditional semantic information can be done via the values applied to links (e.g. `rel="shopping-list"`), and the values for data elements (e.g. `name="first-name"`, etc.).

Finally, the application flow for this design can be built directly into the media type design: Intrinsic. The design calls for simple read/write and filtering so there does not need to be any detailed application flow described within the design. This is another case where coding client applications to recognize the state transfer elements (the collection and item URIs along with the query URIs) is relatively easy and reliable for a wide range of possible uses.

With these design decisions settled, it's time to map out the actual JSON document that can be used to represent a wide range of lists and related queries.

The Collection+JSON Document

For this design, instead of using the possible states and transitions as a starting point, the work will start with a single top-level layout of the possible elements in a Collection State response. From that point, additional details can be added as the design drills down into the lowest level of properties in the hypermedia type.

Below is a brief rendering of the possible contents of that Collection State of a Collection +JSON document.:

```
// sample colleciton.json map
{
  "collection" :
  {
    "version" : "1.0",
    "href" : URI,
    "links" : [ARRAY],
    "items :
    [
       {
         "href" : URI,
         "data" : [ARRAY],
         "links" : [ARRAY]
       },
       ...
    ],
    "queries" : [ARRAY],
    "template" :
    {
       "data" : [ARRAY]
    },
    "error" : {OBJECT}
  }
}
```

There are two simple properties in the collection object (`href` and `version`), two top-level arrays (`links` to hold collection-level URIs and `items` to hold the actual data in the

list), and two child objects (`template` and `error`). The items array has an `href` property, and two child arrays (`data` and `links`), and the template object as a child array (`data`).

That outlines the basics of this design. It is capable of representing a relatively wide range of list types. It can support add, edit, and delete actions using the URIs from the list, and each item along with a write template supplied to the server. It also supports representing a list of available queries that can be executed by the client application as needed.

Below is a more detailed set of design descriptions for the high-level objects and arrays in this JSON-based media type.

A complete set of documentation for this design can be found at the back of this book (See Appendix D).

Objects

In JSON documents, an object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array. The following elements are represented as objects in Collection+JSON documents: `collection`, `error`, and `template`.

The collection object. The `collection` object contains the items in the representation. It is a top-level document property. It has an `href` property that is the URI of the collection. This value can be used to load the collection document and, if supported by the server, can be used as the target URI for adding new items to the collection using HTTP POST.

The `collection` object may have the following child elements: `links`, `items`, `queries`, `template`. It is also possible that the collection response will contain an `error` object:

```
// sample collection object
{
  "collection" :
  {
    "version" : "1.0",
    "href" : URI,
    "links" : [ARRAY],
    "items : [ARRAY],
    "queries" : [ARRAY],
    "template" : {OBJECT},
    "error" : {OBJECT}
  }
}
```

The error object. The `error` object contains additional information on the latest error condition reported by the server. It is a top-level document property. The following elements can appear as child properties: `code`, `message`, and `title`.

```
// sample error object
{
  "error" :
  {
    "title" : STRING,
    "code" : STRING,
    "message" : STRING
  }
}
```

The template object. The `template` object contains all of the input elements used to add or edit collection items. It is a top-level document property and should contain a list of one or more `data` elements. Essentially, the template object is the equivalent in this design of the HTML FORM and INPUT elements. The template object has a variable number of data elements, each representing a name/value pair to be sent to the server for processing. Whenever a client wants to compose a payload for adding a new item, this is the guide. It should also be used when updating an existing item:

```
// sample template object
{
  "template" :
  {
    "data" :
    [
      {"name" : "full-name", "value" : "", "prompt" : "Your Full Name"},
      {"name" : "email", "value" : "", "prompt" : "Your Email Address"}
    ]
  }
}
```

Arrays

In JSON, an array is an ordered sequence of zero or more values. The following elements are always represented as arrays in Collection+JSON documents: `items`, `data`, `links`, and `queries`.

The data array. The data array is a child property of the `items` array and the `template` object. It contains one or more anonymous data objects. Each object can have any of three possible properties: `name` (REQUIRED), `value` (OPTIONAL), and `prompt` (OPTIONAL).

Below is an example of a `data` array within an `item`. This item has two data elements (`full-name` and `email`):

```
// example of a data array
"items" :
[
  {
    "href" : "http://example.org/friends/jdoe",
    "data" :
    [
      {"name" : "full-name", "value" : "J. Doe", "prompt" : "Full Name"},
      {"name" : "email", "value" : "jdoe@example.org", "prompt" : "Email"}
```

```
    ]
  }
]
```

The items array. The items array represents the list of records in the Collection+JSON document. It is a child property of the collection object.

Each element in the items array has an href property with a URI. This URI can be used to retrieve a response representing the associated item. It may also be used to edit (via HTTP PUT) or delete (via HTTP DELETE) the associated item. Typically item elements have a data array and a link array:

```
// sample items array
{
  "collection" :
  {
    "version" : "1.0",
    "href" : URI,
    "items" :
    [
      {
        "href" : URI,
        "data" : [ARRAY],
        "links" : [ARRAY]
      },
      ...
      {
        "href" : URI,
        "data" : [ARRAY],
        "links" : [ARRAY]
      }
    ]
  }
}
```

The links array. The links array is an optional child property of the items array. It contains one or more anonymous objects, each with five possible properties: `href` (REQUIRED), `rel` (REQURIED), `name` (OPTIONAL), `render` (OPTIONAL), and `prompt` (OPTIONAL).

Below is an item with the links collection illustrated:

```
// sample links array
{
  "collection" :
  {
    "version" : "1.0",
    "href" : URI,
    "items" :
    [
      {
        "href" : URI,
        "data" : [ARRAY],
        "links" :
        [
```

```
        {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING, "render" :
"image"},
        {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING}, "render" :
"link",
         ...
         {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING}
       ]
     }
   ]
  }
}
```

The `render` property is optional. If it is set to "image" then the `link` should be treated as an embeddable image (e.g. the HTML IMG tag). If it is missing or set to "link," the `link` should be treated as a navigation link (e.g. the HTML A tag).

The queries array. The queries array is an optional top-level property. It contains one or more anonymous objects; each object composed of five possible properties: href (REQUIRED), rel (REQUIRED), name (OPTIONAL), prompt (OPTIONAL), and a data array (OPTIONAL). If present, the data array represents query parameters for the associated href property of the same object:

```
// query template sample
{
  "queries" :
  [
    {
      "href : "http://example.org/search",
      "rel" : "search",
      "prompt" : "Enter search string",
      "data" :
      [
        {"name" : "search", "value" : ""}
      ]
    }
  ]
}
```

When the query object contains one or more data elements, those data elements are treated as a template for assembling queries. This is similar to the way HTML FORMs work when the method attribute is set to GET. In the above example, if a user entered "JSON" in the "search" data element, the client should construct a GET URI that looks like this:

```
http://example.org/search?search=JSON
```

The Tasks Application Semantics

In cases where the media type design is not Domain-Specific, additional information will be needed in order for clients and servers to share proper understanding of the messages passed between them. This additional information is sometimes called Domain Semantics or Application-Level Semantics. This added semantic information can

be documented, and that documentation can be used when implementing client and server applications that will use the media type to which the semantics are applied.

The details of Application-Level Semantics is covered in detail later in this book (see Chapter 4).

The process of documenting an application's domain semantics involves identifying data elements that are important for the domain (name, address, phone, etc.), the states to represent (a list of names, a single person, etc.), and the state transfers or data-passing operations (send a search string, etc.) needed to support the domain.

Application Semantics vs. Transfer Protocol

In this book, the phrases "Application Semantics" and "Domain Semantics" are used interchangeably. These phrases refer to the application-specific interactions of the target domain. For example, the ability to read and write users' records and query the list of existing users are application-specific. Some of these app-specific operations may require inputs (a search string, etc.), others may not. The exact data that is written to storage for a user record (name, email, address, etc.) are also application-specific. Describing the needed operations and data passed between parties can be viewed as the application semantics. These will be the same no matter what data format, media-type, or network protocol is used when implementing them.

The transfer protocol used for most web implementations is HTTP (Hypertext Transfer Protocol). However, there are other viable transfer protocols such as FTP (File Transfer Protocol), XMPP (Extensible Messaging and Presence Protocol), and others. Each protocol has its own set of interaction rules (methods, status codes, etc.) that are independent of any application. Just as application semantics are the same no matter the data format or media type, protocol rules are the same no matter the application that is implemented.

It is important to keep the semantics of the application and the rules of the protocol separate when implementing a design. This is especially handy if you want to be able to apply the same design details to a new protocol (e.g. implement a user search using XMPP *or* HTTP) or implement the same application semantics using a new media type (e.g. release an XML version of user search *and* a JSON version of user search).

How you go about designing the application semantic details is closely tied to the style of the target media type. For example, in a domain-generic design such as Collection +JSON the options for transferring state are limited to the CRUD+Query model. For Collection+JSON, documenting the application-level semantics means identifying the data model, populating the write template, and identifying the list of queries to support the problem domain.

The Data Model

For this chapter, a simple task management application will be implemented using the Collection+JSON media type. Below are the data model elements needed for a simple task list application:

Description
: The description of the task

Completed
: A simple flag (yes/no, true/false) to indicate the task has been completed

Date created
: Date the task was created (full day, month, year)

Date due
: Date the task is due (full day, month, year)

While the above fields may not cover all of the possibilities of a full-featured task management application, they are enough to illustrate the process of designing and implementing a domain-generic hypermedia type.

To translate this data model into Collection+JSON, the data properties shown above need to be expressed as an item in the Collection+JSON format. Below is a typical example:

```
// expressing the "tasks" data model in Collection.JSON
"items" :
[
  {
    "href" : "...",
    "data" :
    [
      {"name" : "description]", "value" : "this is my first task", "prompt" :
"Description"},
      {"name" : "completed", "value" : "true", "prompt" : "Completed"},
      {"name" : "dateCreated", "value" : "2011-09-01", "prompt" : "Date Created"},
      {"name" : "dateDue", "value" : "2011-09-03", "prompt" : "Date Due"}
    ]
  }
]
```

Notice that there is no value supplied in this example for the `href` property. The "tasks" data model will use this value as the unique identifier for the task item. The value of this property will be left to the server implementation.

A good media type design does not constrain servers' ability to determine their own resource identities. Resource identifiers are the responsibility of server implementors, not media type designers.

The example shown above also includes use of the Collection+JSON `prompt` property. This is an optional property in Collection+JSON, but will be handy in case client applications want to provide prompts to human users when rendering the data.

The Write Template

Another key element in the Collection+JSON hypermedia type is the Write Template. This template allows servers to tell clients the properties that can be written to the server at any given time. In the case of the tasks implementation example, the following write template can work for both create and update operations:

```
// write template for "tasks" application
{
  "template" :
  {
    "data" :
    [
      {"name" : "description]", "value" : "", "prompt" : "Description"},
      {"name" : "completed", "value" : "", "prompt" : "Completed"},
      {"name" : "dateDue", "value" : "", "prompt" : "Date Due"}
    ]
  }
}
```

The `data` array in the template above looks almost identical to the one shown in the previous section. The only difference is the `dateCreated` element that appears in the first array and not the second. This is a example of a read-only property implementation in the Collection+JSON media type.

Predefined Queries

The Collection+JSON design allows servers to return a list of possible queries when returning response representations to clients. The queries can be simple links or may allow clients to supply one or more optional arguments (similar to the way HTML form elements work).

For the task management implementation, four different queries will be used:

All
: Returns all of the records in the system

Open
: Returns all of the records where the `completed` property is set to "false"

Closed
: Returns all of the records where the `completed` property is set to "true"

Date Due
: Returns all of the records where the `dateDue` property is within the supplied date range; accepts two optional arguments: `dateStart` and `dateStop`

With these four queries defined, the next step is to express them using the Collection +JSON media type:

```
// query template sample
{
  "queries" :
  [
    {
      "href" : "...",
      "rel" : "all",
      "prompt" : "All tasks"
    },
    {
      "href" : "...",
      "rel" : "open",
      "prompt" : "Open tasks"
    },
    {
      "href" : "...",
      "rel" : "closed",
      "prompt" : "Closed tasks"
    },
    {
      "href : "..",
      "rel" : "date-due",
      "prompt" : "Date due",
      "data" :
      [
        {"name" : "dateStart", "value" : "", "prompt" : "Start Date"},
        {"name" : "dateStop", "value" : "", "prompt" : "Stop Date")
      ]
    }
  ]
}
```

As was seen in earlier examples, the `href` property is left blank in these designs to allow servers to determine the appropriate URIs for their own implementations.

This completes the task of defining the Task Management application-level semantics for the Collection+JSON hypermedia type. With the media type design completed and the application semantics defined, it is now possible to create a server that can render sample task data in response representations that Collection+JSON-aware clients can understand.

Sample Data

In order to validate both the hypermedia design and the application-level semantics, live data is needed. For this chapter, a simple starter data set will be created in CouchDB. It will support create/update/delete operations using simple validation rules. It will also provide four views to match the predefined queries identified in the previous section.

Once the data store is created, configured, and populated with data, we can move on to creating working server and client implementations that use the Collection+JSON media type.

Task Documents

In this implementation, each task will be stored as a document in CouchDB. Below is a sample task document:

```
{
  "_id" : "task1",
  "description" : "This is my first task.",
  "completed" : false,
  "dateCreated" : "2011-06-01",
  "dateDue" : "2011-12-31"
}
```

Note that the document matches the details of the data model previously outlined (see "The Data Model" on page 66). The _id property is the internal unique identifier required by CouchDB. Server implementations are free to use this value to create a public unique identifier (URI) for the data record.

Design Document

CouchDB uses a design document to hold a number of important aspects of a database including the validation routines for adding/updating records and a list of views that can be used to filter data lists for queries. Since the Collection+JSON media type allows for creating and updating records, this CouchDB implementation has a validation function as part of its design:

```
// couchdb input validation function
...
"validate_doc_update": "function(newDoc, oldDoc, userCtx) {

  function require(field, message) {
    message = message || field + ' is required';
    if (!newDoc[field]) {
        throw({forbidden : message});
    }
  };

  function unchanged(field) {
    if(oldDoc && toJSON(oldDoc[field]) !== toJSON(newDoc[field])) {
      throw({forbidden : field + ' is read-only'});
    }
  };

  require('description');
  require('dateCreated');
  require('dateDue');
```

```
    unchanged('dateCreated');

}"
  ...
```

In addition to the validation function, this database also has a set of predefined views to support the application semantics identified earlier (see "Predefined Queries" on page 67). Below is the list of view functions in the test database for this example:

```
// couchdb views for the "tasks" app
...
"views" : {

  "all" : {
    "map" : "function(doc){
      emit(doc.dateDue,doc);
    }"
  },

  "open" : {
    "map" : "function(doc) {
      if(doc.description && doc.dateCreated && doc.dateDue) {
        if(doc.completed===false) {
          emit(doc.dateDue,doc);
        }
      }
    }"
  },

  "closed" : {
    "map" : "function(doc) {
      if(doc.description && doc.dateCreated && doc.dateDue) {
        if(doc.completed===true) {
          emit(doc.dateDue,doc);
        }
      }
    }"
  },

  "due_date" : {
    "map" : "function(doc) {
      if(doc.description && doc.dateCreated && doc.dateDue) {
        emit(doc.dateDue,doc);
      }
    }"
  }
},
...
```

You may notice that the last view listed above (due_date) has no arguments even though the application semantics call for two optional arguments (`dateStart` and `dateStop`). CouchDB has a set of predefined query parameters for use for any views. For this example, CouchDB's `startkey` and `endkey` parameters will be used to support the `due_date` query.

The Server Code

With a test set of data in place, the next step is to implement a server that supports the Collection+JSON media type with the Tasks profile. As in Chapter 2 on XML designs, the server code shown here will only cover the highlights of the implementation. See "Source Code" on page 217 for details on how to obtain the full source code for this book.

The Collection Response

The collection response is, for this media type, the most common response. Our server implementation will always return a full Collection+JSON response including any collection links, queries, the write template, and the list of items. Below is the Node.js code to handle the response:

```
/* handle default task list */
app.get('/collection/tasks/', function(req, res){

  var view = '/_design/example/_view/due_date';

  db.get(view, function (err, doc) {
    res.header('content-type',contentType);
    res.render('tasks', {
      site  : 'http://localhost:3000/collection/tasks/',
      items : doc
    });
  });
});
```

Along with the above code, Node.js uses a layout and a template for the task list. Below are both of those templates.

First, the layout. Note the inclusion of the `profile` link at the top of the representation. While not required, it is a good idea to include a unique URI that identifies the application-level profile expressed by the representations:

```
{
  "collection" : {
    "href" : "<%=site%>",
    "version" : "1.0",

    "links" : [
      {"rel" : "author", "href" : "mailto:mamund@yahoo.com", "prompt" : "Author"},
      {"rel" : "profile", "href" : "http://amundsen.com/media-types/collection/
profiles/tasks/", "prompt" : "Profile"}
    ],

    "queries" : [
      {"rel" : "all", "href" : "<%=site%>;all", "prompt" : "All tasks"},
      {"rel" : "open", "href" : "<%=site%>;open", "prompt" : "Open tasks"},
      {"rel" : "closed", "href" : "<%=site%>;closed", "prompt" : "Closed tasks"},
      {"rel" : "date-range", "href" : "<%=site%>;date-range", "prompt" : "Date Range",
```

```
        "data" : [
          {"name" : "date-start", "value" : "", "prompt" : "Start Date"},
          {"name" : "date-stop", "value" : "", "prompt" : "Stop Date"}
        ]
      }
    ],

    "template" : {
      "data" : [
        {"name" : "description", "value" : "", "prompt" : "Description"},
        {"name" : "dateDue", "value" : "", "prompt" : "Date Due (yyyy-mm-dd)"},
        {"name" : "completed", "value" : "", "prompt" : "Completed (true/false)?"}
      ]
    },
    <%-body%>
  }
}
```

Next, the template for handling the task items. You can see the unique database id (`_id`) is used to create a unique URI for each item in the collection:

```
"items" : [
      <%for(i=0,x=items.length;i<x;i++) {%>
      {
        "href" : "<%=site%><%=items[i].value._id%>",
        "data" : [
          {"name" : "description", "value" : "<%=items[i].value.description%>",
"prompt" : "Description"},
          {"name" : "completed", "value" : <%=items[i].value.completed%>, "prompt" :
"Completed"},
          {"name" : "dateDue", "value" : "<%=items[i].value.dateDue%>", "prompt" :
"Date Due"}
        ]
      }
      <% if(i<items.length-1){%>,<% } %>
      <% } %>
    ]
```

The Item Response

The Collection+JSON media type also supports representing a single item in the collection. To reduce the size of the response for single items (and to show off some of the flexibility of the Collection+JSON media type design), this server implementation uses an alternate layout when representing single items.

Here is the Node.js code that formulates the response for a single task item:

```
/* handle single task item */
app.get('/collection/tasks/:i', function(req, res){

    var view = '/'+req.params.i;

    db.get(view, function (err, doc) {
    res.header('content-type',contentType);
```

```
    res.header('etag',doc._rev);
    res.render('task', {
      layout : 'item-layout',
      site  : 'http://localhost:3000/collection/tasks/',
      item : doc,
    });
  });
});
```

You can see in the above code the `item-layout` is used to craft the response representation. Below is the item layout:

```
{
  "collection" : {
    "href" : "<%=site%>",
    "version" : "1.0",

    "links" : [
      {"rel" : "author", "href" : "mailto:mamund@yahoo.com", "prompt" : "Author"},
      {"rel" : "profile", "href" : "http://amundsen.com/media-types/collection/
profiles/tasks/", "prompt" : "Profile"},
      {"rel" : "queries", "href" : "<%=site%>;queries", "prompt" : "Queries"},
      {"rel" : "template", "href" : "<%=site%>;template", "prompt" : "Template"}
    ],
    <%-body%>
  }
}
```

In this layout, the `links` collection contains two added members: the `queries` and `template` URIs. By including the links here, instead of the full query and template collections, the representation size can be reduced and client applications can still find the data they need by following these known links.

Below is the template for a single task item. The only difference between this template and the one used to render the full collection is that this template contains no loop implementation for multiple tasks:

```
"items" : [
      {
        "href" : "<%=site%><%=item._id%>",
        "data" : [
          {"name" : "description", "value" : "<%=item.description%>", "prompt" :
"Description"},
         {"name" : "completed", "value" : <%=item.completed%>, "prompt" : "Completed"},
        {"name" : "dateDue", "value" : "<%=item.dateDue%>", "prompt" : "Date Due"}
        ]
      }
    ]
```

The Query Representations

The application-level semantics for the tasks implementation identified four queries: `all`, `open`, `closed`, and `date-range`. All the queries can be executed using a simple static

URI except for the last one (`date-range`), which has two arguments: `date-start` and `date-stop`. Below are the routing routines in Node.js to handle these four query requests.

Note the handling of the two input arguments (`date-start` and `date-stop`) in the last routine:

```
/* filters */
app.get('/collection/tasks/;all', function(req, res){

    var view = '/_design/example/_view/all';

    db.get(view, function (err, doc) {
    res.header('content-type',contentType);
    res.render('tasks', {
      site  : 'http://localhost:3000/collection/tasks/',
      items : doc
    });
  });
});

app.get('/collection/tasks/;open', function(req, res){

    var view = '/_design/example/_view/open';

    db.get(view, function (err, doc) {
    res.header('content-type',contentType);
    res.render('tasks', {
      site  : 'http://localhost:3000/collection/tasks/',
      items : doc
    });
  });
});

app.get('/collection/tasks/;closed', function(req, res){

    var view = '/_design/example/_view/closed';

    db.get(view, function (err, doc) {
    res.header('content-type',contentType);
    res.render('tasks', {
      site  : 'http://localhost:3000/collection/tasks/',
      items : doc
    });
  });
});

app.get('/collection/tasks/;date-range', function(req, res){

    var d1 = (req.query['date-start'] || '');
    var d2 = (req.query['date-stop'] || '');

    var options = {};
    options.startkey=String.fromCharCode(34)+d1+String.fromCharCode(34);
    options.endkey=String.fromCharCode(34)+d2+String.fromCharCode(34);
```

```
    var view = '/_design/example/_view/due_date';

    db.get(view, options, function (err, doc) {
    res.header('content-type',contentType);
    res.render('tasks', {
      site  : 'http://localhost:3000/collection/tasks/',
      items : doc,
      query : view
    });
  });
});
```

Handling Template Writes

Along with supporting representations of the collection, items, and one or more predefined queries, the Collection+JSON media type supports performing create, update, and delete actions for individual items. Below are the three Node.js routines to handle the HTTP POST, PUT, and DELETE requests per the media type documentation:

```
/* handle creating a new task */
app.post('/collection/tasks/', function(req, res){

  var description, completed, dateDue, data;

  // get data array
  data = req.body.template.data;

  // pull out values we want
  for(i=0,x=data.length;i<x;i++) {
    switch(data[i].name) {
      case 'description' :
        description = data[i].value;
        break;
      case 'completed' :
        completed = data[i].value;
        break;
      case 'dateDue' :
        dateDue = data[i].value;
        break;
    }
  }

  // build JSON to write
  var item = {};
  item.description = description;
  item.completed = completed;
  item.dateDue = dateDue;
  item.dateCreated = today();

  // write to DB
  db.save(item, function(err, doc) {
    if(err) {
      res.status=400;
```

```
      res.send(err);
    }
    else {
      res.redirect('/collection/tasks/', 201);
    }
  });
});

/* handle updating an existing task item */
app.put('/collection/tasks/:i', function(req, res) {

  var idx = (req.params.i || '');
  var rev = req.header("if-match", "*");
  var description, completed, dateDue, data;

  // get data array
  data = req.body.template.data;

  // pull out values we want
  for(i=0,x=data.length;i<x;i++) {
    switch(data[i].name) {
      case 'description' :
        description = data[i].value;
        break;
      case 'completed' :
        completed = data[i].value;
        break;
      case 'dateDue' :
        dateDue = data[i].value;
        break;
    }
  }

  // build JSON to write
  var dt = new Date();
  var item = {};
  item.description = description;
  item.completed = completed;
  item.dateDue = dateDue;
  item.dateCreated = today();

  db.save(idx, rev, item, function (err, doc) {
    // return the same item
    res.redirect('/collection/tasks/'+idx, 302);
  });
});

/* handle deleting existing task */
app.delete('/collection/tasks/:i', function(req, res) {
  var idx = (req.params.i || '');
  var rev = req.header("if-match", "*");

  db.remove(idx, rev, function (err, doc) {
    if(err) {
      res.status=400;
```

```
        res.send(err);
      }
      else {
        res.status= 204;
        res.send();
      }
    });
});
```

In the above code, both the `app.post` and `app.put` routines expect a valid Collection +JSON `template` object as the body. This data is parsed and processed into a new internal `item` object and stored in the database. Notice also that the `app.put` and `app.delete` routines expect an `"if-match"` header, which is used as a concurrency check (revision) value by CouchDB. Finally, each routine has a simple response (redirection to a new URI) or simply a `204 - No Content` response.

The Client Code

With the server implementation completed, it's time to move on to building clients that understand the Collection+JSON hypermedia type. For this chapter, two clients will be implemented.

The first will be a browser-based Single Page Interface or SPI client. It relies on a small set of HTML and a script that is capable of parsing, rendering, and processing Collection+JSON messages. In other words, it is a Collection+JSON browser.

The second example application is a simple command line app implemented directly in Node.js. This shows that it is possible to build very simple apps that need not be full-blown hypermedia clients that offer 100% coverage for a custom media type.

The Tasks SPI Example

The SPI example shows that it is relatively easy to build a full-featured client that knows a custom hypermedia type. In this case, the client will render the data from the tasks profile described earlier in this chapter. However, the implementation details of this client are generic; they are targeted to the Collection+JSON media type, not a tasks media type. This means that this client will work just fine for any problem domain expressed in Collection+JSON messages. As long as the response representations match the rules defined in the Collection+JSON media type documentation, this client application will work just fine.

The client implementation shown here is rather basic from the UI perspective. The point here is to illustrate the work of building a client that understands the semantics of a custom media type, in this case, a Collection+JSON engine. There are many shortcuts taken here to reduce the amount of code and focus on only the key operations.

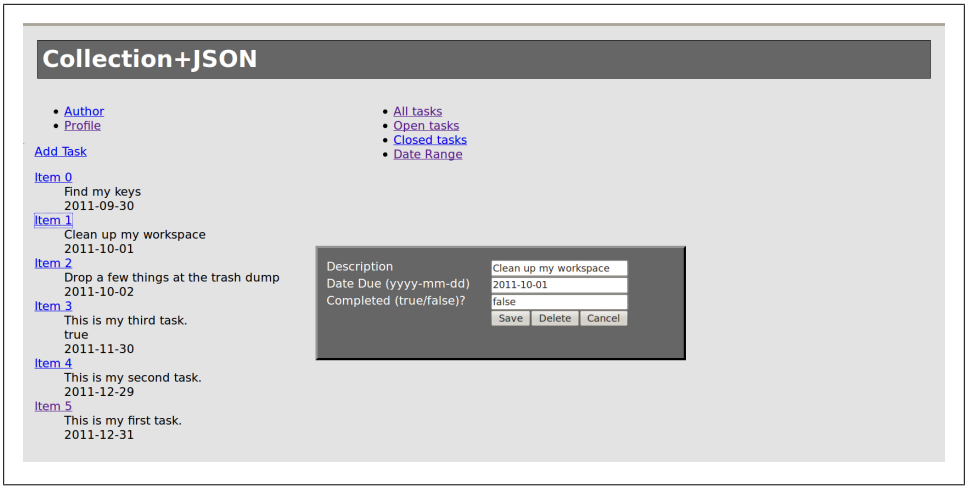


Figure 3-1. Collection+JSON SPI Screenshot

A screenshot of the Collection+JSON SPI screen can be seen in Figure 3-1.

HTML5 markup

As already mentioned, the markup for this example is very simple. The HTML5 page sets out important blocks in the UI and adds links to external JavaScript and CSS files. The JavaScript will do most all the work in the SPI example. The markup also includes a meta tag that points to the Task profile used in this implementation.

Here's the complete HTML5 markup:

```
<!DOCTYPE html>
<html>
  <!--
    /* 2001-05-25 (mca) : index.html */
    /* Designing Hypermedia APIs by Mike Amundsen (2011) */
  -->
  <head>
    <title>Collection+JSON</title>
    <meta name="profile" content="http://www.example.com/profiles/collection.json">
    <link rel="stylesheet" href="/stylesheets/collection.css" />
    <script type="text/javascript" src="/javascripts/collection.js"></script>
  </head>
  <body>
    <h1>Collection+JSON</h1>
    <div id="left">
       <div id="links" class="block"></div>
       <div id="write-template" class="block"></div>
       <div id="collection" class="block"></div>
       <div class="clear"> </div>
    </div>
    <div id="right">
       <div id="queries" class="block"></div>
```

```
        <div id="results" class="block"></div>
        <div class="clear"> </div>
      </div>
    </body>
  </html>
```

JavaScript

The client-side scripting for this example is, essentially, a Collection+JSON parser engine. The code needs to understand the Collection+JSON media type well enough to render responses on screen, handle inputs when needed, process requests for the collection, single items, queries, etc. The parser should also be able to understand the create, update, and delete rules for Collection+JSON representations.

While completing a Collection+JSON engine will take some detailed work, the good news is that it only needs to be written once. By the time this parser engine is complete, it will be able to handle any Collection+JSON representations, no matter what the problem domain (tasks, blogs, contact lists, etc.).

The basics of the Collection+JSON engine are as follows:

- Make a request to the server
- Process the response representation by loading the response and parsing it as JSON
- Render any link objects in the representation
- Render any query objects in the representation
- Render any item objects in the representation (might be a list, one, none)
- Use the template object to build up an input form (could be used for "create" or "update", possibly even for "delete")
- Wait for the user to take an action that results in a request to the server and go back to the start of this list

The code that handles the initial requests and processing the response representation includes handling requests for lists, a single item, and making query calls. Below are the routines involved:

```
function init() {
  g.filterUrl = getArg('filter');
  if(g.filterUrl!='') {
    loadList(unescape(g.filterUrl));
  }
  else {
    loadList();
  }
  showLinks();
  showItems();
  showQueries();
  buildTemplate();
}
```

```
  function loadList(href) {
    var ajax;
    var url = (href || g.collectionUrl);

    ajax=new XMLHttpRequest();
    if(ajax) {
      ajax.open('get',url,false);
      ajax.send(null);
      if(ajax.status===200) {
        g.data = JSON.parse(ajax.responseText);
      }
    }
  }

  function loadItem(href) {
    var ajax;

    ajax=new XMLHttpRequest();
    if(ajax) {
      ajax.open('get',href,false);
      ajax.send(null);
      if(ajax.status===200) {
        g.etag = ajax.getResponseHeader('etag');
        g.item = JSON.parse(ajax.responseText);
      }
    }
  }

  function filterData(href, rel) {
    var url, coll, i, x, a, args, data, part;

    coll = document.getElementsByTagName('a');
    for(i=0,x=coll.length;i<x;i++) {
      if(coll[i].rel===rel) {
        a = coll[i];
        break;
      }
    }

    url = window.location.href
    if(url.indexOf('?')!=-1) {
      url = url.substring(0,url.indexOf('?'))
    }

    args = (a.getAttribute('args') || '');
    if(args==='') {
      window.location = url + "?filter="+encodeURIComponent(href.replace('http://
localhost:3000',''));
    }
    else {
      data = JSON.parse(unescape(args));
      for(i=0,x=data.length;i<x;i++) {
        data[i].value = prompt(data[i].name);
      }
```

```
      url = url + "?filter="+encodeURIComponent(href.replace('http://localhost:
3000','')+'?');

      for(i=0,x=data.length;i<x;i++) {
        if(i>0) {
          url += encodeURIComponent('&');
        }
        url += encodeURIComponent(data[i].name+'='+data[i].value);
      }
      window.location = url;
    }
  }
```

Notice that the last routine (`filterData`) includes code to handle possible argument inputs for some queries. All the code here is generic enough so that any static or dynamic queries that appear in a response representation can be handled by this same code. This is a good example of a hypermedia engine.

The details of processing and rendering links, queries, and the actual item objects is covered in the following code routines:

```
function showLinks() {
  var dst;

  dst = document.getElementById('links');
  if(dst) {
    dst.appendChild(processLinks(g.data.collection.links));
  }
}

function showItems() {
  var dst, coll, dl, dt, dd, i, x;

  dst = document.getElementById('collection');
  if(dst) {
    dl = document.createElement('dl');

    coll = g.data.collection.items;
    if(coll) {
      // handle items
      for(i = 0, x = coll.length; i < x; i++) {
        dt = document.createElement('dt');
        dt.appendChild(processItemLink(coll[i], true, i));

        dd = document.createElement('dd');
        dd.title = coll[i].href;
        dd.appendChild(processData(coll[i].data));
        dd.appendChild(processLinks(coll[i].links));

        dl.appendChild(dt);
        dl.appendChild(dd);
      }
    }
    dst.appendChild(dl);
  }
```

```
}

function showQueries() {
  var dst;

  dst = document.getElementById('queries');
  if(dst) {
    dst.appendChild(processLinks(g.data.collection.queries));
  }
}

function processLinks(coll) {
  var ul, li, i, x, a, args

  ul = document.createElement('ul');

  if(coll) {
    for(i = 0, x = coll.length; i < x; i++) {
      a = document.createElement('a');
      a.href = coll[i].href;
      a.rel = coll[i].rel;
      a.className = coll[i].name || '';
      a.title = coll[i].name || '';

      if(coll[i].data) {
        args = JSON.stringify(coll[i].data);
        a.setAttribute('args',escape(args));
      }

      if(coll[i].rel!=='profile' && coll[i].rel!=='author') {
        a.onclick = function(){filterData(this.href, this.rel); return false;};
      }

      a.appendChild(document.createTextNode(coll[i].prompt || coll[i].rel));

      li = document.createElement('li');
      li.appendChild(a);
      ul.appendChild(li);
    }
  }
  return ul;
}

function processItemLink(item, editable, x) {
  var a, edit;
  edit = editable || true;

  a = document.createElement('a');
  if(item) {
    a.className = 'item-link';
    a.href = item.href;
    a.title = 'item-link';
    a.appendChild(document.createTextNode('Item '+x));
    if(edit === true) {
      a.onclick = function(){showItem(item.href); return false;};
```

```
    }
  }
  return a;
}

function processData(coll) {
  var i, x, ul, li, sp;

  ul = document.createElement('ul');

  if(coll) {
    for(i = 0, x = coll.length; i < x; i++) {
      if(coll[i].name && coll[i].value) {
        li = document.createElement('li');
        sp = document.createElement('span');
        sp.className = coll[i].name;
        sp.title = coll[i].name;
        sp.innerHTML = coll[i].value;
        li.appendChild(sp);
        ul.appendChild(li);
      }
    }
  }
  return ul;
}
```

The code above does all of the work to process the links, queries, and items returned in a response representation. The version shown here is very rudimentary; all `item` objects are rendered using HTML DL, DT, and DD elements, and the `data` collection within each `item` object is rendered using the HTML UL and LI elements. While not very pretty, it works well and consistently.

The final task for handling responses is constructing an input form using the `template` object passed in the representation. The code below handles this last chore:

```
function buildTemplate() {
  var dst, coll, i, x, form, fset;

  dst = document.getElementById('write-template');
  if(dst) {
    form = templateForm();
    fset = document.createElement('fieldset');

    coll = g.data.collection.template.data;
    for(i = 0, x = coll.length; i < x; i++) {
      fset.appendChild(processInputElement(coll[i]));
    }

    fset.appendChild(templateButtons());
    form.appendChild(fset);

    dst.appendChild(templateLink());
    dst.appendChild(form);
  }
}
```

```
function templateForm(href) {
  var form,action;
  action = href || g.collectionUrl;

  form = document.createElement('form');
  form.method="post";
  form.action=action;
  form.style.display='none';
  form.id = 'input-form';
  form.onsubmit = function(){submitInputForm();return false;};

  return form;
}

function processInputElement(item) {
  var lbl, inp, p;

  if(item) {
    lbl = document.createElement('label');
    lbl.for = item.name;
    lbl.innerHTML = item.prompt || item.name;

    inp = document.createElement('input');
    inp.type="text";
    inp.name = item.name;
    if(inp.value!='') {
      inp.value = item.value;
    }
    else {
      inp.placeholder = item.name;
    }

    p = document.createElement('p');
    p.appendChild(lbl);
    p.appendChild(inp);
  }
  return p;
}

function templateButtons() {
  var inp, p;

  inp = document.createElement('input');
  inp.type = 'submit';
  inp.value = 'Save';
  inp.name = 'save';
  p = document.createElement('p');
  p.className = 'buttons';
  p.appendChild(inp);

  inp = document.createElement('input');
  inp.type = 'button';
  inp.value = 'Delete';
  inp.name = 'delete';
```

```
  inp.onclick = function(){deleteItem(false);};
  inp.style.display='none';
  p.appendChild(inp);

  inp = document.createElement('input');
  inp.type = 'button';
  inp.value = 'Cancel';
  inp.name = 'cancel';
  inp.onclick = function(){toggleInputForm(false);};
  p.appendChild(inp);

  return p;
}

function templateLink() {
  var a, p;

  a = document.createElement('a');
  a.href='#';
  a.onclick = function(){toggleInputForm(); return false;};
  a.appendChild(document.createTextNode('Add Task'));
  p = document.createElement('p');
  p.className='input-block';
  p.appendChild(a);

  return p;
}
```

The last code block of interest in this example is the code that handles user actions once the representation has been rendered. The code for handling query links was already covered earlier in this section. The remaining code is the set of routines that renders the template object as an input form and processes the inputs into valid representations to send to the server. This handles the "create", "update", and "delete" semantics of the Collection+JSON media type:

```
function showItem(href) {
  loadItem(href);
  showEditForm(href);
}

function showEditForm(href) {
  var coll, dd, i, x, str;
  str = '';

  form = document.getElementById('input-form');
  if(form) {
    form.action = href;
    form.setAttribute('etag',g.etag);
  }

  coll = g.item.collection.items[0].data;

  for( i = 0, x = coll.length; i < x; i++) {
    name = coll[i].name;
    inp = document.getElementsByName(name)[0];
```

```
    if(inp) {
      inp.value = coll[i].value;
    }
  }

  for( i = 0, x = coll.length; i < x; i++) {
    name = coll[i].name;
    inp = document.getElementsByName(name)[0];
    if(inp) {
      inp.value = coll[i].value;
    }
  }

  inp = document.getElementsByName('delete')[0];
  if(inp) {
    inp.style.display = 'inline';
  }
  toggleInputForm(true);
}

function toggleInputForm() {
  var elm, coll, i, x;

  elm = document.getElementById('input-form');
  if(elm) {
    if(g.inputForm===true) {
      elm.style.display='block';
      g.inputForm=false;
    }
    else {
      elm.style.display='none';
      g.inputForm=true;
      coll = document.getElementsByTagName('input');
      for(i = 0, x = coll.length; i < x; i++) {
        if(coll[i].type === 'text') {
          coll[i].value = '';
        }
      }
      inp = document.getElementsByName('delete')[0];
      if(inp) {
        inp.style.display = 'none';
      }
    }
  }
}

function submitInputForm() {
  var item, form, coll, nam, val, i, x, z, etag, href;

  form = document.getElementById('input-form');
  if(form) {
    coll = form.getElementsByTagName('input');
    item = '{"template" : {"data" : [';
    z=0;
    for(i=0,x=coll.length;i<x;i++) {
```

```
        if(coll[i].type=="text") {
          if(z>0) {
            item +=",";
          }
          item += '{"name" : "'+coll[i].name+'", ';
          item += '"value" : '+(coll[i].value==='true'||coll[i].value==='false'?
coll[i].value+'}':'"'+coll[i].value+'"}');
          z++;
        }
      }
      item += "]}}";

      href = form.action;
      etag = form.getAttribute('etag');

      if(href)
      {
        ajax=new XMLHttpRequest();
        if(ajax) {
          if(etag && etag!=='') {
            ajax.open('put',href,false);
            ajax.setRequestHeader('if-match',etag);
          }
          else {
            ajax.open('post',href,false);
          }
          ajax.setRequestHeader('content-type',g.contentType)
          ajax.send(item);
          if(ajax.status>399) {
            alert('Error sending task!\n'+ajax.status);
          }
          else {
            window.location = window.location;
          }
        }
      }
    }
  }

  function deleteItem() {
    var form,href,etag,ajax;

    form = document.getElementById('input-form');
    if(form) {
      href = form.action;
      etag = form.getAttribute('etag');
      if(href) {
        ajax=new XMLHttpRequest();
        if(ajax) {
          ajax.open('delete',href,false);
          ajax.setRequestHeader('if-match',etag);
          ajax.send(null);
          if(ajax.status>399) {
            alert('Error deleting task!/n'+ajax.status+'\n'+ajax.statusText);
          }
```

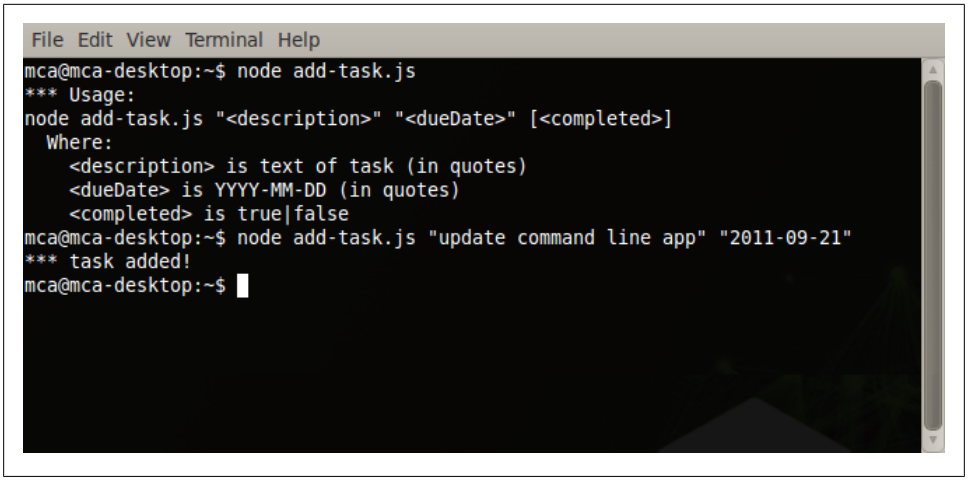


Figure 3-2. Add Tasks Command-line App Screenshot

```
            else {
              window.location = window.location;
            }
          }
        }
      }
      return false;
    }
```

That's all the code for this example. There were a few minor elements not shown (some configuration details, etc.), but it should be noted that the completed working example implements a fully functional Collection+JSON hypermedia engine in less than 500 lines of well-commented JavaScript. It's true that this implementation could benefit greatly from a more detailed UI treatment, but it is encouraging to know that a single set of 500 lines of JS code is all that is needed in order to support any and all servers that represent their resource responses as Collection+JSON.

The Tasks Command Line Example

As an example of another client implementation, a simple command-line interface can be created to allow users to quickly add tasks to their list. This example is implemented directly in Node.js and does not require a browser. It also shows how a client that understands a target hypermedia type need not always be written as a complete hypermedia engine for that media type. In this case, the client has been written to know how to do one thing well: add tasks to a list on the server.

A screenshot of the add-tasks command-line app in action can be seen in Figure 3-2.

The general approach

The general approach for this sample client is very simple: support writing tasks to a list on the server using the Collection+JSON hypermedia type. In this case, it is not important for the client to know how to render list and item responses or support queries. It only needs to write new tasks (not even edit or delete existing ones).

The reader might assume this means the client only needs to support one HTTP request: that of doing a POST to the server. But, in fact, this client makes two requests: one to retrieve the server's write template, and one to actually send data to the server using that template. In this way, the client is protected from a number of future changes to the server's template.

It's always a good idea to design client code to take advantage of hypermedia features, such as links and forms, within a media type design in order to future-proof the implementation as much as possible.

In order to simplify the coding and the requirements for typing on the command line, this application will have the initial URI (host, port, and path) written into the code. An alternate approach would be to use a configuration file to abstract these connection details from the code and/or force the user to supply them on every request. For this example, a more expedient route was taken: hardcode the initial URI.

The add-task.js application

The processing for this app is simple:

- Accept the user's input (show a help screen if they fail to supply the proper data)
- Retrieve the write template from the server (using an initial call to get the collection)
- Use the inputs from the user to populate the template with data
- Send the completed template to the server using HTTP POST as described in the Collection+JSON documentation

Below is the code to initialize the app and validate the user inputs:

```
var http = require('http');

// set the vars
var client, host, port, path, args, help;
host = 'localhost';
port = 3000;
path = '/collection/tasks/'
hdrs = {};
args = {};

help = '*** Usage:\n' +
   'node add-task.js "<description>" "<dueDate>" [<completed>]\n' +
   '  Where:\n' +
```

```
    '    <description> is text of task (in quotes)\n' +
    '    <dueDate> is YYYY-MM-DD (in quotes)\n' +
    '    <completed> is true|false';

// check args and fire off processing
if(process.argv.length<4 || process.argv.length>5) {
  console.log(help);
}
else {
  args.description = process.argv[2];
  args.dateDue = process.argv[3];
  args.completed = (process.argv[4]||false);

  client = http.createClient(port,host);
  getTemplate();
}
```

Note that the code is written to allow users to skip the `completed` argument and allow the app to supply the value of `false` as a default. In a production implementation, the inputs should also be validated for data type (string, date, boolean).

Once the inputs are validated, it's time to retrieve the Collection+JSON `template` object from the server:

```
// get the server's write template
function getTemplate() {
  // request the template
  var req = client.request('GET', path, {'host':host+':'+port});

  // event handlers
  req.on('response', function(response) {
    var body = ''
    response.on('data', function(chunk) {
      body += chunk;
    });
    response.on('error', function(error) {
      console.log(error);
    });
    response.on('end', function() {
      var data = JSON.parse(body);
      buildTask(data.collection.template);
    });
  });

  req.on('error', function(error) {
    console.log(error);
  });

  req.end();
}
```

The code above assumes that a call to the initial UI will always return a representation that contains the Collection+JSON template object. While this is true for the implementation we are testing, that might not be true for all servers. It is possible that servers will send only a link to the template object or maybe no template object at all (if the

data is read-only). This example skips over this detail in order to keep the code short and simple.

It is important to review the semantic requirements of the target media type carefully in order to account for any possible variances in the way representations may be presented by compliant servers. Never assume all servers will implement the media type the same way.

After collecting the template object from the server, the user's inputs can be added and a completed payload can be passed to the routine that will send the data to the server:

```
// write the task data to the server
function buildTask(template) {
  var coll, i, x, msg;

  // populate the template
  coll = template.data
  for(i=0,x=coll.length;i<x;i++) {
    switch(coll[i].name) {
      case 'description':
        coll[i].value = args.description;
        break;
      case 'dateDue':
        coll[i].value = args.dateDue;
        break;
      case 'completed':
        coll[i].value = args.completed;
        break;
    }
  }
  msg = '{"template" : '+JSON.stringify(template)+"}";
  sendData(msg);
}
```

Note that the code above loops through the template in order to match the user's inputs with the `data` elements in the template. This code will work no matter the order of the elements in the template. It will also work if the template has additional optional elements not supplied by the user. However, if the template has been modified to include new required elements or has changed the name of existing elements, this code is likely to break. A production implementation should keep track of the user's inputs and alert the user when new template elements are found or when the user's inputs are not applied.

Extending and versioning media type designs is covered in Chapter 5.

The final step is to take the new data and send it to the server using an HTTP POST request:

```
// send the data to the server for storage
function sendData(msg) {
  // set up request
  var hdrs = {
    'host' : host+':'+port,
    'content-type' : 'application/collection+json',
    'content-length' : (msg.length)
  };

  // pass data to the server
  var req = client.request('POST', path, hdrs);
  req.write(msg);

  // event handlers
  req.on('response', function(response) {
    var body = ''
    response.on('data', function(chunk) {
      body += chunk;
    });
    response.on('error', function(error) {
      console.log(error);
    });
    response.on('end', function() {
      console.log('*** task added!');
    });
  });

  req.on('error', function(error) {
    console.log(error);
  });

  req.end();
}
```

Summary

This chapter introduced the idea of a general media type design that can be used for similar problem domains. In this case, the general use case of managing a list was used as the motivation for the hypermedia design. JSON was used as the data format for a design with a Generic Domain style, Predefined State Transfers (using a CRUD+Query pattern), and Intrinsic application flow. The media type (Collection+JSON) not only supports representing lists and single items, but also allows servers to return a set of links for the collection, one or more queries (including ones that allow additional user input at runtime), and a template for indicating how client apps should format "create" and "update" operations against the list and its members.

Once the design was completed, a problem domain was selected (in this case, managing a list of tasks) and application-level semantics were defined for the problem domain.

This profile was documented in order to allow developers to write clients that not only understand the generic type but also target their implementations to the application semantics of the selected problem domain.

Test data was generated (some task data plus views to support the identified filter requests) and a server was implemented that supports the problem domain using Collection+JSON as the representation format.

Finally, two clients were built. The first one was a generic Single Page Interface (SPI) browser app that serves as a fully functional Collection+JSON engine that not only supports the test data target domain (managing tasks) but any other problem domain implemented using Collection+JSON for resource representations. A second client app (a command line interface) was also created to show how a hypermedia type can be used by applications that do not wish to implement a complete hypermedia engine for the media type.

The design and implementations for this chapter were more generic and a bit more abstract than the Domain Specific example shown in Chapter 2. In the next chapter, an even more generic media type design will be employed, using HTML5 as the base format, that relies solely on the application-level semantic profile to distinguish itself from other implementations using the same media type.

CHAPTER 4

HTML5 Hypermedia

The only usefulness of a map or a language depends on the similarity of structure between the empirical world and the map-languages.

- Alfred Korzybski

This last hands-on design chapter covers the techniques involved when using HTML5 as the base media type for your design. Since HTML already has a rich set of hypermedia controls (A, LINK, FORM, INPUT, etc.), the process of designing hypermedia types with HTML focuses on expressing the domain semantics within the existing elements and attributes of HTML.

The scenario here involves expressing common microblogging semantics (Twitter, Identi.ca, etc.) in a hypermedia type. After capturing the semantics and completing the design, a simple server will be built that emits the hypermedia responses along with two distinct client implementations. One client relies on only the HTML responses (no JavaScript support), and the other takes advantage of the Ajax style to build a web bot that knows enough about the hypermedia type to register as a new user (if needed) and begin posting into the data stream.

Scenario

For this example, a functional microblog hypermedia type is needed. This design should allow users to:

- Create, update, and delete accounts
- Add new messages and share existing messages
- Create and remove relationships to other users
- Search for existing users and messages

Since the design will be based on HTML5 (an existing hypermedia type), the implementation should provide basic functionality without the need for client-side scripting

(e.g. JavaScript). However, the design should also allow scripted clients (e.g. Ajax-style browser clients) to implement a rich user interface, too. As in the previous examples, data will be stored using CouchDB, the server will be implemented using Node.js, and the client applications will be implemented using HTML5 and JavaScript.

Designing the Microblog Media Type

The aim of this example is to illustrate the process of designing hypermedia types using HTML5 as the base format. Unlike the previous examples, which were based on XML and JSON (both formats devoid of native hypermedia controls), this example starts with a format that already supports basic hypermedia factors.

Starting with HTML5 provides both advantages and drawbacks when designing a hypermedia solution. The good news is the details of document design (which elements are valid, etc.), the specific hypermedia controls, and the method of expressing state transitions are already decided. This means designers do not need to worry about how to define and document much of the hypermedia type. However, with so many details already decided, there is less room for expressing the problem domain in ways clients can easily understand.

This duality of well-defined hypermedia controls and a limit on the options for expressing the problem domain is the key challenge behind working with existing hypermedia types and is, in part, the reason so few new APIs use HTML5 as the base. For many, it seems easier to start from zero using a non-hypermedia type such as XML or JSON and create all of the details from the beginning. But sometimes this option is not available to designers. Instead, designers often must be sure their implementations work well with existing clients (e.g. web browsers) with little or no support from custom scripting or other client-side plug-ins.

Expressing Application Domain Semantics in HTML5

In cases where the design needs to rely on existing hypermedia types like HTML5, a different approach is needed to express domain semantics. Instead of creating new elements (as in XML) or objects (as in JSON), domain semantics must be expressed using existing HTML5 elements (`<article>`, `<section>`, `<p>`, `<span>`, etc.). Instead of creating new state transition and process flow elements, designers need to use those already available in HTML5 (`<form>`, `<input>`, `<a>`, etc.).

However, it is still important to know the semantic value (the meaning in relation to the problem domain) of each of these already-defined elements. Instead of creating an `<email>` element or `<update-user>` transition block, designers need to use existing features of HTML5 to add semantic meaning to the representations. The way to do this is to use key HTML5 attributes to decorate the existing elements. These attributes are:

`id`
: Used to identify a unique data element/block within the representation

`name`
: Used to identify a state transition element (i.e. input) within the representation

`class`
: Used to identify a non-unique data element/block within the representation

`rel`
: Used to identify a non-unique process flow element within the representation

The `id` attribute can be applied to any element in the document, is a single string value (no spaces allowed), and must be unique in a document:

```
<span id="invoice-001">...</span>
```

The `name` attribute is also a single string value (without spaces). It can be assigned to a limited set of elements (those associated with input), and multiple elements within the same document can share the same value:

```
<label>Enter first invioce:</label><input name="invoice">...</input>
<label>Enter second invoice:</label><input name="invoice">...</input>
```

The `class` attribute can be applied to any element in the document, can appear on several elements within the document, and allows multiple values to be applied as long as they are separated by a space. This makes it possible to mark several document elements with the same value as well as mark the same element with several values:

```
<span class="important invoice overdue">...</span>
<span class="invoice pending">...</span>
```

The `rel` attribute has features of both the `class` and `name` attributes. Like the `name` attribute, a `rel` can only appear on a limited set of elements (those associated with links) and need not be unique within the document. Also, like the `class` attribute, the `rel` supports multiple strings separated by spaces:

```
<a href="..." rel="invoice">...</a>
<a href="..." rel="invoice overdue">...</a>
```

Through the proper application of these four attributes to a wide range of existing HTML5 elements, it is possible to adequately express any problem domain details within a hypermedia type design. It is worth noting that each of the above attributes has slightly different rules in HTML5.

Microformat, Microdata, and RDFa

The process of expressing application semantics using the `id`, `name`, `class`, and `rel` attributes of HTML5 described here is based loosely on the Microformat design principles. Microformat design patterns rely primarily on the `class` attribute, but also use the `id`, `name`, `rel`, and `rev` attributes. The author's method, while similar to Microformats, does not always follow the design patterns used in many published Microformats.

There are additional ways to add semantic information to HTML representations. At the time of this writing, the W3C has a draft specification for Microdata. Microdata consists of a group of items, each with a set of name-value pairs. These items can be nested, too. RDFa is a W3C specification that uses attributes to embed semantic data within markup languages, including the HTML family. As the name suggests, RDFa is a way to apply the principles of the Resource Description Framework to markup via attributes.

The primary aim here is to introduce the reader to the notion of adding semantic details to existing media types. It is beyond the scope of this text to cover these various standards for adding semantic information to documents. Instead these specifications are mentioned here in order to encourage the reader to explore them, and any other related standards, and learn the strengths and shortcomings of each. In the end, which method is used to add application domain semantics to a media type design choice is influenced by the preferences of both the media type author and the community (client and server developers, end users, etc.) for which the media type is created.

Identifying the State Transitions

Just as in other base formats, a key step in implementing a hypermedia design using HTML5 is the process of identifying the needed state transitions. As has already been stated above, this example implements the basic features of a microblogging application. Below is a set of states that need to be represented:

- The list of users
- A single user
- The list of messages
- A single message
- The list of possible queries (search users, view the user's list of followers, etc.)
- A template for creating a new user
- A template for updating an existing user
- A template for following an existing user
- A template for searching for existing users
- A template for adding a new message
- A template for replying to an existing message
- A template for searching for existing messages

As may be evident to the reader at this point, the above list identifies three unique block types within a representation (users, messages, queries) and seven transition blocks (create user, update user, follow user, search user, add message, reply to a message, and search messages). The unique blocks can be expressed using HTML5's `<div>` ele-

ment and the transition blocks can be expressed using the `<form>` element. Each of these blocks will have a number of child elements.

The usual error representation has been left out of this design to save time and reduce repetition of the same material. When creating a complete design for production use, you should always include an error representation.

The details of these element blocks can easily be expressed in short HTML5 examples. These examples will serve as reference material when creating the application profile later in this chapter (see "The Microblog Application Profile" on page 104).

State blocks

This example identifies three main types of content that may appear within a representation: users, messages, and queries. These can be expressed as HTML5 `<div>` elements with unordered lists (`<ul>`, `<li>`) and other child elements. Below are examples of each of the three main content blocks.

Users. There are two ways to represent users within this design: as a list of users and as a single user.

The list of users is represented using an HTML5 unordered list:

```
<!-- representing a list of users -->
<div id="users">
  <ul class="all">
    <li>
      <span class="user-text">User1</span>
      <a rel="user" href="..." title="profile for User1">profile</a>
      <a rel="messages" href="..." title="messages by User1">messages</a>
    </li>
    ...
    <li>
      <span class="user-text">UserN</span>
      <a rel="user" href="..." title="profile for UserN">profile</a>
      <a rel="messages" href="..." title="messages by UserN">messages</a>
    </li>
  </ul>
</div>
```

A single user is represented in a similar way:

```
<!-- representing a single user -->
<div id="users">
  <ul class="single">
    <li>
      <img class="user-image" src="..." alt="image of User1"/>
      <a rel="user" href="..." title="profile for User1">
        <span class="user-text">User One</span>
      </a>
      <span class="description">
```

```
        I am known to all as User One
      </span>
      <a rel="website" href="..." title="website">
        User1's web site
      </a>
      <a rel="messages" href="..." title="messages by User1">messages</a>
    </li>
  </ul>
</div>
```

Messages. Messages can also be represented in two ways (as a list and as a single message).

Here is the list representation:

```
<!-- representing a list of messages -->
<div id="messages">
  <ul class="all">
      <li>
        <span class="message-text">
          this is a message
        </span>
        @
        <a rel="message" href="..." title="message">
          <span class="date-time">
            2011-08-17 04:04:09
          </span>
        </a>
        by
        <a rel="user" href="..." title="User1">
          <span class="user-text">User1</span>
        </a>
      </li>
      ...
      <li>
        <span class="message-text">
          this is also a message
        </span>
        @
        <a rel="message" href="..." title="message">
          <span class="date-time">
            2011-08-17 04:10:13
          </span>
        </a>
        by
        <a rel="user" href="..." title="UserN">
          <span class="user-text">UserN</span>
        </a>
      </li>
  </ul>
</div>
```

And here is the way a single message is represented:

```
<!-- representing a single message -->
<div id="messages">
```

```
  <ul class="single">
    <li>
      <span class="message-text">
        This is a message
      </span>
      <span class="single">@</span>
      <a rel="message" href="..." title="message">
        <span class="date-time">
          2011-08-17 04:04:09
        </span>
      </a>
      <span class="single">by</span>
      <a rel="user" href="..." title="User1">
        <span class="user-text">User1</span>
      </a>
    </li>
  </ul>
</div>
```

Queries. This design uses a small set of simple queries. These are just links that return the list of messages, users, or the registration page:

```
<!-- representing the queries list -->
<div id="queries">
  <ul>
    <li><a rel="messages-all index" href="..." title="Home page">Home</a></li>
    <li><a rel="users-all" href="..." title="User List">Users</a></li>
    <li><a rel="register" href="..." title="Register">Register</a></li>
  </ul>
</div>
```

Note that the first link in the list has two values for the `rel` attribute.

Transfer blocks

This example identifies seven client-initiated state transfers. Each of them can be expressed as HTML5 `<form>` elements with `<input>` child elements. Below are each of the transfer blocks along with the details for each child element.

Create new user. The server will include this block in the representation to allow clients to create a new microblog user:

```
<!-- state transfer for adding a new user -->
<form method="post" action="..." class="user-add">
  <input type="text" name="name" value="" required="true"/>
  <input type="text" name="email" value="" required="true"/>
  <input type="password" name="password" value="" required="true" />
  <textarea name="description"></textarea>
  <input type="file" name="avatar" value="" />
  <input type="text" name="website" value="" />

  <input type="submit" value="Send" />
</form>
```

Notice that some fields are grouped under the SHOULD keyword and others are under the MAY keyword. These are just comments to indicate which fields "should" be returned by the server and which ones "may" be returned by the server. These annotations will be used later when writing up the documentation.

Update existing user. Once a user has been created, it should be possible to update that user account:

```
<!-- state transfer for updating an existing user -->
<form method="post" action="..." class="user-update">
  <input type="text" name="name" value="" required="true"/>
  <input type="text" name="email" value="" required="true"/>
  <input type="password" name="password" value="" required="true"/>
  <textarea name="description"></textarea>
  <input type="file" name="avatar" value="" />
  <input type="text" name="website" value="" />

  <input type="submit" value="Send" />
</form>
```

It is likely that update transitions will be prepopulated with existing data. It is also possible that one or more of the `<input />` elements will be marked as read-only by the server. These details will be left up to each server implementation to determine.

Follow a user. Following a user is as simple as sending the identifier of the user you wish to follow:

```
<!-- state transition to follow an existing user -->
<form method="post" action="..." class="user-follow">
  <input type="text" name="user" value="" required="true"/>

  <input type="submit" value="Send" />
</form>
```

You may notice that there is nothing in the above transition to indicate the current user that wants to follow the identity in the user input element. For all of the transition examples shown here, the server is assumed to be able to know (when it is important) the identity of the current (or logged-in) user. This can be done via information in the action URI or via control data (HTTP Headers) such as the Authorization header or the Cookie header (see "Current user and state data" on page 104 for details).

Search for users. Here is the transition block for searching for users:

```
<!-- state transfer for searching users -->
<form method="get" action="..." class="user-search">
  <input type="text" name="search" value="..." required="true"/>

  <input type="submit" value="Send" />
</form>
```

Since this is a search action, the method attribute is set to GET, not POST.

Add a new message. Adding a new message is also a simple state transfer:

```
<!-- state transfer for adding a message -->
<form method="post" action="..." class="message-post">
  <textarea name="message" requried="true"></textarea>

  <input type="submit" value="Send" />
</form>
```

Reply to an existing message. Replying to an existing message involves sending both the original user identifier and any reply text, which may include the original message text, to the server:

```
<!-- state transfer for replying to a message -->
<form method="post" action="..." class="message-reply">
  <input type="hidden" name="user" value="..." requried="true"/>
  <textarea name="message"></textarea>

  <input type="submit" value="Send" />
</form>
```

Search for messages. The message search transfer block is almost identical to the one used for user searches:

```
<!-- state transfer for searching messages -->
<form method="get" action="..." class="message-search">
  <input type="text" name="search" value="..." requried="true"/>

  <input type="submit" value="Send" />
</form>
```

Selecting the Basic Design Elements

Since the point of this chapter is to cover implementation details when using HTML5, the remaining standard design elements have already been decided. However, it is still valuable to go over each of them here.

Using HTML5 as the base format means that the State Transfer style will be ad hoc. HTML5 Forms will be used to make all client-imitated transfers via GET (read-only) or POST (add/update). Even though the ad hoc style is used in HTML5, designers can still establish required and optional inputs in their hypermedia design. In this example, the design uses both.

The domain style of HTML5 is agnostic. The element and attribute names are all independent of any domain semantics. However, as mentioned earlier in this chapter (see "Expressing Application Domain Semantics in HTML5" on page 96), HTML5 supports domain semantic expression using common attribute values (`id`, `name`, `class`). Finally, the Application Flow style for HTML5 is applied via values for the `rel` attribute on `href` tags.

Usually hypermedia designs use the `rel` attribute (or its equivalent in the data format) to mark all state transitions, including ones that require arguments. However, HTML5 does not support the `rel` attribute on the `form` element. For this reason, transitions that require arguments (e.g. forms) will be marked with a `class` attribute instead.

Although HTML5 is a domain-agnostic base format, the built-in state transfer and application flow elements make most of the design details very easy. The only creative work that needs to be done is arranging existing HTML5 elements (`article`, `section`, `div`, `p`, `span`, etc.) and decorating them with the necessary attributes (`id`, `name`, `class`, `rel`).

The application of these attribute decorations is covered in the next section.

The Microblog Application Profile

Since HTML5 is a domain-agnostic media type, all domain-specific information (both the data elements and the transition details) needs to be specified as additional information in each representation. As has already been pointed out earlier in this chapter (see "Expressing Application Domain Semantics in HTML5" on page 96), in HTML5 you can express domain-specific information using a set of attributes (`id`, `name`, `class`, and `rel`).

Also, this implementation will require users to log in before posting new messages. That means the implementation will need to be able to identify the current logged-in user.

Current user and state data

This example implementation will rely on HTTP's Authorization header to identify the currently logged-in user. That means this example server will use HTTP Basic Authentication for selected state transitions (Update a User, Follow a User, Add a New Message, and Reply to a Message). It is important to point out that user identification is not specified in the media type design; it is an implementation detail left to servers and clients to work out themselves.

The decision to leave user authentication independent of the media type has a number of advantages. First, this allows clients and servers to negotiate for an appropriate authentication scheme at runtime (the HTTP `WWW-Authenticate` head is used to advertise supported authentication schemes). Second, leaving it out of the media type means that servers are free to establish and transition details on their own. For example, Open Auth (OAuth) has a set of requirements for interacting with more than one web server in order to complete authentication. Finally, leaving authentication details out of the media type design allows servers to take advantage of whatever means may become available in the future.

ID attribute values

This design relies on three unique identifiers for representations:

messages
: Applied to a `div` tag. The list of messages in this representation. This list may contain only one message.

queries
: Applied to a `div` tag. The list of valid queries in this representation. This is a list of simple queries (represented by the HTML anchor tag).

users
: Applied to a `div` tag. The list of users in this representation. This list may contain only one user.

Class attribute values

There are a number of `class` attributes that can appear within a representation. Clients should be prepared to recognize the following values:

all
: Applied to a `UL` tag. A list representation. When this element is a descendant of `DIV.id="messages"` it MAY have one or more `LI.class="message"` descendant elements. When this element is a descendant of `DIV.id="users"` it MAY have one or more `LI.class="user"` descendant elements.

date-time
: Applied to a `SPAN` tag. Contains the UTC date-time the message was posted. When present, it SHOULD be valid per RFC3339.

description
: Applied to a `SPAN` tag. Contains the text description of a user.

friends
: Applied to a `UL` tag. A list representation. When this element is a descendant of `DIV.id="messages"` it contains the list of messages posted by the designated user's friends and MAY have one or more `LI.class="message"` descendant elements. When this element is a descendant of `DIV.id="users"` it contains the list of users who are the friends of the designated user and MAY have one or more `LI.class="user"` descendant elements.

followers
: Applied to a `UL` tag. A list representation of all the users from the designated user's friends list. MAY have one or more `LI.class="user"` descendant elements.

me
: Applied to a `UL` tag. When this element is a descendant of `DIV.id="messages"` it contains the list of messages posted by the designated user and MAY have one or more `LI.class="message"` descendant elements. When this element is a descendant

of `DIV.id="users"` it SHOULD contain a single descendant `LI.class="user"` with the designated user's profile.

mentions
: Applied to a `UL` tag. A list representation of all the messages that mention the designated user. It MAY contain one or more `LI.class="message"` descendant elements.

message
: Applied to an `LI` tag. A representation of a single message. It SHOULD contain the following descendant elements:

```
SPAN.class="user-text"
A.rel="user"
SPAN.class="message-text"
A.rel="message"
```

It MAY also contain the following descendant elements:

```
IMG.class="user-image"
SPAN.class="date-time"
```

message-post
: Applied to a `FORM` tag. A link template to add a new message to the system by the designated (logged-in) user. The element MUST be set to `FORM.method="post"` and SHOULD contain a descendant element:

```
TEXTAREA.name="message"
```

message-reply
: Applied to a `FORM` tag. A link template to reply to an existing message. The element MUST be set to `FORM.method="post"` and SHOULD contain the following descendant elements:

`INPUT[hidden].name="user"` (the author of the original post)
`TEXTAREA.name="message"`

single
: When this element is a descendant of `DIV.id="messages"` it contains the message selected via a message link. SHOULD have a single `LI.class="message"` descendant element. When this element is a descendant of `DIV.id="users"` it contains the user selected via a user link. SHOULD have a single `LI.class="user"` descendant element.

messages-search
: Applied to a `FORM` tag. A link template to search of all the messages. The element MUST be set to `FORM.method="get"` and SHOULD contain the following descendant elements:

```
INPUT[text].name="search"
```

message-text
: Applied to a `SPAN` tag. The text of a message posted by a user.

search
: Applied to a `UL` tag. A list representation. When this element is a descendant of `DIV.id="messages"` it contains a list of messages and MAY have one or more `LI.class="message"` descendant elements. When this element is a descendant of `DIV.id="users"` it contains a list of users and MAY have one or more `LI.class="user"` descendant elements.

shares
: Applied to a `UL` tag. A list representation of all the messages posted by the designated user that were shared by other users. It MAY contain one or more `LI.class="message"` descendant elements.

user
: Applied to an `LI` tag. A representation of a single user. It SHOULD contain the following descendant elements:

```
SPAN.class="user-text"
A.rel="user"
A.rel="messages"
```

  It MAY also contain the following descendant elements:

```
SPAN.class="description"
IMG.class="avatar"
A.rel="website"
```

user-add
: Applied to a `FORM` tag. A link template to create a new user profile. The element MUST be set to `FORM.method="post"` and SHOULD contain the following descendant elements:

```
INPUT[text].name="user"
INPUT[text].name="email"
INPUT[password].name="password"
```

  It MAY also contain the following descendant elements:

```
TEXTAREA.name="description"
INPUT[file].name="avatar"
INPUT[text].name="website"
```

user-follow
: Applied to a `FORM` tag. A link template to add a user to the designated user's friend list. The element MUST be set to `FORM.method="post"` and SHOULD contain the descendant element:

```
INPUT[text].name="user"
```

user-image
: Applied to an `IMG` tag. A reference to an image of the designated user.

user-text
: Applied to a `SPAN` tag. The user nickname text.

user-update
: Applied to a `FORM` tag. A link template to update the designated user's profile. The element MUST be set to `FORM.method="post"` and SHOULD contain the following descendant elements:

```
INPUT[hidden].name="user"
INPUT[hidden].name="email"
INPUT[password].name="password"
```

It MAY also contain the following descendant elements:

```
TEXTAREA.name="description"
INPUT[file].name="avatar"
INPUT[text].name="website"
```

users-search
: Applied to a `FORM` tag. A link template to search of all the users. The element MUST be set to `FORM.method="get"` and SHOULD contain the descendant element:

```
INPUT[text].name="search"
```

Name attributes values

HTML5 uses the `name` attribute to identify a representation element that will be used to supply data to the server during a state transition. Clients should be prepared to supply values for the following state transition elements:

description
: Applied to a TEXTAREA element. The description of the user.

email
: Applied to an INPUT[text] or INPUT[hidden] element. The email address of a user. When supplied, it SHOULD be valid per RFC5322.

message
: Applied to a TEXTAREA element. The message to post (for the designated user).

name
: Applied to an INPUT[text] element. The (full) name of a user.

password
: Applied to an INPUT[password] element. The password of the user login.

search
: Applied to an INPUT[text]. The search value to use when searching messages (when applied to `FORM.class="message-search"`) or when searching users (when applied to `FORM.class="users-search"`).

user
: Applied to an INPUT[text] or INPUT[hidden] element. The public nickname of a user.

avatar
: Applied to an INPUT[file] element. The image for the user.

website
: Applied to an INPUT[text]. The URL of a website associated with the user profile. When supplied, it SHOULD be valid per RFC 3986.

Rel attribute values

This design also identifies a number of possible simple state transitions, or static links, that may appear within representations. These will appear as HTML anchor tags with the following `rel` attribute values:

index
: Applied to an A tag. A reference to the starting URI for the application.

message
: Applied to an A tag. A reference to a message representation.

message-post
: Applied to an A tag. A reference to the message-post FORM.

message-reply
: Applied to an A tag. A reference to the message-reply FORM.

message-share
: Applied to an A tag. A reference to the message-share FORM.

messages-all
: Applied to an A tag. A reference to a list representation of all the messages in the system.

messages-search
: Applied to an A tag. A reference to the messages-search FORM.

user
: Applied to an A tag. A reference to a user representation.

user-add
: Applied to an A tag. A reference to the user-add FORM.

user-follow
: Applied to an A tag. A reference to the user-follow FORM.

user-update
: Applied to an A tag. A reference to the user-update FORM.

users-all
: Applied to an A tag. A reference to a list representation of all the users in the system.

users-friends
: Applied to an A tag. A reference to list representation of the designated user's friend users.

users-followers
: Applied to an A tag. A reference to list representation of the users who follow the designated user.

users-search
: Applied to an A tag. A reference to the users-search FORM.

website
: Applied to an A tag. A reference to the website associated with a user.

Sample Data

The test data for this example design can be represented using three different documents in CouchDB: the User, Message, and Follows documents. This implementation also relies on a CouchDB design document that includes a handful of views and a validation routine for writing documents into the data store.

User Documents

For this implementation, users will be represented in the data store as follows:

```
{
  "_id" : "mamund",
  "type" : "user",
  "name" : "Mike Amundsen",
  "email" : "mamund@yahoo.com",
  "password" : "p@ssWOrd",
  "description" : "learnin hypermedia",
  "imageUrl" : "http://amundsen.com/images/mca-photos/mca-icon-b.jpg",
  "websiteUrl" : "http://amundsen.com",
  "dateCreated" : "2011-06-21"
}
```

Message Documents

Each message document in the data store looks like this:

```
{
  "type" : "post",
  "text" : "My first message!",
  "user" : "mamund",
  "dateCreated" : "2011-06-29"
}
```

Follow Documents

The system will also track which users follow each other using a Follow document:

```
{
  "type" : "follow",
  "user" : "mamund",
  "follows" : "lee"
}
```

Design Document

The predefined views and validation routines for the CouchDB data for this implementation are:

```
{
  "_id" : "_design/microblog",

  "views" : {

    "users_search" : {
      "map" : "function(doc){
        if(doc._id && doc.type==='user') {
          emit(doc._id,doc);
        }
      }"
    },
    "users_by_id" : {
      "map" : "function(doc){
        if(doc._id && doc.type==='user') {
          emit(doc._id,doc);
        }
      }"
    },

    "posts_all" : {
      "map" : "function(doc) {
        if(doc._id && doc.type==='post') {
          emit(doc.dateCreated.split('-'),doc);
        }
      }"
    },
    "posts_by_id" : {
      "map" : "function(doc) {
        if(doc._id && doc.type==='post') {
          emit(doc._id,doc);
        }
      }"
    },
    "posts_by_user" : {
      "map" : "function(doc) {
        if(doc._id && doc.type==='post') {
          emit(doc.user, doc);
        }
      }"
```

```
  },
  "posts_search" : {
    "map" : "function(doc) {
      if(doc._id && doc.type==='post') {
        emit(doc.user, doc);
      }
    }"
  },
  "posts_by_user" : {
    "map" : "function(doc) {
      if(doc.user && doc.type==='post') {
        emit(doc.dateCreated.split('-').concat(doc.user),doc);
      }
    }"
  },

  "follows_user_is_following" : {
    "map" : "function(doc) {
      if(doc.user && doc.type==='follow') {
        emit(doc.user, {_id:doc.follows});
      }
    }"
  },
  "follows_is_following_user" : {
    "map" : "function(doc) {
      if(doc.follows && doc.type==='follow') {
        emit(doc.follows, {_id:doc.user});
      }
    }"
  }
},

"validate_doc_update": "function(newDoc, oldDoc, userCtx) {

  function require(field, message) {
    message = message || field + ' is required';
    if (!newDoc[field]) {
        throw({forbidden : message});
    }
  };

  function unchanged(field) {
    if(oldDoc && toJSON(oldDoc[field]) !== toJSON(newDoc[field])) {
      throw({forbidden : field + ' is read-only'});
    }
  };

  if(newDoc._deleted) {
    return true;
  }
  else {
    switch(newDoc.type) {
      case 'user':
        require('name');
        require('email');
```

```
            require('password');
            break;
          case 'post':
            require('text');
            require('user');
            require('dateCreated');
            break;
          case 'follow':
            require('user');
            require('follows');
            break;
        }
      }
    }"
}
```

The Server Code

With sample data in place, the next step is to implement server code to test the design. Since this example application has quite a few state transitions, the server-side code is a bit more involved than previous examples. However, an advantage of using HTML5 as the base media type is that it is rather easy to test since the server output will easily render within common web browsers.

Authenticating Users

This implementation relies on HTTP Basic Authentication to identify users. Below is the top-level routine to handle requesting the username and password and then comparing the results to information stored in a CouchDB User document:

```
/* validate user (from  db) via HTTP Basic Auth */
function validateUser(req, res, next) {

  var parts, auth, scheme, credentials;
  var view, options;

  // handle auth stuff
  auth = req.headers["authorization"];
  if (!auth){
    return authRequired(res, 'Microblog');
  }

  parts = auth.split(' ');
  scheme = parts[0]
  credentials = new Buffer(parts[1], 'base64').toString().split(':');

  if ('Basic' != scheme) {
    return badRequest(res);
  }
  req.credentials = credentials;

  // ok, let's look this user up
```

```
  view = '/_design/microblog/_view/users_by_id';

  options = {};
  options.descending='true';
  options.key=String.fromCharCode(34)+req.credentials[0]+String.fromCharCode(34);;

  db.get(view, options, function(err, doc) {
    try {
      if(doc[0].value.password===req.credentials[1]) {
        next(req,res);
      }
      else {
        throw new Error('Invalid User');
      }
    }
    catch (ex) {
      return authRequired(res, 'Microblog');
    }
  });
};
```

In the above routine, the code first checks for the presence of the `Authorization` header. If it does not exist, the server sends a response that asks the client to supply credentials before continuing (the `authRequired` method call). Once an `Authorization` header is supplied by the client, the code first ensures that it uses the Basic authentication scheme and then proceeds to parse the header into its two key parts (`username:password`). The first part (`username`) is used to perform a look up against the data store and, if a record is found, the second part (`password`) is compared against the data store in the User document. If there is a match, then the user has been authenticated and the code execution continues as usual (the `next` method call).

Registering a New User

Since this implementation supports adding new users, there is a view for rendering the state transition form and code to handle both returning the form and processing the posted data (using POST).

First, here is the code to return the input form:

```
/* get user register page */
app.get('/microblog/register/', function(req, res){

  res.header('content-type',contentType);
  res.render('register', {
    title: 'Register',
    site: baseUrl,
  });
});
```

And the view template to render that form:

```
<h2 id="page-title"><%= title %></h2>
<form class="user-add" action="<%=site%>users/" method="post">
```

```
  <fieldset>
    <h4>Account</h4>
    <p class="input">
      <label>Handle:</label>
      <input name="user" value=""
        placeholder="coolhandle" required="true"/>
    </p>
    <p class="input">
      <label>Password:</label>
      <input name="password" type="password" value=""
        placeholder="mys3cr3t" required="true" />
    </p>
  </fieldset>
  <fieldset>
    <h4>User Info</h4>
    <p class="input">
      <label>Email Address:</label>
      <input name="email" type="email" value=""
        placeholder="user@example.com"  required="true"/>
    </p>
    <p class="input">
      <label>Full Name:</label>
      <input name="name" value="" placeholder="Jane Doe"/>
    </p>
    <p class="input">
      <label>Description:</label>
      <textarea name="description"></textarea>
    </p>
    <p class="input">
      <label>Avatar URL:</label>
      <input name="avatar" type="url" value=""
        placeholder="http://example.com/images/my-avatar.jpg"/>
    </p>
    <p class="input">
      <label>Website URL:</label>
      <input name="website" type="url" value=""
        placeholder="http://example.com/my-blog/"/>
    </p>
  </fieldset>
    <p class="buttons">
      <input type="submit" value="Submit" />
      <input type="reset" value="Reset" />
    </p>
</form>
```

In the above template, you can see that some fields are marked as `required="true"` and some have `placeholder` values included to give users a hint on how to fill out the various inputs.

Below is the server code that runs when users submit the state transition form:

```
/* post to user list page */
app.post('/microblog/users/', function(req, res) {

  var item,id;
```

```
  id = req.body.user;
  if(id===''') {
    res.status=400;
    res.send('missing user');
  }
  else {
    item = {};
    item.type='user';
    item.password = req.body.password;
    item.name = req.body.name;
    item.email = req.body.email;
    item.description = req.body.description
    item.imageUrl = req.body.avatar;
    item.websiteUrl = req.body.website;
    item.dateCreated = today();

    // write to DB
    db.save(req.body.user, item, function(err, doc) {
      if(err) {
        res.status=400;
        res.send(err);
      }
      else {
        res.redirect('/microblog/users/', 302);
      }
    });
  }
});
```

Message Responses

There are two responses for representing messages: the message list and message details. There is also a state transition for adding a new message to the data store.

Below is the code that returns the message list:

```
/* starting page */
app.get('/microblog/', function(req, res){

  var view = '/_design/microblog/_view/posts_all';

  var options = {};
  options.descending = 'true';

  ctype = acceptsXml(req);

  db.get(view, options, function(err, doc) {
    res.header('content-type',ctype);
    res.render('index', {
      title: 'Home',
      site: baseUrl,
      items: doc
    });
  });
});
```

And the view template that renders the message list:

```
<h2 id="page-title"><%= title %></h2>
<form class="message-post" action="<%=site%>messages/" method="post">
  <fieldset>
    <h4>What's Up?</h4>
    <textarea name="message" cols="50" rows="1" size="140" required="true"></textarea>
    <span class="message-buttons">
      <input type="submit" value="Submit" />
      <input type="reset" value="Reset" />
    </span>
  </fieldset>
</form>

<div id="messages">
  <ul class="all">
    <% for(i=0,x=items.length;i<x;i++) { %>
      <li>
        <span class="message-text">
          <%=items[i].value.text%>
        </span>
        @
        <a rel="message" href="<%=site%>messages/<%=items[i].value._id%>"
title="message">
          <span class="date-time">
            <%=items[i].value.dateCreated%>
          </span>
        </a>
        by
        <a rel="user" href="<%=site%>users/<%=items[i].value.user%>" title="<
%=items[i].value.user%>">
          <span class="user-text"><%=items[i].value.user%></span>
        </a>
      </li>
    <% } %>
  </ul>
</div>
```

Note that this view template also includes the state transition block for adding a new message to the system (`message-post`). Below is the server code that handles the message-post state transition:

```
// add a message
app.post('/microblog/messages/', function(req, res) {

  validateUser(req, res, function(req,res) {

    var text;

    // get data array
    text = req.body.message;
    if(text!=='') {
      item = {};
      item.type='post';
      item.text = text;
      item.user = req.credentials[0];
```

```
      item.dateCreated = now();

      // write to DB
      db.save(item, function(err, doc) {
        if(err) {
          res.status=400;
          res.send(err);
        }
        else {
          res.redirect('/microblog/', 302);
        }
      });
    }
    else {
      return badReqest(res);
    }
  });
});
```

Finally, here is the code to handle representing a single message. This can be reached by activating the `rel="message"` links in the message list:

```
/* single message page */
app.get('/microblog/messages/:i', function(req, res){

  var view, options, id;
  id = req.params.i;

  view = '/_design/microblog/_view/posts_by_id';
  options = {};
  options.descending='true';
  options.key=String.fromCharCode(34)+id+String.fromCharCode(34);

  db.get(view, options, function(err, doc) {
    res.header('content-type',contentType);
    res.render('message', {
      title: id,
      site: baseUrl,
      items: doc
    });
  });
});
```

Below is the template for rendering a single message response:

```
<div class="message-block">
  <div id="messages">
    <ul class="single">
      <% for(i=0,x=items.length;i<x;i++) { %>
        <li>
          <span class="message-text">
            <%=items[i].value.text%>
          </span>
          <span class="single">@</span>
          <a rel="message" href="<%=site%>messages/<%=items[i].value._id%>"
title="message">
```

```
            <span class="date-time">
              <%=items[i].value.dateCreated%>
            </span>
          </a>
          <span class="single">by</span>
          <a rel="user" href="<%=site%>users/<%=items[i].value.user%>" title="<
%=items[i].value.user%>">
            <span class="user-text"><%=items[i].value.user%></span>
          </a>
        </li>
      <% } %>
    </ul>
  </div>
   
</div>
```

User Responses

This sample implementation has two representations for user responses: the user list and the user details. Below is the user list server code:

```
/* get user list page */
app.get('/microblog/users/', function(req, res){

  var view = '/_design/microblog/_view/users_by_id';

  db.get(view, function(err, doc) {
    res.header('content-type',contentType);
    res.render('users', {
      title: 'User List',
      site: baseUrl,
      items: doc
    });
  });
});
```

And the user list view to match:

```
<h2 id="page-title"><%= title %></h2>
<div id="users">
  <ul class="all">
    <% for(i=0,x=items.length;i<x;i++) { %>
      <li>
        <span class="user-text"><%= items[i].value.name %></span>
        <a rel="user" href="<%=site%>users/<%=items[i].value._id%>"
           title="profile for <%=items[i].value._id%>">profile</a>
        <a rel="messages" href="<%=site%>user-messages/<%=items[i].value._id%>"
           title="messages by <%=items[i].value._id%>">messages</a>
      </li>
    <% } %>
  </ul>
</div>
```

There is also code to return a single user record:

```
/* single user profile page */
app.get('/microblog/users/:i', function(req, res){

  var view, options, id;
  id = req.params.i;

  view = '/_design/microblog/_view/users_by_id';
  options = {};
  options.descending='true';
  options.key=String.fromCharCode(34)+id+String.fromCharCode(34);

  db.get(view, options, function(err, doc) {
    res.header('content-type',contentType);
    res.render('user', {
      title: id,
      site: baseUrl,
      items: doc
    });
  });
});
```

Along with the view template for rendering single user responses:

```
<h2 id="page-title"><%= title %></h2>
<div id="users">
  <ul class="single">
      <li>
        <% if(items[0].value.imageUrl) { %>
        <img class="avatar" src="<%=items[0].value.imageUrl%>" />
        <% } %>

        <a rel="user" href="<%=site%>users/<%=items[0].value._id%>"
          title="profile for <%=items[0].value._id%>">
          <span class="user-text"><%= items[0].value.name %></span>
        </a>

        <% if(items[0].value.description) { %>
        <span class="description">
          <%=items[0].value.description%>
        </span>
        <% } %>

        <% if(items[0].value.websiteUrl) { %>
        <a rel="website" href="<%=items[0].value.websiteUrl%>" title="website">
          <%=items[0].value.websiteUrl%>
        </a>
        <% } %>

        <a rel="messages" href="<%=site%>user-messages/<%=items[0].value._id%>"
           title="messages by <%=items[0].value._id%>">messages</a>
      </li>
  </ul>
</div>
```

Note that the single user template will optionally include the user's profile image (`user-image`), description, and website URL if they have been supplied. You should also notice

that this representation includes a link to see all of the messages created by this user (rel="messages"). The server code and view template for that response are:

```
/* user messages page */
app.get('/microblog/user-messages/:i', function(req, res){

  var view, options, id;

  id = req.params.i;

  view = '/_design/microblog/_view/posts_by_user';
  options = {};
  options.descending='true';
  options.key=String.fromCharCode(34)+id+String.fromCharCode(34);;

  db.get(view, options, function(err, doc) {
    res.header('content-type',contentType);
    res.render('user-messages', {
      title: id,
      site: baseUrl,
      items: doc
    });
  });
});
```

```
<h2 id="page-title">Messages from <%=title%></h2>
<div class="user-message-block">
  <div id="messages">
    <ul class="search">
      <% for(i=0,x=items.length;i<x;i++) { %>
        <li>
          <span class="message-text">
            <%=items[i].value.text%>
          </span>
          <span class="single">@</span>
          <a rel="message" href="<%=site%>messages/<%=items[i].value._id%>"
            title="message">
            <span class="date-time">
              <%=items[i].value.dateCreated%>
            </span>
          </a>
          <span class="single">by</span>
          <a rel="user" href="<%=site%>users/<%=items[i].value.user%>"
            title="<%=items[i].value.user%>">
            <span class="user-text"><%=items[i].value.user%></span>
          </a>
        </li>
      <% } %>
    </ul>
  </div>
   
</div>
```

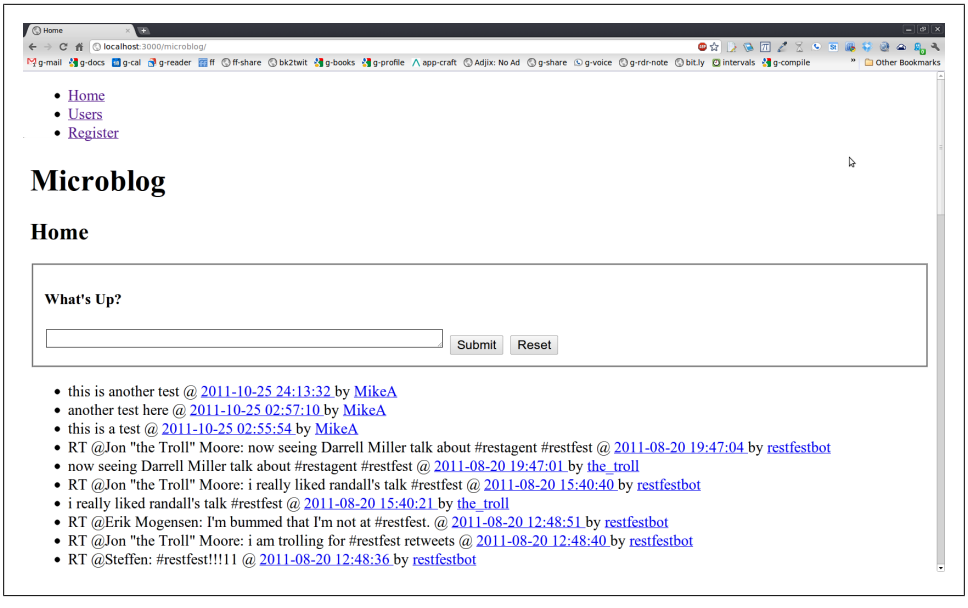


Figure 4-1. Microblog Plain POSH Client Screenshot

The Client Code

Once the server is implemented, it's time to work through example clients. As was already mentioned, using HTML5 as the based media type means that the implementation will just run within common web browsers without any modification. However, plain HTML5 (without a Cascading Style Sheet, or CSS) is not very pleasing to the eye. It is a rather easy process to create a CSS stylesheet to spiff up plain HTML5 into a decent looking client. This results in a client that supports all of the required functionality without relying on any client-side scripting. This is sometimes called a Plain Old Semantic HTML or POSH client.

It is also possible to treat well-formed HTML5 as an XML document and render the responses using an Ajax-style user interface. This does, however, require that the HTML5 be rendered as valid XML. Lucky for our case, the view templates already meet this requirement.

The POSH Example

The basic HTML5 that is rendered by the server is fully functional, but a bit unappealing to view as can be seen in Figure 4-1.

However, with just a bit of CSS work, this view can be turned into a much more inviting user interface (see Figure 4-2).

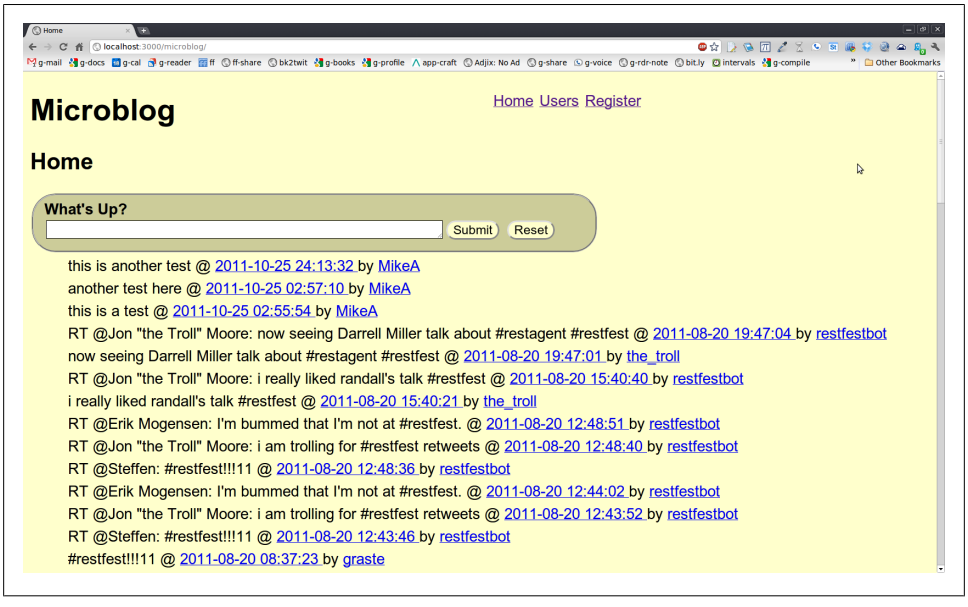


Figure 4-2. Microblog CSS POSH Client Screenshot

The CSS file for this rendering includes rules for rendering the home page and message lists:

```
body {
  background-color: #FFFFCC;
  font-family: sans-serif;
}

div#queries {
  width:500px;
  float:right;
}

div#queries ul {
  margin:0;
  list-style-type:none;
}
div#queries ul li {
  float:left;
  padding-right: .4em;
}

div#messages ul {
  margin:0;
  list-style-type:none;
}
div#messages ul li {
  margin-top: .3em;
}
```

```
div#messages ul.single li span.message-text {
  font-size:large;
  font-weight:bold;
}

div.message-block {
  border: 1px solid black;
  padding:.5em;
  width:500px;
  margin:auto;
  -moz-border-radius: 25px;
  border-radius: 25px;
}

div#messages ul.single li a,
div#messages ul.single li span.single {
  float:left;
  margin-right: .3em;
}

ul.single {
  margin:0;
  list-style-type:none;
}

ul.single a,
ul.single span,
ul.single img
{
  display:block;
}
```

The CSS document also includes details for rendering state transition blocks (HTML forms):

```
form.message-post {
  width:600px;
}
form.message-post textarea {
  display:block;
  float:left;
}
span.message-buttons {
  display:block;
  float:left;
}

fieldset {
  margin-top: 1em;
  -moz-border-radius: 25px;
  border-radius: 25px;
  background-color: #CCCC99;
}
fieldset h4 {
  margin:0;
```

```
}

form.user-add {
  width:300px;
}

p.input {
  margin-top:0;
  margin-bottom:0;
}
p.input label {
  display:block;
  width:150px;
}
p.input input,
p.input textarea {
  float:left;
  width:200px;
}

span.message-buttons input {
  background-color: #ffffcc;
  -moz-border-radius: 15px;
  border-radius: 15px;
}
p.buttons input {
  background-color: #cccc99;
  -moz-border-radius: 15px;
  border-radius: 15px;
}
```

Quite a bit more work could be done in this area, too. The CSS specification offers a wide range of options that make rendering POSH responses very easy and straightforward. The example here is given as just a starter for those who want to explore the area of user interface design via CSS.

The Ajax QuoteBot Example

In this example, a small Ajax client that knows enough about the microblogging hypermedia profile to interact with any server that supports this media type will be implemented. The bot shown here is able to determine if it needs to register as a new account on the server and then post quotes into the data stream at regular intervals. This example application shows that HTML can be successfully used as a machine-to-machine media type. It also provides guidance on one way to write stand-alone client applications for machine-to-machine scenarios.

Below is an example run of the QuoteBot Figure 4-3.

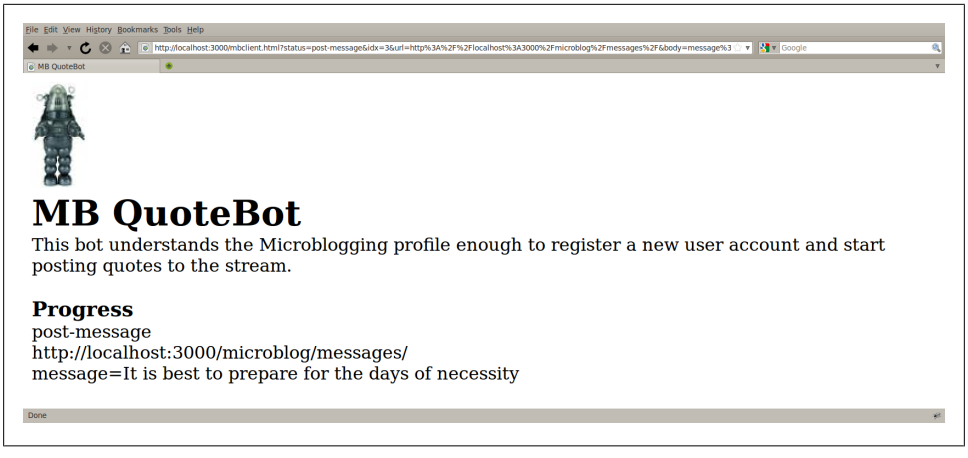


Figure 4-3. Microblog QuoteBot Screenshot

The QuoteBot scenario

For this example, the QuoteBot client will have the job of writing quotes to the microblogging server. If needed, this bot will also be able to register a new account on the target server. In order to accomplish these tasks, the QuoteBot will need to understand enough of the microblogging profile to perform a handful of low-level tasks such as loading the server's home page, getting a list of users, finding and completing the user registration form, posting new messages, etc. Below is a list of these tasks along with notes on how the bot can use the microblogging profile specification to accomplish them:

- Load an entry page on the server (the starting URI)
- Get a list of registered users for this server (find the `@rel="users-all"` link)
- See if the bot is already registered (find the `@rel="user"` link with the value of the bot's username)
- Load the user registration page (find the `@rel="register"` link)
- Find the user registration form (locate the `@class="user-add"` form on the page)
- Fill out the user registration form and submit it to the server (find the input fields outlined in the spec, populate them, and send the data to the server)
- Load the message post page (`@rel="message-post"` link)
- Find the message post form (locate the `@class="message-post"` form on the page)
- Fill out the message post form and submit it to the server (find the input fields identified in the spec, populate, and send)

As the list above illustrates, the process of coding a machine-to-machine client for hypermedia interactions involves identifying two types of elements in response representations: links and forms. Clients need to be able to activate a link (i.e. follow the link) and fill in forms and activate them, too (i.e. submit the data to the server). Essentially,

the client application needs to understand links and forms in general as well as the semantic details of the particular hypermedia profile.

Reachable Links and Forms

It is important to point out that hypermedia clients depend on servers to make these links and forms available (or reachable). In other words, the bot must be able to successfully find them. It is also possible to write client applications that can use multilevel searching and backtracking in order to find target links and forms buried at various locations in the Web, then track these links and forms for future use. This is very much what a link spider does when traversing and indexing the WWW.

In this example implementation, all the links needed for this work are presented either on the home page (or starting URI) or on a page reachable from that home page. This makes the task of finding links and forms relatively easy for machine clients and simplifies the illustration for this chapter.

QuoteBot HTML5

The HTML5 for the QuoteBot example is very minimal. All of the activity on the page is driven by the associated JavaScript. Below is the complete HTML5 markup:

```
<!DOCTYPE html>
<html>
  <head>
    <title>MB QuoteBot</title>
    <script type="text/javascript" src="javascripts/base64.js"></script>
    <script type="text/javascript" src="javascripts/mbclient.js"></script>
    <style>
      h1 {margin:0;}
      h2,h3,h4 {margin-bottom:0;}
      p {margin-top:0;}
      div#side {float:left;margin:auto 1em auto auto;}
      div#main {float:left;}
    </style>
  </head>
  <body>
    <sidebar>
      <div id="side">
        <img src="http://amundsen.com/images/robot.jpg" />
      </div>
    </sidebar>
    <article>
      <div id="main">
        <header>
          <h1>MB QuoteBot</h1>
        </header>
        <section>
          <p>
            This bot understands the Microblogging profile enough to
            register a new user account and
            start posting quotes to the stream.
```

```
        </p>
      </section>
      <section>
        <h3>Progress</h3>
        <p id="status"></p>
      </section>
    </div>
    <div id="output">
    </div>
  </article>
 </body>
</html>
```

QuoteBot JavaScript

The JavaScript for the QuoteBot looks as if it is complicated, but is actually rather simple. The code can be separated into a handful of sections:

Setup
: This code contains the initialization code, the variables used to fill out expected forms, and the list of quotes to send to the server.

Making requests
: This is a short bit of general code used to format and excecute an HTTP request. The code is smart enough to include a body and authentication information, if needed.

Processing responses
: This section contains all of the methods used to process the response representations from the server. This includes parsing the response, looking for links, and looking for and filling in forms. These routines either conclude with an additional request or, in case of an error, stop and report the status of the bot and stop.

Supporting routines
: These are utility functions to inspect arguments in the URI, format a URI for the next request, and look for elements in the response.

Setup code. The setup code includes details on shared state variables, details on forms to fill out, error messages, and a list of quotes to send to the server:

```
var g = {};

/* state values */
g.startUrl = '/microblog/'
g.wait=10;
g.status = '';
g.url = '';
g.body = '';
g.idx = 0;

/* form@class="add-user" */
g.user = {};
g.user.user = 'robieBot5';
```

```
  g.user.password = 'robie';
  g.user.email = 'robie@example.org';
  g.user.name = 'Robie the Robot';
  g.user.description = 'a simple quote bot';
  g.user.avatar = 'http://amundsen.com/images/robot.jpg';
  g.user.website = 'http://robotstxt.org';

  /* form@class="message-post" */
  g.msg = {};
  g.msg.message = '';

  /* errors for this bot */
  g.errors = {};
  g.errors.noUsersAllLink = 'Unable to find a@rel="users-all" link';
  g.errors.noUserLink = 'Unable to find a@rel="user" link';
  g.errors.noRegisterLink = 'Unable to find a@rel="register" link';
  g.errors.noMessagePostLink = 'Unable to find a@rel="message-post" link';
  g.errors.noRegisterForm = 'Unable to find form@class="add-user" form';
  g.errors.noMessagePostForm = 'Unable to find form@class="message-post" form';
  g.errors.registerFormError = 'Unable to fill out the form@class="add-user" form';
  g.errors.messageFormError = 'Unable to fill out the form@class="message-post" form';

  /* some aesop's quotes to post */
  g.quotes = [];
  g.quotes[0] = 'Gratitude is the sign of noble souls';
  g.quotes[1] = 'Appearances are deceptive';
  g.quotes[2] = 'One good turn deserves another';
  g.quotes[3] = 'It is best to prepare for the days of necessity';
  g.quotes[4] = 'A willful beast must go his own way';
  g.quotes[5] = 'He that finds discontentment in one place is not likely to find
happiness in another';
  g.quotes[6] = 'A man is known by the company he keeps';
  g.quotes[7] = 'In quarreling about the shadow we often lose the substance';
  g.quotes[8] = 'They are not wise who give to themselves the credit due to others';
  g.quotes[9] = 'Even a fool is wise-when it is too late!';
```

Making requests. When the page first loads, a set of state variables are populated based on data in the query string. This data is then used to fire off a request to the microblogging server:

```
function init() {
  g.status = getArg('status')||'start';
  g.url = getArg('url')||g.startUrl;
  g.body = getArg('body')||'';
  g.idx = getArg('idx')||0;

  updateUI();
  makeRequest();
}

function newQuote() {
  g.idx++;
  nextStep('start');
}
```

```
  function updateUI() {
    var elm;

    elm = document.getElementById('status');
    if(elm) {
      elm.innerHTML = g.status + '<br />' + g.url + '<br />' + unescape(g.body);
    }
  }

  function makeRequest() {
    var ajax, data, method;

    ajax=new XMLHttpRequest();
    if(ajax) {
      ajax.onreadystatechange = function() {
        if(ajax.readyState==4 || ajax.readyState=='complete') {
          processResponse(ajax);
        }
      };

      if(g.body!=='') {
        data = g.body;
        method = 'post';
      }
      else {
        method = 'get';
      }

      ajax.open(method,g.url,true);

      if(data) {
        ajax.setRequestHeader('content-type','application/x-www-form-urlencoded');
        ajax.setRequestHeader('authorization','Basic '+Base64.encode(g.user.user
+':'+g.user.password));
      }

      g.url='';
      g.body='';

      ajax.setRequestHeader('accept','application/xhtml+xml');
      ajax.send(data);
    }
  }
```

Processing responses. Processing responses is the heart of this example application. Each response representation from the server can potentially contain links and/or forms of interest for this bot. The code needs to know what is expected in this response (e.g. "There should be a 'register' link in this response somewhere...") and know how to find it (e.g. "Give me all of the links in this response and see if one is marked `rel='register'`"). This first code snippet shows the routine used to route response representations to the proper function for processing:

```
/* these are the things this bot can do */
function processResponse(ajax) {
```

```
  var doc = ajax.responseXML;

  if(ajax.status===200) {
    switch(g.status) {
      case 'start':
        findUsersAllLink(doc);
        break;
      case 'get-users-all':
        findMyUserName(doc);
        break;
      case 'get-register-link':
        findRegisterLink(doc);
        break;
      case 'get-register-form':
        findRegisterForm(doc);
        break;
      case 'post-user':
        postUser(doc);
        break;
      case 'get-message-post-link':
        findMessagePostForm(doc);
        break;
      case 'post-message':
        postMessage(doc);
        break;
      case 'completed':
        handleCompleted(doc);
        break;
      default:
        alert('unknown status: ['+g.status+']');
        return;
    }
  }
  else {
    alert(ajax.status)
  }
}
```

The following function looks for a link and responds accordingly:

```
function findMyUserName(doc) {
  var coll, url, href, found;

  found=false;
  url=g.startUrl;

  coll = getElementsByRelType('user', 'a', doc);
  if(coll.length===0) {
    alert(g.errors.noUserLink);
  }
  else {
    for(i=0,x=coll.length;i<x;i++) {
      if(coll[i].firstChild.nodeValue===g.user.user) {
        found=true;
        break;
      }
```

```
    }

    if(found===true) {
      g.status = 'get-message-post-link';
    }
    else {
      g.status = 'get-register-link';
    }
    nextStep(g.status,url);
  }
}
```

This bot is also able to locate a form and fill it in based on data already in memory:

```
function findRegisterForm(doc) {
  var coll, url, msg, found, i, x, args, c, body;

  c=0;
  args = [];
  found=false;

  elm = getElementsByClassName('user-add','form',doc)[0];
  if(elm) {
    found=true;
  }
  else {
    alert(g.errors.noRegisterForm);
    return;
  }

  if(found===true) {
    url = elm.getAttribute('action');

    coll = elm.getElementsByTagName('input');
    for(i=0,x=coll.length;i<x;i++) {
      name = coll[i].getAttribute('name');
      if(g.user[name]!==undefined) {
        args[c++] = {'name':name,'value':g.user[name]};
      }
    }
    coll = elm.getElementsByTagName('textarea');
    for(i=0,x=coll.length;i<x;i++) {
      name = coll[i].getAttribute('name');
      if(g.user[name]!==undefined) {
        args[c++] = {'name':name,'value':g.user[name]};
      }
    }
  }

  if(args.length!=0) {
    body = '';
    for(i=0,x=args.length;i<x;i++) {
      if(i!==0) {
        body +='&'
      }
      body += args[i].name+'='+encodeURIComponent(args[i].value);
```

```
    }
    alert(body);
    nextStep('post-user',url,body);
  }
  else {
    alert(g.errors.registerFormError);
  }
}
```

There are a number of other processing routines in the working example. See for "Source Code" on page 217 details on how to locate and download the full source for this book.

Support routines. This implementation contains a few routines used to support the high-level processing outlined in previous sections. For example, below is a function to parse the query string of the current document and a function used to assemble the URI for the next request in the task chain:

```
function getArg(name) {
  var match = RegExp('[?&]' + name + '=([^&]*)').exec(window.location.search);
  return match && decodeURIComponent(match[1].replace(/\+/g, ' '));
}

function nextStep(status,url,body) {
  var href,adr;

  href = window.location.href;
  href = href.substring(0,href.indexOf('?'));
  adr = href + '?status=' + status;
  adr += '&idx=' + g.idx;
  if(url) {adr += '&url=' + encodeURIComponent(url);}
  if(body) {adr += '&body=' + encodeURIComponent(body);}

  window.location.href = adr;
}
```

This last routine handles parsing a response representation to find elements that match a specific link-relation value. This is very similar to looking for elements with specific values in the `class` attribute:

```
function getElementsByRelType(relType, tag, elm)
{
  var testClass = new RegExp("(^|\\s)" + relType + "(\\s|$)");
  var tag = tag || "*";
  var elements = (tag == "*" && elm.all)? elm.all : elm.getElementsByTagName(tag);
  var returnElements = [];
  var current;
  var length = elements.length;
  for(var i=0; i<length; i++){
    current = elements[i];
    if(testClass.test(current.getAttribute('rel'))){
      returnElements.push(current);
    }
  }
```

```
    return returnElements;
}
```

Summary

This chapter covered the topic of using HTML5 as a base media type. This has the advantages of a fully-functional hypermedia type (supports almost all the H-Factors identified in "Identifying Hypermedia : H-Factors" on page 13). Using HTML5 also means that implementing the basic server will result in a working client implementation that can run in common web browsers without the need for client-side scripting. This is a great way to build quick sample implementations to test server transition details and media type design aspects.

Since HTML5 is a domain-agnostic media type, it has no built-in domain specific markup elements and no predefined transitions as does Atom/AtomPub and the Collection+JSON example in Chapter 3. This example showed how designers can use HTML5 attributes (`id`, `name`, `class`, and `rel`) to apply domain-specific details to responses.

A server implementation of a simple CouchDB data model was created that included support for HTTP Basic Authentication. This showed that user authentication details can be implemented independently of the actual media type design. And finally, two sample clients were reviewed. The first was simply a CSS restyling of the plain HTML5 rendered by the server (the POSH client). The second client was an Ajax-style web bot implementation that parsed the HTML5 responses looking for desired links and forms in order to post messages to the server.

This implementation showed that it is possible to use an already-existing hypermedia type as the basis for your own unique design. It also showed the importance of documenting domain-specific details in ways that both server and client implementors can understand. The next (and final) chapter explores the role of documentation in more detail.

CHAPTER 5
Documenting Hypermedia

We think in generalities, but we live in detail.

- Alfred North Whitehead

Documenting and publishing hypermedia designs makes it possible for the design to gain wider adoption. The process of publishing also can mean attracting expert review that will result in an improved, possibly more useful design. There are a number of organizations that may become involved in the review process including the IANA, W3C, IETF, etc.

This chapter covers a number of the mechanical details of documenting, publishing, and registering media type designs and link relation types. First, a standard for documenting requirements and compliance levels based on the guidelines in RFC 2119 is covered.

Next, the details of writing solid documentation for media type designs of various formats (XML, JSON, HTML) are reviewed. This includes the process of recording the mapping of domain-specific information to media types.

The difference between extending and versioning media types is covered along with the steps for registering media types and link relations with various standards bodies. Finally, a set of design and documentation tips are provided.

Requirements, Compliance, and RFC 2119

When documenting media types, you often need to indicate both to the author and consumers of that type, which elements, if any, are required in a representation (which are optional, etc.). Expressing these requirement levels in a manner that readers can easily understand will go a long way toward making your documentation easy to read and in turn, making your media type easier to implement.

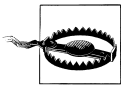
The IETF (Internet Engineering Task Force), responsible for many of the Internet standards documents in use today, recognized the need for a consistent method for

communicating requirements levels. To that end, the IETF published "Key words for use in RFCs to Indicate Requirement Levels" (RFC 2119). This document outlines several keywords for use in indicating levels of requirement. The document also sets a standard for indicating compliance. These keywords, combined with the compliance standard, comprise an excellent set of tools for use in documenting media types, too.

The RFC 2119 Keywords

The keywords defined in RFC 2119 can be used to indicate requirements levels from absolutely required to recommended to optional. It also has standards for indicating negative requirements such as "must not" and other similar phrases. Since these keywords and phrases are already defined by the IETF and their meaning is well-established, they are an excellent tool for documenting your media type. Using these keywords throughout the media type definition document provides clear directions to both server and client developers.

Below are the words and phrases defined in RFC 2119 along with the meaning associated:



The established method for using the RFC 2119 keywords is to represent them in ALL CAPS. This makes it clear to the reader that the documentation is invoking the RFC 2119 meaning of that word. It is also important that you do not attempt to redefine the meaning of RFC 2119 keywords. If you want to indicate some other level of requirement for your documentation, you need to establish your own keywords with their own meaning and use.

MUST, SHALL, REQUIRED
: The referenced feature or element of the specification is absolutely required.

SHOULD, RECOMMENDED
: There may be valid reasons for not implementing/support the referenced feature or element of the specification, but implementors should consider carefully before deciding not to support this feature/element.

MAY, OPTIONAL
: The referenced feature or element of the specification is optional; not all implementations should be expected to support it. Implementations should continue to function in cases where this feature/element is missing.

MUST NOT, SHALL NOT
: The referenced feature or element is absolutely prohibited.

SHOULD NOT, NOT RECOMMENDED
: There may be valid reasons for implementing/supporting the referenced feature or element of the specification, but implementors should consider carefully before supporting this feature/element.

Sample Documentation Using RFC 2119 Keywords

Once you understand the use and meaning of the RFC 2119 keywords, it is easy to put them into practice in your own documentation. Below are a few simple examples:

> All valid response representations MUST begin with `<root>` as the first element.
>
> Servers SHOULD return a response code of 204 if the HTTP DELETE request was successful.
>
> Clients MAY include the `Accept"` HTTP header when sending a request to the server.
>
> The "userid" is a REQUIRED.
>
> Clients MUST NOT store responses returned via HTTPS.
>
> Servers SHOULD NOT include the user's secret key in clear-text responses.

Defining Compliance

One of the key benefits of using the RFC 2119 keywords is to clearly establish compliance rules for implementors of your media types. Typically this is done by including text that: 1) indicates the use of RFC 2119 keywords in your document and, 2) tells implementors how to achieve levels of compliance such as:

- Unconditionally Compliant
- Conditionally Compliant
- Non-Compliant

Below is a common text block (typically appearing early in the document) to include in your documentation to tell readers that RFC 2119 words are used within the text:

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

In addition to the text above, authors can include a paragraph to indicate levels of compliance. This helps client and server developers know what it takes to create "unconditionally compliant" or "conditionally compliant" implementations. It also helps parties recognize "non-compliant" implementations and make necessary changes. Below is sample text that indicates compliance levels:

> An implementation is "non-compliant" if it fails to satisfy one or more of the MUST or REQUIRED level requirements. An implementation that satisfies all of the MUST or REQUIRED and all of the SHOULD level requirements is said to be "unconditionally compliant"; an implementation that satisfies all of the MUST level requirements but not all of the SHOULD level requirements is said to be "conditionally compliant."

By combining the use of RFC 2119 keywords with a clear statement on compliance levels, developers will be able to interpret the documentation of a media type design and create implementations that have a high probability of interoperability with other independent implementations of the same media type.

Documenting Media Type Designs

One of the key aspects of designing media types is producing useful documentation for them. In hypermedia designs, it is essential that both client and server have a clear understanding of the media type and how it can be used in a target implementation. Consistent, readable, and clear documentation is always the goal. When supporting hypermedia, it is vital.

This section provides guidance on a general layout for media type documentation and covers format-specific considerations for the most common data formats including XML, JSON, and HTML. In addition, information on how to document domain-specific aspects of your design as well as publishing your documentation are covered.

The IETF, W3C, and IANA

The document format described in this chapter is a hybrid of features and requirements taken from three different sources: the Internet Engineering Task Force (IETF), the World Wide Web Consortium (W3C), and the Internet Assigned Numbers Authority (IANA). Each of these organizations have their own rules and requirements for publishing specifications.

The IETF process for publishing specifications starts with creating an I-D (Internet Draft) and shepherding that document through a process of review that eventually may be published as an RFC (Request For Comment) document. The IETF maintains the "Guidelines to Authors of Internet Drafts," which provides details on the authoring I-Ds. The document "The Internet Standards Process—Revision 3" (RFC 2026) describes the IETF approval process itself.

The W3C provides the "QA Framework: Specification Guidelines" document to help authors create and submit specifications for review. Their "World Wide Web Consortium Process Document" provides a full description of "the organizational structure of the W3C and the processes related to the responsibilities and functions they exercise to enable W3C to accomplish its mission."

Finally, all media type designs should be registered with the IANA. See "Registering Media Types and Link Relations" on page 157 for more on the document format and process of registering media types and link relations.

General Layout

No matter what style of media type you design, regardless of the data formats involved, it is a good idea to use a consistent layout for all of your documentation, including any support documents, examples, errata, and other materials. A consistent presentation makes it easy for readers to find what they are looking for. It also makes it easy for readers to spot updates, modifications, and extensions that may occur over time.

It is also important to keep in mind that media type documentation can have multiple audiences. Some readers will be primarily interested in how to use the media type to express domain-specific information. Typically these authors are not focused on how to implement a client or server, but they are more interested in whether the media type can successfully carry the domain information needed to make an application work. Some readers may be focused on creating working implementations. Server implementors will want to know how to use the media type to emit valid representations. Client developers will want to understand how to interpret representations received from servers. A small but important group of readers will want to make sure the media type design is safe to use, stable, and reliable. All of these readers need to be able to find what they are looking for within the media type documents.

Below are four suggested top-level sections that can make media type design documentation easy to maintain and easy to read.

Front matter

The front matter of a media type design should contain basic information that helps readers get a quick understanding of the aims, status, and adoption of the media type. One way to accomplish these goals is to include the following details:

Description
: Provide a short, clear description of the media type and the general design emphasis, problem domain, or other related information. This should be only a couple sentences in length and should help the reader know if this media type design might meet the reader's needs. For example:

> "Collection+JSON is a JSON-based read/write hypermedia-type designed to support management and querying of simple collections. It is similar to the The Atom Syndication Format (RFC 4287) and The Atom Publishing Protocol (RFC 5023). However, Collection+JSON defines both the format and the semantics in a single media type. It also includes support for Query Templates and expanded write support through the use of a Write Template."

Registration Status
: Currently, HTTP relies on the MIME media type standard for identifying media type designs. As of this writing, the IANA is the sole registration point for MIME media types. Your document should announce the registration status of your design by listing the MIME media type string and an indication of the media type's current registration status. Here are some possible examples:

- `application/vnd.collection+json` (unregistered)
- `application/vnd.collection+json` (application pending)
- `application/vnd.collection+json` (approved)

See "Registering Media Types and Link Relations" on page 157 for more on registering media type designs.

Known Implementations
: It is always helpful for readers to know there are other implementations of the media type design already in place. Adding references to known implementations not only provides readers with possible examples to reference, it also acts as an indicator of the adoption level of the design. This section need only list implementations and provide links to external references including client or server implementations, available libraries that support the design, etc. However, if the design has been formally adopted by a standards body, company, or established developer group adding these references can help readers further evaluate the design.

Update History
: All designs undergo changes over time. Whether it is the initial series of updates before reaching "stable release" or continued improvement and extension as the design reaches a wider audience, it is important to keep a good history of the media type's modifications. Readers will be able to see whether the media type is undergoing frequent modification, has reached a stable level, or has not been updated in quite a while. This, too, is valuable information for those assessing the applicability of the media type design for their needs.

Along with the four elements listed above, the front matter should include links to the following three sections:

Format
: The detailed description of the media type design.

Examples
: A set of stand-alone examples of the media type including sample representations used for requests and responses.

Tutorials
: A section that provides one or more walk-through narratives that cover common use cases for the media type design. This can be used to show readers a more concrete demonstration of how the media type can be used to complete typical tasks, solve familiar problems, etc.

Format

The Format section contains the details of the media type design itself. This includes a full description of all the elements, attributes, objects, properties, etc. that can appear within a valid representation. The exact layout and contents of this section depends on the data format (XML, JSON, etc.) and domain style (Specific, Generic, Agnostic) of the design.

This section should also contain a list of all predefined enumerated values that may be used in the document. Below is an example that shows the valid values for the `method` attribute of a hypermedia control:

```
Sending data to the server:

Example:
<send method="{method-value}" ... />

Where {method-value} is one of the following:
- GET
- POST
- PUT
- DELETE
```

The Format section should also include documentation on any Link Relation values used for the media type. These may include references to already-defined values, values specific to this design, and references to other possible sources for Link Relation values.

A list of supported data types should also be provided in this section. These may be references to existing data standards such as XML Types or unique definitions of data types native to the design (i.e. integer, float, alphanumeric, etc.).

Finally, when appropriate, the Format section should include information on whether the design is extensible and/or subject to versioning. If extensions are allowed, the section can include information on how extensions are created, registered (if appropriate), and how versions can be detected in responses.

Examples

The Examples section should contain a set of stand-alone interactions between client and server that illustrate typical request and response patterns. The goal in this section of the documentation should be to help readers quickly locate a common interaction and easily see how the client and server can interact to complete the conversation.

Usually the collection of examples reads like a list of well-known use cases for the media type. For example, if the media type is domain-specific and designed to manage a simple accounting ledger, some of the examples might be:

- Creating a chart of accounts
- Getting a list of the accounts
- Adding a new account
- Modifying an existing account
- Deleting an existing account
- Getting a list of customers
- Adding a new customer

In cases where the media type design is domain-agnostic, you can still list common use cases even if they are more general:

- Creating a new list
- Getting the list of items
- Adding a new item to the list
- Editing an existing item in the list
- Removing an item from the list
- Querying the list of items

Note that each of the examples above can stand alone. Developers interested in a particular task should be able to scan the list of examples and find one or more items that fill their needs. While the order of the items is not all that important, grouping them by task or alphabetically is usually the best option.

The actual content of each example can be quite simple. Often a sample client request followed by a server response is all that is needed in order to get the point across:

> **Getting a List of Mazes Using the Maze+XML Media Type**
>
> Clients can request a list of available mazes by sending an HTTP GET request to the server using the "Collection URI":
>
> ```
> *** REQUEST
> GET /mazes/ HTTP/1.1
> Host: www.example.org
> Accept: application/vnd.amundsen.maze+xml
> ```
>
> Servers SHOULD respond to valid requests to a "Collection URI" with a collection document. Below is a simple example:
>
> ```
> *** RESPONSE
> HTTP/1.1 200 OK
> Content-Type: application/vnd.amundsen.maze+xml
> Content-Length: XXX
>
> <maze>
>   <collection href="http://www.example.org/mazes/">
>     <link href="http://www.example.org/mazes/1" rel="maze" id="m3"/>
>     <link href="http://www.example.org/mazes/2" rel="maze" id="m2"/>
>     <link href="http://www.example.org/mazes/3" rel="maze" id="m3"/>
>   </collection>
> </maze>
> ```

Tutorials

Unlike the short, to-the-point style of the examples section, the tutorials section should provide a more detailed explanation of how implementors can use the media type design. Usually this involves showing working code for client and/or server as well as possible request/response representations like those that appear in the examples section. If the media type design is not domain-specific, details on how to apply application-domain information to media type representations should also be included. Ideally, the tutorial section of the documentation should walk a developer through the process of expressing a problem domain with the media type, illustrate sample server code that converts private component data into valid media type representations, and

show sample client code that consumes the server representation and properly parses and renders the responses.

Adding to the Maze Collection

Servers may allow users to add new mazes to the collection. If this is true, the URI returned in the `<collection>` element SHOULD be used. The exact details of how to add a new maze to the collection is beyond the scope of this document. Servers may define their own required and optional parameters, select which media types can be used to send data to the server, and which application-level protocol(s) can be used (HTTP, XMPP, etc.).

Below is an example showing a server that supports the HTTP POST method and the application/`x-www-form-urlencoded` media type to send a single parameter (size). In this case the server will generate a simple connected square maze where each side has a maximum number of cells equal to the parameter value:

```
*** REQUEST
POST /mazes/ HTTP/1.1
Host: www.example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: XXX

size=10

*** RESPONSE
HTTP/1.1 201 Created
Location: http://www.example.org/mazes/1
```

Using HTTP POST in the manner shown here is not required. Servers may, for example, support HTTP PUT, FTP STOR, etc. to add new mazes to the collection, may support a wide range of parameters, and may even allow clients to upload fully defined mazes using any number of media types (JSON, HTML, CSV, etc.).

Documenting XML Designs

Documenting media type designs that use XML as the native format should include the following sections:

See Appendix C for a complete example of documentation for XML-based media type design.

Elements

The list of XML elements valid for this media type including any possible attributes and child elements.

The `<cell>` Element

The `<cell>` element represents a single cell or block in the maze. The `<cell>` element SHOULD have two attributes: `href` and `rel`. The `href` attribute MUST contain a valid URI. The `rel` attribute SHOULD be set to "current". The `<cell>` element MAY have one or more of the following attributes: `debug`, `total`, and `size`. The `<cell>` element MAY have one or more `<link>` child elements.

Attributes

The list of possible attributes and to which elements they may be assigned. If the attribute supports an enumerated list of possible values, these values should be included, too.

The `href` Attribute

This attribute specifies the location of a Maze+XML resource, thus defining a link between the current resource and the destination resource defined by this attribute. The value of the attribute MUST always be a valid URI. This is an OPTIONAL attribute for the following elements: `<collection>`, `<item>`, `<cell>`, and `<error>`. This is a REQUIRED attribute for the `<link>` element.

Link Relations

The list of possible link relation values used in this media type. If the design supports the use of author-defined link relation values or other values defined in other documents, this information should be included.

The current Link Relation

Refers to a resource containing the most recent item(s) in a collection of resources. Registered through IANA Link Relations. When used in the Maze+XML media type, the associated URI returns the client's current position in the active maze.

Data Types

The list of data types (their names and valid values) used in the design. Typically this can be a reference to existing standards documents.

The NUMBER Data Type

Numbers MUST contain at least one digit ([0-9]). The characters ".", "-", and "+" MAY also appear.

The URI Data Type

URI is defined by RFC 3986.

Documenting JSON Designs

Documenting media type designs that use JSON as the base format should include the following sections:

See Appendix D for a complete example of documenting a JSON-based media type design.

Objects

The list of JSON objects valid for this media type including any possible child objects, arrays, and properties.

The template Object

The template object contains all of the input elements used to add or edit collection records. This is an OPTIONAL object and there MUST NOT be more than one template object in a Collection+JSON document. It is a top-level document property. The template object SHOULD have a data array child property.

```
// sample template object
{
  "template" :
  {
    "data" : [ARRAY]
  }
}
```

Arrays

The list of possible JSON arrays that may appear in a representation including any child objects, arrays, and properties.

The data Array

The data array is a child property of the items array and the template object. The data array SHOULD contain one or more anonymous objects. Each object MAY have any of three possible properties: name (REQUIRED), value (OPTIONAL), and prompt (OPTIONAL).

```
// sample data array
{
  "template" :
  {
    "data" :
    [
      {"prompt" : STRING, "name" : STRING, "value" : VALUE},
      {"prompt" : STRING, "name" : STRING, "value" : VALUE},
      ...
      {"prompt" : STRING, "name" : STRING, "value" : VALUE}
    ]
  }
}
```

Properties

A listing of all the possible properties that are valid for the media type design. If the property supports an enumerated list of values, these values should be supplied as well.

The render Property

The render property MAY be a child of the links array element. It SHOULD be a STRING data type. The value MUST be either "image" or "link". If the render property does not appear on a links array element, it should be assumed to be set to "link".

Link Relations

The list of possible link relation values used in this media type. If the design supports the use of author-defined link relation values or other values defined in other documents, this information should be included.

The collection Link Relation

The target IRI points to a resource that represents a list of which the context IRI is a member. When used in the Collection+JSON media type, this link relation value refers to a collection document. Logged with the Microformats Existing Rel Values.

Data Types

The list of data types (their names and valid values) used in the design. Typically this can be a reference to existing standards documents.

The OBJECT Data Type

An OBJECT structure is represented as a pair of curly brackets surrounding zero or more name/value pairs (or members). A name is a string. A single colon comes after each name, separating the name from the value. A single comma separates a value from a following name. The names within an object SHOULD be unique.

Documenting HTML Designs

Unlike media types based on XML, JSON, or other formats that support domain-specific designs, HTML-based media types are, by definition, domain-agnostic and already have their own set of native hypermedia controls. For this reason, documenting HTML-based media type designs focuses almost exclusively on using a limited set of existing attributes to apply domain-specific information to representations.

See Appendix E for a complete example of documenting an HTML-based media type design.

Documentation for HTML-based media type designs should include the following sections:

Class Attribute Values

In HTML designs, the `class` attribute can be used to mark data elements in a response (`div`, `p`, `span`, etc.). This includes indicating possible child elements and whether they are optional or required.

The message Class

Applied to an LI tag. A representation of a single message. It SHOULD contain the following descendant elements:

- SPAN.class="user-text"
- A.rel="user"
- SPAN.class="message-text

- A.rel="message"

It MAY also contain the following descendant elements:

- IMG.class="user-image"
- SPAN.class="date-time"

The `class` attribute can also be used to identify state transition elements (`form`, `a`, etc.).

The user-search Class

Applied to a FORM tag. A link template to search all of the users. The element MUST be set to FORM.method="get" and SHOULD contain the descendant element: INPUT[text].name="search".

Name Attribute Values

The `name` attribute is used in HTML-based designs to indicate values used in state transitions. These are usually applied to `input`, `textarea`, and `select` elements.

user-image

Applied to an INPUT[file] element. The image for the user.

website

Applied to an INPUT[text]. The URL of a website associated with the user profile. When supplied, it SHOULD be valid per RFC 3986.

ID Attribute Values

The `id` attribute can be used to identify a unique element or block within the representation. Often this is handy for marking sections of the representation that only appear once within the response.

The queries ID

Applied to a DIV tag. The list of valid queries in this representation. MAY have one or more FORM and/or A descendant elements (see "rel" and "class" section for details).

Link Relation Values

Link relation values are used to mark HTML.A tags with identifiers that can be used by user agents to render and/or activate in order to initiate a state transition. Some transitions may actually point to HTML.FORM elements that can be used to perform a query or update data on the server.

message

Applied to an A tag. A reference to a message representation.

message-post

Applied to an A tag. A reference to the message-post FORM.

messages-search

Applied to an A tag. A reference to the messages-search FORM.

Documenting Application Domain Specifics

Most media type designs (that are not domain-specific) require additional documentation of application domain specifics. For example, implementing a to-do application using HTML will require designers to mark elements in the response as key data fields (task name, due date, description, status, etc.). Since HTML does not have native markup elements with these names, existing elements need to be annotated or decorated with the application-specific information.

As was shown in "Expressing Application Domain Semantics in HTML5" on page 96, HTML has several attributes that are used to hold application-specific information (`id`, `name`, `class`, and `rel`). Other media type designs that are not domain-specific will have similar features. For example, the Collection+JSON media type design introduced in Chapter 3, provides the `name` and `rel` properties for application-specific information.

Data elements

Media type designs need to support identifying domain-specific data elements in representations. For example, a to-do application will have a set of domain-specific data points such as "title", "description", "due-date", and "status", etc.

The examples given here are just that, examples. The key point is to make sure developers have clear documentation mapping domain-specific information to existing elements in the media type design. When the design is domain-specific, this is relatively easy. When the design is domain-generic, additional text is needed to complete the mapping.

Domain-specific data elements. Media type designs that are domain-specific will have elements with these same names:

```
Expressing data elements in a domain-specific design

<!-- an XML example -->
<task>
  <title>Sample Task</title>
  <description>This is a sample task item.</description>
  <due-date>2012-03-01</due-date>
  <status>completed</status>
</task>

/* a JSON example */
{
  "task" :
  {
    "title" : "Simple Task",
    "description" : "THis is a simple task item.",
    "due-date" : "2012-03-01",
    "status" : "completed"
  }
}
```

Using the above example as a guide, media type documentation would list the elements by name and provide descriptive text regarding the meaning and use of the element.

> The `item` element's name property should be set to the value "task". The `item` element should have four data child elements. Their name properties should be set to "title", "description", "due-date", and "status". The value properties of the "title" and "description" `data` elements MUST contain a valid `xsd:string` value. The value properties of the "due-date" `data` element MUST contain a valid `xsd:dateTime` value. The value property of the "status" `data` element MUST contain either the string "completed" or "pending".

Domain-generic data elements. Media type designs that are not domain-specific will need to support other ways for authors to indicate domain-specific information. Typically this is done using attributes to decorate XML-style media types or, in other formats, designated properties.

Expressing data elements in a domain-generic design

```
<!-- an XML example -->
<item name="task">
  <data name="title">Sample Task</data>
  <data name="description">This is a sample task item.</data>
  <data name="due-date">2012-03-01</data>
  <data name="status">completed</data>
</item>

/* a JSON example */
{
  "item" :
  {
    "name" : "task",
    "data" [
      {"name" : "title", "value" : "Sample Task"},
      {"name" : "description", "value" : "This is a simple task."},
      {"name" : "due-date", "value" : "2011-03-01"},
      {"name" : "status", "value" : "completed"}
    ]
  }
}
```

In this second example, the documentation of the media type will be separate from the documentation of the application-domain information. Typically, developers will rely on two sets of documentation. First, the documentation of the media type itself and second, a document that matches the domain-specific data elements with the domain-generic media type elements.

> The `task` element has four possible child elements (`title`, `description`, `due-date`, and `status`). The `title` and `description` fields MUST contain a valid `xsd:string` value. The `due-date` field MUST contain a valid `xsd:dateTime` value. The `status` field MUST contain either the string "completed" or "pending".

Hypermedia affordances

Along with expressing domain-specific data in representations, responses from the server need to include hypermedia details that describe the possible transitions. These are sometimes referred to as hypermedia "affordances" or simply hypermedia controls. The hypermedia controls need to communicate the basic H-Factors described in "Identifying Hypermedia : H-Factors" on page 13 (LE, LO, LT, LN, and LI). In domain-specific designs, hypermedia controls will have very clear names and can be mapped directly to the protocol-level actions associated with them.

Just as in the handling of domain data elements, the style of the media type design dictates the methods used to apply domain-specific information to hypermedia controls in the representation.

Domain-specific hypermedia controls. In domain-specific designs, the hypermedia controls have names that relate directly to application-domain information:

```
Expressing hypermedia controls in domain-specific designs

<!-- an XML example -->
<create href="http://example.org/tasks/">
  <title>Sample Task</title>
  <description>This is a sample task item.</description>
  <due-date>2012-03-01</due-date>
  <status>pending</status>
</create>
...
<task-list href="http://example.org/tasks/">
...
<delete href="http://example.org/tasks/1">

/* a JSON example */
{
  "create" :
  {
    "href" : "http://example.org/tasks/"
    "title" : "Simple Task",
    "description" : "THis is a simple task item.",
    "due-date" : "2012-03-01",
    "status" : "pending"

  }
}
...
{"task-list" : "http://example.org/tasks/"}
...
{"delete" : "http://example.org.tasks/1"}
```

Media type documentation for domain-specific hypermedia controls needs to include details on which protocol methods should be used with each control. In the previous example, the documentation might read as follows:

> The `create` element is used to indicate clients should create a body representation using the `application/x-www-form-urlencoded` media type where the child element name and the child element value are converted to name-value pairs in the form `{name}={value}` followed by the `&` character to separate consecutive pairs. The resulting representation should be sent via the `POST` method (for HTTP) to the URI supplied by the `href` property of the create element.

Domain-generic hypermedia controls. In media type designs that are not domain-specific, the hypermedia controls are usually identified with the protocol-level action they support:

```
Expressing hypermedia in domain-generic designs

/* a JSON example */
{
  "send" :
  {
    "action" : "ceate",
    "href" : "http://example.org/tasks/"
    "data" [
      {"name" : "title", "value" : "Sample Task"},
      {"name" : "description", "value" : "This is a simple task."},
      {"name" : "due-date", "value" : "2011-03-01"},
      {"name" : "status", "value" : "completed"}
    ]
  }
}
...
{"link" : {"action" : "read", href" : "http://example.org/tasks/", "rel" : "tasks"}}
...
{"link" : {"action" : "remove", "href" : "http://example.org/tasks/1", "rel" : "edit"}}

<!-- an HTML example -->
<form method="post" action="http://example.org/tasks/" class="tasks">
  <input name="title" value="Sample Task" />
  <textarea name="description">This is a simple task.</textarea>
  <input name="due-date" value="2011-03-01" />
  <select name="status">
    <option>pending</option>
    <option selected="true">completed</option>
  </select>
</form>
...
<a href="http://example.org/tasks/" rel="tasks">Tasks</a>
```

As can be seen from the above examples, domain-generic hypermedia controls need to support the use of properties or attributes that associate the controls with domain-specific information. Usually this is in the form of `rel`, `name`, and other similar names. Documentation for domain-generic hypermedia controls also reflects this approach:

> The `send` element with the `action` property set to "create" can be used to create new tasks on the server. This element should have four child `data` elements ("title", "description", "due-date", and "status"). Client implementations should use the data elements to create a valid JSON payload and send that payload to the URI contained in the `href` property using the HTTP POST method.

Publishing Media Type Designs

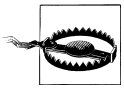
Media type designs should be published online at a stable address. This means the URL for the documentation should be one that will be supported for a long time. For example: `http://www.example.org/media-types/maze`. If the document must be moved, a permanent redirect (`301 Moved Permanently`) should be associated with the stable address so that existing bookmarks and written documentation will still be able to locate the current version of the documentation.

If, over time, multiple editions of the design documents are produced (e.g. "Maze+XML 1.0", "Maze+XML 1.1", etc.). The stable address should respond with a general document that lists the current version, all historical versions, and allows users to navigate to the document of their choice.

If the media type design will be registered with the IANA or some other party, a stable address will be required. It is a good idea to select a location over which the designer(s) will have control for the foreseeable future.

Extending and Versioning Media Types

No matter how much effort is put into the initial media type design, it is quite likely that not all possible uses are anticipated, not all flaws in the design are expunged. Even if the initial design is solid, it is possible that the media type will become so widely used that new and inventive use cases will emerge for the media type. In these cases, it is important to have a clear understanding of the possible ways in which an already-deployed media type design can be improved, corrected, and expanded to meet the needs of the developer community.



In a distributed network like the WWW, it is possible that developers will implement clients and servers based on a registered media type design without ever consulting or informing the media type designer. For this reason, once a design has been registered and made available to the public, it is important that media type designers take special care when modifying its design.

This section looks at two opposing approaches to updating an existing media type: extending and versioning. Extending a media type design works when the aim is to introduce changes in way that is compatible with already existing implementations. Versioning a media type can be used when it is important to break or invalidate any existing implementations.

Extending

The word "extend" comes from the Latin *pandere* or "stretch." The word "expand" has the same etymology. Essentially, extending a media type design means adding new features or elements: expanding or stretching the media type. For the purposes of this text, extending a media type design means adding features or functionality in a way that does not break existing client or server implementations. Whenever possible, media type designers should use the technique of extending to make modifications to a media type design.

Extending a media type design means supporting compatibility. In other words, changes in the media type can be accomplished without causing crashes or misbehavior in existing client or server implementations. There are two forms of compatibility to consider when extending a media type: forward and backward.

Forward compatibility

An extension can be considered "forward compatible" if the changes do not break existing implementations. This means existing client applications will continue to work successfully when they receive a representation that contains the new features. Servers will continue to function properly when they receive a representation that contains the new features. Usually, this means the new features/functionality can be safely ignored by implementations that do not recognize them.

Forward-compatible design changes are ones that are safe to add to the media type without adversely affecting previously existing implementations.

Backward compatibility

Backward-compatible changes are those that allow any future implementations to consume older versions of the design. For example, if a client that supports one or more of the new extensions receives a representation from a server that does not support these new extensions, the client implementation will continue to function successfully. Usually, this means the new extensions do not create a required dependency that causes implementations to break if that extension is missing.

Backward-compatible changes are ones that are safe to add to the media type design without adversely affecting future implementations.

Often backward-compatible changes mean that implementations need to be prepared to handle representations that are missing expected elements. This may mean that some

functionality or feature is not available to an implementation and that implementation must work around that missing element or fall back to a mode that supports an older design of the media type.

Rules for extending media types

In order to support both forward and backward compatibility, there are some general guidelines that should be followed when making changes to media type designs.

Existing design elements cannot be removed. If the media type has established that a feature/element may exist in representations, then an extension cannot require that the element not appear in representations. For example, if the media type requires that every representation start with the `<root>` element, designers cannot remove this rule. Or, if the media type defines an optional `{"revision" :{...}}` object, future extensions of the media type cannot require implementations to reject representations that contain that element.

The meaning or processing of existing elements cannot be changed. Once a feature/element is published as part of the media type design, the meaning associated with the element or the processing rules for the element cannot be changed. Or, if the media type defines the `{"status" :{...}}` element as a representation of "the status of the current document," extensions cannot change the meaning of that element to "the status of the current logged-in user," etc. Or, if the design states that the `<suggested-name>` element in client representations will be used by servers when establishing new resource identifiers, a media type extension cannot require servers to ignore this element and not process it as was originally described.

New design elements must be treated as optional. Any new representation elements (including the processing of those elements) must be defined as optional. For example, if the media type extension supports adding a `{"concurrency-check : "..."}` client representations sent to the server, the extension cannot make that value a required element for all client-initiated representations.

Versioning

The word "version" has its roots in the Latin *vertere* ("to turn") and its derivation *versus*. One dictionary definition of the word "version" is "a particular form or variation of an earlier or original type." For the purposes of this text, "versioning" a media type means creating a new variation on the original, a new media type.

Versioning a media type means making changes to the media type that will likely cause existing implementations of the original media type to "break" or misbehave in some significant way. Designers should only resort to versioning when there is no possible way to extend the media type design in order to achieve the required feature or functionality goals. Versioning should be seen as a last resort.

There may be cases where it is desirable to introduce a new version of a media type. For example, a designer might want to invalidate all previous implementations of a media type in order to overcome a serious flaw in the original design or as a way of obsoleting existing implementations for some other reason.

Deciding when to version a media type

Any change to the design of a media type that does not meet the requirements previously described in are indications that a new version of the media type is needed. Examples of these changes are:

A change that alters the meaning or functionality of an existing feature or element.
: For example, the meaning of the `<id>` element of a media type must be changed from the identifier of the document to the identifier of the version of the document.

A change that causes an existing element to disappear or become disallowed.
: For example, the `{"user-secret-key" : "..."}` property of the existing media type is now absolutely prohibited in representations sent from the client to the server.

A change that converts an optional element into a required element
: For example, the `<input name="document-tag" value="..." />` in representations from the client has changed from OPTIONAL to REQUIRED.

If the media type design must be changed in these ways, a new version of the media type should be declared.

Rules for versioning a media type

While versioning a media type should be seen as a last resort, there are times when it is necessary. The following guidelines can help when creating a new version of a media type.

It should be easy to identify new versions of a media type. When the version of a media type changes, it should be easy for implementors, both client and server, to recognize the change. There are a number of ways in which this can be done.

The most explicit option is to register a new MIME Media Type identifier with the IANA:

```
application/vnd.custom+xml
application/vnd.custom-v2+xml
```

A similar approach is to define and use a version parameter in your MIME media type registration:

```
application/custom+JSON;version=1
application/custom+JSON;version=2
```

RFC 4229 (HTTP Header Field Registrations) lists the `Version` header. This header is described in a 1992 W3C online (Object Headers) as:

> "[A] string defining the version of an evolving object. Its format is currently undefined, and so it should be treated as opaque to the reader, defined by the information provider. The version field is used to indicate evolution along a single path of a particular work."

```
*** REQUEST ***
PUT /users/1 HTTP/1.1
Host: www.example.org
Content-Type: application/vnd.custom+xml
Length:xxx
Version: 2
...

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.custom+xml
Length:xxx
Version: 2
...
```

It is also possible to place version information directly in the representation body:

```
<custom version="2">
...
</custom>

{"custom" :
  "version" : "2",
  {"content" : {...}}
}
```

While the exact method of communicating versions in media type representations is a design choice, it should be noted that the MIME media type identifier option is likely to be the most easily recognizable and consistently supported option. It is possible that intermediaries such as caches and proxies may ignore or strip out the `Version` header and intermediaries will not be able to see the version information carried in the representation body.

These last two points can have implications on the caching of representations. Imagine a case where there are two servers, each supporting a different version of the media type, each presenting the same representations (i.e. some public data shared among servers). It is possible that caching intermediaries can receive multiple media type versions of the same representation. If the version information is not recognized (in the case of the header) or unavailable (in the case of the representation body), it is possible that caches will overwrite that representation and/or deliver the wrong version of the representation to clients.

Implementations should reject unsupported versions. Any client or server that does not explicitly support a particular version of a media type should reject the representation and issue an appropriate message. When using the HTTP protocol, servers can report status code 415 (Media Type Not Supported) when that server receives a representation that is not a supported version. Clients should inspect the version of the response representation and reject any unsupported versions.

In order to work properly, support for versioning needs to be coded into both clients and servers ahead of time. In other words, clients and servers must check for unsupported versions of the media type upon receipt of every single representation and be prepared to respond accordingly.

Registering Media Types and Link Relations

While registering media type designs and unique link relations is not required, it is very valuable and should be encouraged. First, the process of registering designs with organizations such as the IANA, IEFT, and W3C usually involves some level of peer review. This can help improve the design and is likely to promote uniformity and consistency in designs in general.

Also, registering designs is an excellent way to publish work and gain adoption. Some implementors will use registration as a measure of the value of the design and may favor registered designs over those that are not.

There are a number of ways in which to register media types and link relations. This text covers interactions with three organizations: the Microformats community, the IANA, and the IETF.

Media Types

Internet RFC 4288 (Media Type Specifications and Registration Procedures) outlines the details of media type registration and review. Currently the IANA is the organization charged with handling the registration of media types.

A complete list of the currently registered media types can be found at the IANA MIME Media Types page.

The RFC document also identifies a categorization of registrations called "Registration Trees." The procedure for two of the categories ("Vendor" and "Personal") are relatively easy and brief. The procedure for the "Standard" category is more involved.

Vendor/Personal tree media type registrations

Registering a media type with the IANA under the Vendor and Personal categories (called trees in RFC 4288) requires the following:

Publish the media type design
: This includes the process of completing the documentation and publishing that documentation at a stable address (see "Documenting Media Type Designs" on page 138).

Submit the completed IANA media type registration template
: The IANA requires all media type registrations via a standard form. The IANA conveniently supplied an online form at their website (*http://www.iana.org/cgi-bin/mediatypes.pl*) that designers can use to complete the registration document (see Appendix F for an example of the IANA Media Type registration form).

Respond to reviewer queries
: Once the IANA registration is completed, Vendor and Personal registrations are sent to an IANA-designated reviewer. The reviewer may send the designer questions/suggestions via email in order to clarify and, if needed, modify the registration details in order to comply with IANA standard for Vendor and Personal registrations.

Throughout the process, designers can monitor the progress of the registration using the IANA website (*https://tools.iana.org/public-view/*).

After all questions have been answered and the reviewer is satisfied that the design meets the IANA standard for Vendor and Personal media type registrations, the designer will receive notice of the approved registration along with a link to the IANA website where the official registration will be located.

The process of achieving IANA approval for Vendor/Personal registrations usually takes four weeks or fewer, depending on backlog and the amount of interaction needed between the reviewer and the designer.

Standard tree media type registrations

The process of gaining IANA registration for media types in the Standards Tree is essentially the same (publish, register, review, approval). However, the process of publishing is different. Unlike Vendor/Personal registrations where the only publishing requirement is to produce a complete document at a stable address, Standards Tree registrations require the designer to publish the documentation through the IETF.

See Appendix G for an example of a completed IETF RFC for a Standards Tree Media Type registration.

The IETF process for publishing a media type starts with the creation of an Internet Draft (I-D). This draft is subject to a much more extensive review. Instead of a single IANA-designated reviewer, I-Ds are subject to peer review via an IETF mailing list (*iesg@ietf.org*). Along the way, it is typical for the designer to produce several versions of the I-D in order to accommodate the feedback and comments of IANA peers.

The IETF uses its own XML schema to publish I-Ds and makes templates available to help authors complete valid documents. Each draft is submitted to the IETF via its Data Tracker website (*https://datatracker.ietf.org/submit/*), which includes basic syntax and template checking before automatically logging the new draft, producing diff documents and announcing the new draft to the appropriate mailing list for further review. Details on producing I-Ds can be found here: *http://www.ietf.org/id-info/guidelines.html*.

Once the I-D review is complete, designers will receive confirmation that the document is ready for final review and elevation to an RFC (Request for Comment), the final stage of an IETF publication document. At this time the designer will be able to complete the IANA registration and use the published RFC as the stable address for the registration.

Suggested Stages for Registering a Media Type Design

Typically the first stage for a media type design is the initial publication. This document can be used as a guide for sample implementations. At this point the design could be considered an unstable draft as it is likely that the design will change as new discoveries and uses emerge in the early iterations of the design.

Once the design is stable, it makes sense to register the design with the IANA under the Vendor/Personal Tree. This will subject the design to basic review by the IANA and result in a public registration available for implementors to find and utilize. When the IANA registration is completed, the design can be marked as "IANA registered."

For many uses, achieving IANA registrations in the Vendor/Personal tree is all that is needed. However, if the design is on the path for wide adoption, it is wise to pursue additional review and publication status. This is usually handled via the IEFT, but can also be done through the W3C, ISO, or some other organization.

IANA registration via the Standards Tree means publishing an I-D through the IETF, which can result in an RFC. At this point the RFC becomes the official documentation of the media type. The W3C has a similar process (see "The IETF, W3C, and IANA" on page 138 above) for details.

Link Relation Types

The values used to identify link relations are also subject to registration and review. In 2010, the IETF published RFC 5988 (Web Linking), which describes the process of registering Link Relations Types (sometimes referred to as "rel values"). In that document, the IANA is designated as the official registrar of Link Relation Types. Adding values to the IANA registry currently requires publishing an RFC through the IETF.

Section 4.2 of RFC 5988 also describes Extension Relation Types, which can be used without the need for format registration. In this case, as long as the Link Relation Type is a fully qualified URI, it does not need to be registered in order to be considered a fully compliant Link Relation Type.

In addition to RFC 5988, the W3C's HTML5 documentation designates the Microformats community as the official repository of Link Relation Types for the HTML media type. Adding values to the Microformats repository involves editing a public wiki page and undergoing expert review via the Microformats community.

Registering link relations with the Microformats community

In order to register your Link Relation Type with the Microformats community you need to do the following:

Maintain an active Microformats wiki account
: The Microformats community maintains an active wiki where, among other things, their Link Relations Values (*http://microformats.org/wiki/existing-rel-values*) repository is maintained. An active wiki account is needed in order to contribute to this repository. As of this writing, obtaining an account is very simple and requires only a completed registration form.

Edit the Existing Link Rels Repository
: The Existing Link Rels page on the Microformats wiki includes a number of lists. The most likely place to add new Link Relation Types is the "Non-HTML" list. There are also other lists on that page that, depending on your media type design, may be more appropriate.

Add a new Link Relation Value
: The wiki page expects the following values:

  - Link Relation Value
  - Description (a short sentence that describes the semantic meaning of this value)
  - Documention Reference (e.g. the media type document, RFC, etc. that defines this value)
  - Existing uses of this value (e. g. examples from in the wild where this value is in active use)

Respond to reviewer queries
: Once the new value is added to the wiki, one or more individuals from the Microformats community will review the entry and provide feedback on the addition. This can occur via email or via the Microformats IRC channel (*irc://irc.freenode.net/microformats*).

See Appendix G for an example of a completed Link Relations Value entry for the Microformats community.

Assuming the values are acceptable, they will be allowed to remain on the active wiki page listing. If not, they will be removed after a final decision is made. The total time it takes to successfully publish a new Link Relation Value can be as little as the time it takes to edit the wiki and respond to feedback on the IRC channel, possibly only minutes.

Once the value is accepted, designers can update their documentation to indicate the Link Relation Type is registered with the Microformats community.

Registering link relations with the IANA

Registering a Link Relations Type with the IANA requires submitting an Internet Draft (I-D) to the IETF and shepherding that document to RFC status. Once the RFC is approved, the value will be added to the IANA Link Relations Registry (*http://www.iana.org/assignments/link-relations/link-relations.xml*).

See Appendix G for an example of a completed I-D for an IANA-registered Link Relations Type.

The IETF maintains a public mailing list (*link-relations@ieft.org*) where new relations should be first proposed. This list contains a number of expert reviewers willing to provide feedback and suggestions on how to complete the I-D/RFC process for Link Relation Types. There is also a simple I-D template available to help authors submit proper documents to the IETF Data Tracker.

The process for registering a new Link Relations Type through the IANA is as follows:

Join the IETF's Link Relations mailing list
: Joining the list (*link-relations@ietf.org*) is as easy as providing your email and setting a password. Once you are a member of the list you can post an informal message to the list indicating your interest in publishing a new Link Relations Type. Members of the list will provide general feedback and most likely encourage starting the process of creating an I-D using the supplied template.

Publish an I-D and solicit review
: After publishing the first draft of the I-D, post the results to the mailing list and respond to any and all queries from list members.

Submit the I-D for final review
: Once all the feedback from the mailing list is incorporated in the I-D, the designer will be given the go-ahead to advance the I-D as a proposed RFC. This will trigger a final round of review at the IETF level which can result in a new RFC. Once the RFC is published, the new Link Relation Type(s) will be added to the IANA registry.

Since the process of registering Link Relations Types with the IANA involves completing an IETF RFC, typically this takes weeks, possibly months to complete.

Suggested Stages for Registering a Link Relation Type

If your design needs a new link relation identifier, the first logical choice is to follow RFC 5988 and use a unique URI as the link relation value.

If the new Link Relation turns out to be useful to a wider audience, it may make sense to convert the URI to a single value and register that value with the Microformats community. This step provides an initial level of expert review and can gain an even wider adoption for the Link Relation.

If the Microformats-registered Link Relation achieves wide use, it is a good idea to add that value to the IANA Link Relation Type registry via an IETF RFC document.

Design and Implementation Tips

Aside from the details of proper documentation, registration, and maintenance of a media type design, it helps to keep some general design and implementation tips in mind when creating a new media type or applying a domain-specific implementation to an existing hypermedia type. Below are some additional insights and advice from a number of sources that can assist in building quality, robust, and useful hypermedia APIs.

Joshua Bloch's Characteristics of a Good API

In 2006, Joshua Bloch gave a talk entitled "How to Design a Good API and Why it Matters." While his talk was not geared toward media type designers, his list of "Characteristics of a Good API" is still applicable to hypermedia designs.

Below is a summary of his list:

Easy to learn
: The design of the API should feel natural or familiar.

Easy to use, even without documentation
: The design should not be so complex or unpredictable that developers must constantly consult the documentation in order to use it.

Hard to misuse
: Good APIs won't let you do the wrong thing.

Easy to read and maintain code that uses it
: The design of the API should encourage readable and easily maintainable code.

Sufficiently powerful to satisfy requirements
: More power isn't necessarily better. The API should be powerful enough.

Easy to extend
: Even if you get the API right the first time, you are not likely to get it complete. Over time people will want to do more things with it and you should design your API with extension in mind.

Appropriate to the audience
: An API written for physics is probably not going to be good for finance. You should write the API "in the language of the people who are going to use it."

Bloch also emphasized in the same presentation that APIs can be either a company's or individual's greatest asset, or its greatest liability. It is a good idea to review media type designs with these characteristics as a measure. The easier the media type is to use, maintain, and extend, the more likely it is to become popular and widely implemented.

Roy Fielding's Hypertext API Guidelines

In October of 2008, Roy Fielding posted his list of rules for designing hypermedia APIs ("REST APIs must be hypertext-driven"). That blog post and the subsequent comment stream that followed is a very interesting read, a window into some of the thinking that went into the REST architectural style. Even in cases where Fielding's REST style is not employed, his list for how to approach hypermedia implementations is still valuable.

Below is a summary of Fielding's six rules:

A REST API should not be dependent on any single communication protocol.
: While most designers will assume HTTP is the protocol to be used for an API, it is possible implementations may want to use FTP, XMPP, or some other protocol in the future. Good API designs do not restrict the transfer protocols used in sending and receiving media type messages.

A REST API should not contain any changes to the communication protocols.
: When designing an API, there is no need to create new protocol-level methods (i.e. adding a `SUBSCRIBE` method to the HTTP protocol or redefining the meaning of the HTTP `HEAD` method, etc.). Instead, APIs should allow implementors to use existing protocol methods as they were originally described.

A REST API should spend almost all of its descriptive effort in defining the media type(s) used [and] in defining extended relation names and/or hypertext-enabled markup for existing standard media types.
: API designs should focus on media types and the details surrounding them (decorating data elements, hypermedia controls, etc.). See "Hypermedia Design Elements" on page 20 for details on this process.

A REST API must not define fixed resource names or hierarchies.

Good APIs do not require servers or clients to use specific URIs or hierarchies. Instead, they allow servers to determine their own URI scheme and allow clients to treat URIs as opaque identifiers. This principle was first mentioned in the Tip on page 41.

A REST API should never have typed resources that are significant to the client.

As was discussed in "The Type-Marshaling Dilemma" on page 7, media type designs are not simply serializations of private domain objects. For example, developers should not need to know the object graphs implemented on the server in order to implement a compliant client application. This is covered in "The Type-Marshaling Dilemma" on page 7.

A REST API should be entered with no prior knowledge beyond the initial URI and set of standardized media types.

Good APIs only require client applications to know where to start (an initial URI) and what media types are used. All details of application flow, such as state transitions, should be supplied as hypermedia controls within response representations. Client applications should not contain hardcoded rules on how to navigate links or how and when to compose transitions, but should instead know how to recognize this information within the response representations themselves. See "Application Flow" on page 29 for more on this topic.

By keeping these rules in mind, media type designers can avoid a number of common pitfalls when creating hypermedia APIs.

Jon Postel's Robustness Principle

When authoring the Transmission Control Protocol (TCP) document (RFC 761), Jon Postel included a section entitled "The Robustness Principle."

> "TCP implementations should follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others."

The same general rule applies to implementing media types. Implementations (clients and servers), whenever possible, should strive to send "unconditionally compliant" representations. They should also be lenient when receiving representations. Whenever possible, implementors should do their best to parse and processing representations unless they are found to be unsupportable (based on the Versioning rules above) or absolutely noncompliant (based on the Extending rules above).

A common example of being "liberal in what you accept" is for the receiver to ignore any and all elements that are not recognized or understood. This is commonly referred to as the "must ignore" rule.

Media type designers should keep Postel in mind. Designers can make supporting the Robustness Principle easier for implementors by keeping the number of MUST elements in the compliance profile to a minimum. The fewer MUST elements implementors need to support, the more likely it is that they will be able to craft compliant representations using that media type.

Other Considerations

Below are some other tips and considerations when designing media types and hypermedia APIs.

Getting it right but not complete

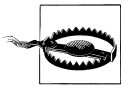
When designing a media type, it is important to get it right the first time. The primary reason for this goal is that public media types are forever. Once you register your media type and/or create working implementations, it will be extremely difficult to alter your design in ways that break existing implementations. As has already been pointed out, you can safely extend your design, but rarely can you version it without adversely affecting already working clients and severs.

Good media type designers focus on getting the initial design right (i.e. doing a good job of solving the problem at hand) and ensuring the media type can easily support extensions for future uses.

This last point is important. A well-designed media type supports future extendability. That means it is not necessarily important that your first release is complete. It need not cover all perceived use cases, but your design does need to support the possibility of future extensions to cover unforeseen uses and applications.

Maintain media type design and schema separately

Media types themselves exist as standalone designs. While it may be that some media types lend themselves to validation via a published schema (DTD, XSD, RelaxNG, etc.), implementing media types should not require the use of schema documents. Part of the reason for this is to continue to encourage Postel's Robustness Principle. Implementations should be liberal in what they accept. The use of schema document in implementations may lead to rejecting representations that could be successfully processed.



It is the author's opinion that one of the things that has hampered the adoption of XHTML (1.0, 1.1, and 2.0) is the strict implementation of schema validation employed by common web browsers.

This is not to say that media type designs cannot benefit from schema. A simple definition of the word schema is "a diagrammatic representation; an outline or model." In this light, every media type design relies on some form of schema even if that schema exists only in the head of the designer. Independent validation of representations can be useful diagnostic aid when testing an implementation. The W3C maintains several validation services that allow users to submit representations and/or links to representations in order to test their work. It may be a good idea to provide a validation service for your designs, too.

Finally, it should be noted that it is possible to update the schema of an existing media type without changing its version. For example, the HTML media type has gone through several updates (1.0, 2.0, 3.2, 4.01, 5) and in each case the MIME media type identifier has stayed the same (`text/html`). That is because the media type has not changed versions (see "Versioning" on page 154 for more on this topic) but has instead only changed its schema.*

Authentication is not part of the hypermedia design

The reader may have noticed that none of the designs shown in this book include authentication elements. That is because authentication is usually orthogonal to common transfer protocols. HTTP, for example, supports an extensible authentication model based on two key headers (`WWW-Authenticate` and `Authorization`) along with specific status codes (`401 Not Authorized` and `403 Forbidden`).

As of this writing, there is an RFC that defines two authentication schemes for HTTP: Basic and Digest (RFC 2617). There are additional authentication schemes, some of which do not use the method outlined for HTTP.

It is a good design practice to keep authentication details removed from media type designs. Ideally, media types should not be coupled with, or limited to, any authentication scheme. This allows implementors to select their own authentication details and even change them over time without any impact on the media type design itself.

Testing media type designs

When creating media type designs, be sure to test them by implementing example servers and clients. Often, designs that seem to make sense in the beginning turn out to have notable weaknesses or shortcomings once you try to create working implementations. If possible, try to enlist others into creating clients and servers based on your media type design.

* The concept of viewing schema updates differently than media type versions was first introduced to me by Jan Algermissen.

Also, it is important to note that testing media type designs requires more than validating requests and responses that are properly formed. The ultimate aim of those using a media type is to do something (i.e. add or modify record in the system, compute and result, produce a report, etc.). Therefore it is important to test whether client applications can actually accomplish goals they set out to achieve. In other words, designers should assess not just the correctness of their designs, but also their suitability for meeting the needs of those using it.

One way to test the quality of media type design work is to create an environment where several test developers have no reference material other than the media type design and any accompanying domain-specific mapping for that media type. If the resulting working implementations can achieve true interoperability among each other, even when the testing parties completed their implementations independently, it is likely that the design and the documentation are both of sufficient quality to support wide adoption.

Afterword

When you have completed 95 percent of your journey, you are only halfway there.

- Japanese Proverb

And so we reach the end of this little adventure.* But, even though the page count for this project has reached its conclusion, the book itself is far from *complete*.

Over the last few years, I have been researching, experimenting, and speaking about the role hypermedia plays in implementing long-lived solutions running over HTTP. A distillation of many of those ideas appears within these pages. However, there are still many more areas to explore, many more examples that could be provided to amplify and clarify the ideas exposed here. In my mind, at least, this list is endless.

However, the aim of this book was not to create a definitive work on designing hypermedia APIs. Instead, it was to identify helpful concepts, suggest useful methodologies, and provide pertinent examples that encourage architects, designers, and developers to see the value and utility of hypermedia in their own implementations. This book is more of an invitation to explore than an admonition to follow my lead.

At the start of many of my presentations I warn my audience to be aware that I have an underlying agenda of which they should be aware. That, along with imparting information about the power and promise of hypermedia-oriented designs, my work has another purpose. I tell my listeners that one of my goals is, quite simply, to increase the amount of links and forms in the data messages sent along the Internet; to increase the usability and utility of each and every response representation.

I hope that this book gives readers the tools and encouragement to do the same; that readers will be able to look at their current projects with a new eye for the value hypermedia can provide and that, as a result, we can enjoy the benefits of a more connected more hyper-Internet.

Mike Amundsen, September 2011

* As Simon (my exasperated editor) will attest, the end has been too long in coming!

APPENDIX A
References

Chapter 1

HTTP/0.9, *http://www.w3.org/Protocols/HTTP/AsImplemented.html*

W3 address syntax: BNF, *http://www.w3.org/Addressing/BNF.html*

Basic HTTP as defined in 1992, *http://www.w3.org/Protocols/HTTP/HTTP2.html*

Hypertext Transfer Protocol -- HTTP/1.0, *http://tools.ietf.org/html/rfc1945*

Hypertext Transfer Protocol -- HTTP/1.1, *http://tools.ietf.org/html/rfc2068*

Hypertext Transfer Protocol -- HTTP/1.1, *http://tools.ietf.org/html/rfc2616*

Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, *http://tools.ietf.org/html/rfc2046*

Returning Values from Forms: multipart/form-data, *http://tools.ietf.org/html/rfc2388*

Scalable Vector Graphics (SVG) 1.1 (Second Edition), *http://www.w3.org/TR/SVG/*

Cascading Style Sheets, *http://www.w3.org/Style/CSS/*

The Atom Publishing Protocol, *http://tools.ietf.org/html/rfc5023*

The Atom Syndication Format, *http://tools.ietf.org/html/rfc4287*

HTML Current Status, *http://www.w3.org/standards/techs/html*

JSON, *http://json.org/*

JSONPath, *http://goessner.net/articles/JsonPath/*

JSON Schema, *http://json-schema.org/*

Extensible Markup Language (XML), *http://www.w3.org/XML/*

XML Inclusions (XInclude) Version 1.0 (Second Edition), *http://www.w3.org/TR/xinclude/*

XSL Transformations (XSLT), *http://www.w3.org/TR/xslt*

XML Path Language (XPath), *http://www.w3.org/TR/xpath/*

XQuery 1.0: An XML Query Language (Second Edition), *http://www.w3.org/TR/xquery/*

XML Schema, *http://www.w3.org/XML/Schema*

Schematron, *http://www.schematron.com/*

XML Pointer Language (XPointer), *http://www.w3.org/TR/xptr/*

XML Linking Language (XLink) Version 1.1, *http://www.w3.org/TR/xlink11/*

XForms 1.1, *http://www.w3.org/TR/xforms11/*

Resource Description Framework (RDF), *http://www.w3.org/RDF/*

RDF Forms, *http://www.markbaker.ca/2003/05/RDF-Forms/*

RDF/XML Syntax Specification (Revised), *http://www.w3.org/TR/rdf-syntax-grammar/*

RDFa, *http://rdfa.info/*

Markdown, *http://daringfireball.net/projects/markdown/*

YAML: YAML Ain't Markup Language, *http://www.yaml.org/*

Common Format and MIME Type for Comma-Separated Values (CSV) Files, *http://www.rfc-editor.org/rfc/rfc4180.txt*

Protocol Buffers, *http://code.google.com/apis/protocolbuffers/*

Voice Extensible Markup Language (VoiceXML) 2.1, *http://www.w3.org/TR/voicexml21/*

Microformats Existing Rel Values, *http://microformats.org/wiki/existing-rel-values*

IANA Link Relations, *http://www.iana.org/assignments/link-relations/link-relations.xml*

Dublin Core Terms, *http://dublincore.org/documents/dcmi-terms/*

URI Resolution Services Necessary for URN Resolution - The text/uri-list Internet Media Type, *http://tools.ietf.org/html/rfc2483#section-5*

OpenSearch 1.1 (Draft 4), *http://www.opensearch.org/Specifications/OpenSearch/1.1*

The application/opensearchdescription+xml media type (Expired Internet Draft), *http://tools.ietf.org/html/draft-ellermann-opensearch-01*

Web Linking, *http://tools.ietf.org/html/rfc5988*

URI Template draft-gregorio-uritemplate-04, *http://tools.ietf.org/html/draft-gregorio-uritemplate-04*

Latest SOAP versions, *http://www.w3.org/TR/soap/*

Chapter 2

Key words for use in RFCs to Indicate Requirement Levels, *http://tools.ietf.org/html/rfc2119*

REST APIs must be hypertext-driven, *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven*

Uniform Resource Identifier (URI): Generic Syntax (RFC3986), *http://tools.ietf.org/html/rfc3986*

HTML 4.01 Document Type Definition: Link Types, *http://www.w3.org/TR/html401/sgml/dtd.html#LinkTypes*

HTML 4.01 - SGML Basic Types, *http://www.w3.org/TR/html4/types.html#h-6.2*

HTML 4.01 Specification, http://www.w3.org/TR/html4/

Namespaces in XML 1.0 (Third Edition), *http://www.w3.org/TR/xml-names/*

Chapter 3

The Atom Publishing Protocol, *http://tools.ietf.org/html/rfc5023*

The Atom Syndication Format, *http://tools.ietf.org/html/rfc4287*

Key words for use in RFCs to Indicate Requirement Levels, *http://tools.ietf.org/html/rfc2119*

JSON Grammar, *http://tools.ietf.org/html/rfc4627#section-2*

The application/json Media Type for JavaScript Object Notation (JSON) (RFC4627), *http://tools.ietf.org/html/rfc4627*

Uniform Resource Identifier (URI): Generic Syntax (RFC3986), *http://tools.ietf.org/html/rfc3986*

Chapter 4

XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition), *http://www.w3.org/TR/xhtml1/*

HTML 4.01 Specification, *http://www.w3.org/TR/html401/*

REST APIs must be hypertext-driven, *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven*

Xhtml Meta Data Profiles, *http://gmpg.org/xmdp/*

H-Factors, *http://amundsen.com/hypermedia/hfactor/*

HTTP Authentication: Basic and Digest Access Authentication [RFC2617], *http://tools.ietf.org/html/rfc2617*

Key words for use in RFCs to Indicate Requirement Levels [RFC2119], *http://tools.ietf.org/html/rfc2119*

HTTP State Management Mechanism [RFC2109], *http://tools.ietf.org/html/rfc2109*

Internet Message Format [RFC5322], *http://tools.ietf.org/html/rfc5322*

Uniform Resource Identifier (URI): Generic Syntax [RFC3986], *http://tools.ietf.org/html/rfc3986*

Date and Time on the Internet: Timestamps [RFC3339], *http://tools.ietf.org/html/rfc3339*

Microformats, *http://microformats.org*

W3C Markup Validation Service, *http://validator.w3.org/*

Microdata, *http://www.w3.org/TR/microdata/*

RDFa in XHTML: Syntax and Processing, *http://www.w3.org/TR/rdfa-syntax/*

RDFa Primer, *http://www.w3.org/TR/xhtml-rdfa-primer/*

Chapter 5

Key words for use in RFCs to Indicate Requirement Levels, S. Bradner, 1997, *http://tools.ietf.org/html/rfc2119*

Guidelines to Authors of Internet-Drafts, R. Housely, 2010, *http://www.ietf.org/id-info/guidelines.html*

The Internet Standards Process -- Revision 3, S. Bradner 1996, *http://tools.ietf.org/html/rfc2026*

QA Framework: Specification Guidelines, Dubost, et al, 2005, *http://www.w3.org/TR/qaframe-spec/*

HTTP Header Field Registrations, Nottingham & Mogul, 2005, *http://tools.ietf.org/html/rfc4229*

Object Headers, Tim Berners-Lee, 1992, *http://www.w3.org/Protocols/HTTP/Object_Headers.html*

TRANSMISSION CONTROL PROTOCOL, Jon Postel, 1981, *http://tools.ietf.org/html/rfc793*

How to Design a Good API and Why it Matters, Joshua Bloch, 2006, *http://www.infoq.com/presentations/effective-api-design*

REST APIs must be hypertext-driven, Roy Fielding, 2008 *http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven*

HTTP Authentication: Basic and Digest Access Authentication, Franks, et al., 1999, *http://www.ietf.org/rfc/rfc2617.txt*

Media Type Specifications and Registration Procedures, Freed & Klensin, 2005, *http://tools.ietf.org/html/rfc4288*

APPENDIX B
Additional Reading

Books

Alexander, Christopher. The Timeless Way of Building. New York; Oxford University Press; 1979.

Anderson, J Chris; Lehnardt, Jan; Slater, Noah. CouchDB : The Definitive Guide, Sebastopol, CA: O'Reilly, 2011.

Brusilovsky, Peter; Kobsa, Alfred, Vassileva, Julita. Adpative Hypertext and Hypermedia, Klwer Academic Publishers, 1998

Holt, Bradley. Scaling CouchDB, Sebastopol, CA: O'Reilly, 2011.

Holt, Bradley. MapReduce Views in CouchDB, Sebastopol, CA: O'Reilly, 2011

Hughes-Croucher, Thomas. Node: Up and Running, O'Reilly Media. 2011

Lowe, David; Hall, Wendy. Hypermedia & the Web : An Engineering Approach, Wiley & Sons, 1999

Nelson, Ted. Computer Lib/Dream Machines, Redmond, WA : Tempus Books, 1974.

Pilgrim, Mark. HTML5: up and running. Sebastopol, CA: O'Reilly, 2010.

Taylor, Medividovic, Dashofy. Software Architecture: Foundations, Theory, and Practice. Wiley, 2009

Teixeira, Pedro. Hands On Node.JS, Self-Published. 2011

Vignelli, Massimo. The Vignelli Canon. Baden, Switzerland; Lars Muller Publishers, 2010.

Other

Bush, Vannevar (July 1945). "As We May Think," Atlantic Magazine *http://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/3881/*

Engelbart, Douglas C. (October 1962). "Augmenting Human Intellect: A Conceptual Framework." SRI Summary Report AFOSR-3223. Prepared for: Director of Information Sciences, Air Force Office of Scientific Research. *http://dougengelbart.org/pubs/augment-3906.html*

Fielding, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000. *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

APPENDIX C

Maze+XML Media Type

The Maze+XML media type is an XML data format for sharing maze state information between clients and servers. It can be used to implement simple mazes, adventure games, and other related data.

The IANA Registry lists the media type identifier as `application/vnd.amundsen.maze+xml`.

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Elements

Below is a map of the Maze+XML media type. This map shows all of the possible elements, attributes, link relations, and data types.

It should be noted that the media type map below does not necessarily represent a valid instance of a Maze+XML document. It is only useful as a map to explore the various features of the media type:

```
<maze version="1.0">
  <collection href="URI">
    <link href="URI" rel="maze" />
    <link href="URI" rel="maze" />
    ...
  </collection>
  <item href="URI" >
    <link href="URI" rel="start" />
    <debug>CDATA</debug>
  </item>
  <cell href="URI"  debug="TEXT" total="NUMBER" side="NUMBER">
    <link href="URI" rel="current" debug="TEXT" total="NUMBER" side="NUMBER" /> <!--
alternate 'current' link -->
    <link href="URI" rel="north" />
```

```
    <link href="URI" rel="south" />
    <link href="URI" rel="east" />
    <link href="URI" rel="west" />
    <link href="URI" rel="exit" />
  </cell>
  <error href="URI">
    <title>TEXT</title>
    <code>TEXT</code>
    <message>CDATA</message>
  </error>
</maze>
```

cell

The `<cell>` element represents a single cell or block in the maze.

The `<cell>` element SHOULD have two attributes: `href` and `rel`. The `href` attribute MUST contain a valid URI. The `rel` attribute SHOULD be set to "current".

The `<cell>` element MAY have one or more of the following attributes: `debug`, `total`, and `size`.

The `<cell>` element MAY have one or more `<link>` child elements.

code

The `<code>` element is a child element of the `<error>` element. It can be used to provide a human-friendly error code for the error being reported. It SHOULD contain valid TEXT data. It is an OPTIONAL element.

collection

The `<collection>` element represents the list of available mazes in this collection.

The `<collection>` element SHOULD have an `href` attribute with a URI value that points to this resource. If present, this URI MAY be used with the HTTP POST method to create a new maze or game as a child resource of the collection URI.

The `<collection>` element MAY have one or more `<link>` child elements. Each of these child elements SHOULD have a `rel` attribute with a value of "maze". This indicates the associated href of the link element has a URI which points to the the starting point of a maze or game.

debug

This element is a child of the `<item>`. It contains additional information that can be used to help debug or inspect the state of the maze or game. This data is defined by and supplied by the server. This element SHOULD contain only valid CDATA. This is an OPTIONAL element.

error

The <error> element is an OPTIONAL element. It MAY contain one or more child elements. The list of valid child elements is: <title>, <code>, <message>, and <link>.

Each of the above child elements MUST appear no more than once within the <error> element. If the <error> element appears in the document, it SHOULD be the only child element of the <maze> element.

item

The <item> element represents a complete maze or game. The <item> element SHOULD have an href attribute. The href attribute MUST contain a valid URI. The <item> element SHOULD have a <link> child element whose attributes SHOULD point to the start of a maze or game. The <item> element MAY have a <debug> child element.

link

The <link> element MUST have two attributes: href and rel. The href attribute MUST contain a valid URI. The rel attribute SHOULD contain one of the valid Link Relation values. The <link> MAY appear as the child element of the <collection>, <item>, and <error> elements.

If the <link> element has a rel attribute set to "current" it MAY also have the following additional attributes: debug, size, and total.

maze

A valid Maze+XML document MUST have a root element of <maze>. It MAY have one or more of the following child elements: <collection>, <item>, <cell>, and <error>.

Each of these child elements MUST NOT appear more than once within the document. If the <error> element appears, it SHOULD be the only child element of the document.

message

The <message> element is a child element of the <error> element. It holds additional information about the error that is represented by the <error> element. It SHOULD contain only valid CDATA characters. It is an OPTIONAL element.

title

The `<title>` element is a child element of the `<error>` element. It can be used to provide a human-friendly title for the error being reported. It SHOULD contain valid TEXT data. It is an OPTIONAL element.

Attributes

Below is a list of all the attributes used in the Maze+XML media type:

debug

This attribute contains text data that represents debugging information supplied by the server. Its value MUST be TEXT. This is an OPTIONAL attribute. If present, it MUST only appear on the `<cell>` element.

href

This attribute specifies the location of a Maze+XML resource, thus defining a link between the current resource and the destination resource defined by this attribute. The value of the attribute MUST always be a valid URI.

This is an OPTIONAL attribute for the following elements: `<collection>`, `<item>`, `<cell>`, and `<error>`.

This is a REQUIRED attribute for the `<link>` element.

rel

This attribute describes the relationship from the current representation to the URI specified by the `href` attribute. The value of this attribute SHOULD be one of the values listed in the Attributes section of this document. This is a REQUIRED attribute for the `<link>` element.

side

This attribute indicates the number of cells on a single side of the maze represented by this document. The value of this attribute MUST be a valid NUMBER. This is an OPTIONAL attribute. If it is present, it MUST only appear on the `<cell>` element.

total

This attribute indicates the total number of cells in the maze represented by this document. The value of this attribute MUST be a valid NUMBER. This is an OPTIONAL attribute. If it is present,it MUST only appear on the `<cell>` element.

version

Indicates the version of this Maze+XML media-type document. The value of this attribute MUST be set to "1.0" for this version of the documentation. This is an OPTIONAL attribute. If it is present, it MUST only appear on the `<maze>` element.

Link Relations

Below is a list of all the valid link relation values for the `rel` attribute of the Maze+XML document:

collection

Refers to a resource that returns a list of the target document/entities, etc. Logged with the Microformats Existing Rel Values.

When used in the Maze+XML media type, the associated URI returns the available collections of mazes.

current

Refers to resource containing the most recent item(s) in a collection of resources. Registered through IANA Link Relations.

When used in the Maze+XML media type, the associated URI returns the client's current position in the active maze.

east

Refers to a resource to the east of the current resource. Logged with the Microformats Existing Rel Values.

When used in the Maze+XML media type, the associated URI points to a neighboring cell resource to the east in the active maze.

exit

Refers to a resource that represents the exit or end of the current client activity or process. Logged with the Microformats Existing Rel Values.

When used in the Maze+XML media type, the associated URI points to the final exit resource of the active maze.

maze

Refers to a single maze/game document.

When used in the Maze+XML media type, the associated URI points to an existing resource that represents a maze.

north

Refers to a resource that is north of the current resource. Logged with the Microformats Existing Rel Values.

When used in the Maze+XML media type, the associated URI points to a neighboring cell resource to the north in the active maze.

south

Refers to a resource that is south of the current resource. Logged with the Microformats Existing Rel Values.

When used in the Maze+XML media type, the associated URI points to a neighboring cell resource to the south in the active maze.

start

Refers to the first resource in a collection of resources. Registered through IANA Link Relations.

When used in the Maze+XML media type, the associated URI refers to the starting cell resource within a maze.

west

Refers to a resource that is west of the current resource. Logged with the Microformats Existing Rel Values.

When used in the Maze+XML media type, the associated URI points to a neighboring cell resource to the west in the active maze.

Data Types

Below are the data types used in Maze+XML documents. The reference for most of the data type definitions below is the SGML basic types section of the HTML 4.01 Specification.

CDATA

CDATA is a sequence of characters from the document character set and may include character entities.

NUMBER

Numbers MUST contain at least one digit ([0-9]). The characters ".", "-", and "+" MAY also appear.

TEXT

Text is meant to be human readable.

URI

URI is defined by RFC 3986.

Extensibility

This document describes the Maze+XML markup vocabulary. Markup from other vocabularies (i.e. foreign markup) can be used in a Maze+XML document. Any extensions to the Maze+XML vocabulary MUST not redefine any elements, attributes, link relations, or data types defined in this document. Clients that do not recognize extensions to the Maze+XML vocabulary SHOULD ignore them.

The details of designing and implementing Maze+XML is beyond the scope of this document.

It is possible that future forward-compatible modifications to this specification will include new elements, attributes, link relations, and data types. Extension designers should take care to prevent future modifications from breaking or redefining those extensions. The safest way to do this is to use a unique XML Namespace for each extension.

APPENDIX D

Collection+JSON Media Type

Collection+JSON is a JSON-based read/write hypermedia type designed to support the management and querying of simple collections. It is similar to the The Atom Syndication Format (RFC 4287) and The Atom Publishing Protocol (RFC 5023) . However, Collection+JSON defines both the format and the semantics in a single document and includes support for Query Templates and expanded write support through the use of a Write Template.

The IANA Registry lists the media type identifier as *application/vnd.Collection+JSON*.

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

General Concepts

This section contains general concepts that apply to all Collection+JSON documents.

The Collection+JSON hypermedia type is designed to support full read/write capability for simple lists (e.g. contacts, tasks, blog entries, etc.). The standard application semantics supported by this media type include Create, Read, Update, and Delete (CRUD) along with support for predefined queries including query templates (similar to HTML GET forms). Write operations are defined using a template object supplied by the server as part of the response representation.

Each item in a Collection+JSON collection has an assigned URI (via the `href` property) and an optional array of one or more `data` elements along with an optional array of one or more `link` elements. Both arrays support a `name` property for each object in the collection in order to decorate these elements with domain-specific semantic information (e.g. `"data" : [{"name" : "first-name", ...},...]`).

The Collection+JSON hypermedia type has a limited set of predefined link relation values and supports additional values applied by implementors in order to better describe the application domain to which the media type is applied.

The following sections describe the process of reading and writing data using the Collection+JSON hypermedia type as well as the way to parse and execute Query Templates.

Reading and Writing Data

The Collection+JSON media type supports a limited form of read/write semantics: the Create-Read-Update-Delete (or CRUD) pattern. In addition to CRUD operations, the Collection+JSON media type supports Query Templates.

This section describes the details of how clients can recognize and implement reads and writes found within Collection+JSON responses.

Reading collections

To get a list of the `items` in a `collection`, the client sends an HTTP GET request to the URI of a collection. A Collection+JSON Document is returned whose `item` array contains the URI of `item` resources. The response may describe all, or only a partial list, of the items in a collection:

```
*** REQUEST ***
GET /my-collection/ HTTP/1.1
Host: www.example.org
Accept: application/vnd.Collection+JSON

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.Collection+JSON
Content-Length: xxx

{ "collection" : {...}, ... }
```

Adding an item

To create a new item in the collection, the client first uses the template object to compose a valid item representation and then uses HTTP POST to send that representation to the server for processing.

If the item resource was created successfully, the server responds with a status code of 201 and a Location header that contains the URI of the newly created item resource:

```
*** REQUEST ***
POST /my-collection/ HTTP/1.1
Host: www.example.org
Content-Type: application/vnd.Collection+JSON

{ "template" : { "data" : [ ...] } }
```

```
*** RESPONSE ***
201 Created HTTP/1.1
Location: http://www.example.org/my-collection/1
```

Once an item resource has been created and its URI is known, that URI can be used to read, update, and delete the resource.

Reading an item

Clients can retrieve an existing item resource by sending an HTTP GET request to the URI of an item resource. If the request is valid, the server will respond with a representation of that item resource.

```
*** REQUEST ***
GET /my-collection/1 HTTP/1.1
Host: www.example.org
Accept: application/vnd.Collection+JSON

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.Collection+JSON
Content-Length: xxx

{ "collection" : { "href" : "...", "items" [ { "href" : "...", "data" : [...].} } }
```

Note that the valid response is actually a complete collection document that contains only one item (and possibly related queries and template properties).

Updating an item

To update an existing resource, the client uses the template object as a guide to composing a replacement item representation and then uses HTTP PUT to send that representation to the server.

If the update is successful, the server will respond with HTTP status code 200 and possibly a representation of the updated item resource representation:

```
*** REQUEST ***
PUT /my-collection/ HTTP/1.1
Host: www.example.org
Content-Type: application/vnd.Collection+JSON

{ "template" : { "data" : [ ...] } }

*** RESPONSE ***
200 OK HTTP/1.1
```

Deleting an item

Clients can delete existing resources by sending an HTTP DELETE request to the URI of the item resource.

If the delete request is successful, the server SHOULD respond with an HTTP status code of 204:

```
*** REQUEST ***
DELETE /my-collection/ HTTP/1.1
Host: www.example.org

*** RESPONSE ***
204 No Content HTTP/1.1
```

Query Templates

Clients that support the Collection+JSON media type SHOULD be able to recognize and parse query templates found within responses. Query templates consist of a data array associated with an `href` property. The queries array supports query templates.

For query templates, the name/value pairs of the data array set are appended to the URI found in the `href` property associated with the queries array (with a question mark as separator) and this new URI is sent to the processing agent:

```
// query template sample
{
  "queries" :
  [
    {
      "href : "http://example.org/search",
      "rel" : "search",
      "prompt" : "Enter search string",
      "data" :
      [
        {"name" : "search", "value" : ""}
      ]
    }
  ]
}
```

In the above example, if the user supplied "JSON" for the value property, the user agent would construct the following URI:

```
http://example.org/search?search=JSON
```

Objects

An object is an unordered collection of zero or more name/value pairs, where a name is a string and a value is a string, number, boolean, null, object, or array. The following elements are always represented as objects in Collection+JSON documents: collection, error, and template.

collection

The collection object contains all of the records in the representation. This is a REQUIRED object and there MUST NOT be more than one collection object in a Collection+JSON document. It is a top-level document property.

The collection object SHOULD have a version property. For this release, the value of the version property MUST be set to 1.0. If there is no version property present, it should be assumed to be set to 1.0.

The collection object SHOULD have an `href` property. The `href` property MUST contain a valid URI. This URI SHOULD represent the address used to retrieve a representation of the document. This URI MAY be used to add a new record (See Reading and Writing Data).

The collection object MAY have a links array child property.

The collection object MAY have an items array child property.

The collection object MAY have a queries array child property.

The collection object MAY have a template object child property.

The collection object MAY have an error object child property.

```
// sample collection object
{
  "collection" :
  {
    "version" : "1.0",
    "href" : URI,
    "links" : [ARRAY],
    "items : [ARRAY],
    "queries" : [ARRAY],
    "template" : {OBJECT},
    "error" : {OBJECT}
  }
}
```

error

The error object contains additional information on the latest error condition reported by the server. This is an OPTIONAL object and there MUST NOT be more than one error object in a Collection+JSON document. It is a top-level document property.

The following elements MAY appear as child properties of the error object: code, message, and title.

```
// sample error object
{
  "error" :
  {
    "title" : STRING,
    "code" : STRING,
```

```
    "message" : STRING
  }
}
```

template

The template object contains all of the input elements used to add or edit collection records. This is an OPTIONAL object and there MUST NOT be more than one template object in a Collection+JSON document. It is a top-level document property.

The template object SHOULD have a data array child property:

```
// sample template object
{
  "template" :
  {
    "data" : [ARRAY]
  }
}
```

Arrays

An array is an ordered sequence of zero or more values. The following elements are always represented as arrays in Collection+JSON documents: data, items, links, and queries.

data

The data array is a child property of the items array and the template object. The data array SHOULD contain one or more anonymous objects. Each object MAY have any of three possible properties: name (REQUIRED), value (OPTIONAL), and prompt (OPTIONAL).

```
// sample data array
{
  "template" :
  {
    "data" :
    [
      {"prompt" : STRING, "name" : STRING, "value" : VALUE},
      {"prompt" : STRING, "name" : STRING, "value" : VALUE},
      ...
      {"prompt" : STRING, "name" : STRING, "value" : VALUE}
    ]
  }
}
```

items

The items array represents the list of records in the Collection+JSON document. It is a child property of the collection object. Each element in the items array SHOULD contain an `href` property. The `href` property MUST contain a URI. This URI MAY be used to retrieve a Collection+JSON document representing the associated item. It MAY be used to edit or delete the associated item. See Reading and Writing Data for details.

The items array MAY have a data array child property.

The items array MAY have a links array child property.

```
// sample items array
{
  "collection" :
  {
    "version" : "1.0",
    "href" : URI,
    "items" :
    [
      {
        "href" : URI,
        "data" : [ARRAY],
        "links" : [ARRAY]
      },
      ...
      {
        "href" : URI,
        "data" : [ARRAY],
        "links" : [ARRAY]
      }
    ]
  }
}
```

links

The links array is an OPTIONAL child property of the items array. It SHOULD contain one or more anonymous objects. Each has five possible properties: href (REQUIRED), rel (REQURIED), name (OPTIONAL), render (OPTIONAL), and prompt (OPTIONAL).

```
// sample links array
{
  "collection" :
  {
    "version" : "1.0",
    "href" : URI,
    "items" :
    [
      {
        "href" : URI,
        "data" : [ARRAY],
```

```
        "links" :
        [
         {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING, "render" :
"image"},
         {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING, "render" :
"link"},
          ...
         {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING}
        ]
      }
    ]
  }
}
```

queries

The queries array is an OPTIONAL top-level property of the Collection+JSON document. The queries array SHOULD contain one or more anonymous objects. Each object composed of five possible properties: href (REQUIRED), rel (REQUIRED), name (OPTIONAL), prompt (OPTIONAL), and a data array (OPTIONAL).

If present, the data array represents query parameters for the associated href property of the same object. See Query Templates for details.

```
// sample queries array
{
  "queries" :
  [
    {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING},
    {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING,
      "data"
      [
        {"name" : STRING, "value" : VALUE}
      ]
    },
    ...
    {"href" : URI, "rel" : STRING, "prompt" : STRING, "name" : STRING}
  ]
}
```

Properties

Properties represent individual name/value pairs for objects within Collection+JSON documents.

code

The `code` property MAY be a child property of the error object. It SHOULD be a `STRING` type.

href

The `href` property MAY be a child property of the `collection` object, `items` array, `links` array, and `queries` array elements. It MUST contain a valid URI.

message

The `message` property MAY be a child property of the `error` object. It SHOULD be a `STRING` type.

name

The `name` property MAY be a child element of the `data` array and the `links` array. It SHOULD be a `STRING` data type.

prompt

The `prompt` property MAY appear as a child property of the `queries` array, the `links` array, and the `data` array. It SHOULD be a `STRING` data type.

rel

The `rel` MAY be a child property of the `links` array elements and `queries` array elements. It SHOULD be a `STRING` data type.

render

The `render` property MAY be a child property of the `links` array element. It SHOULD be a `STRING` data type. The value MUST be either `image` or `link`. If the `render` property does not appear on a `links` array element, it should be assumed to be set to `link`.

title

The `title` property MAY be a child property of the `error` object. It SHOULD be a `STRING` type.

value

The `value` property MAY be a child element of `data` array elements. It MAY contain one of the following data types: `STRING`, `NUMBER`, `true`, `false`, or `null`.

version

The `version` SHOULD be a child property of the `collection` element. It SHOULD be a `STRING` data type. For this release, the version SHOULD be set to "1.0".

Link Relations

Link relation values can be used to annotate the `links` array and `queries` array properties.

collection

Refers to a `collection` document.

items

Refers to an individual `item` in a `collection`.

template

Refers to the `template` object of a `collection`.

queries

Refers to the `queries` array of a `collection`.

Other Link Relation Values

The Collection+JSON media type has a limited set of defined link relations. Implementors are free to use any additional link relation values as needed and are encouraged to use already-defined link relation values found in existing registries. Suggested registries include but are not limited to:

IANA Link Relations
Microformats Existing Rel Values
Expressing Dublin Core metadata using HTML/XHTML meta and link elements

Implementors may also wish to create their own unique link relation values using the standards outlined in RFC 5988.

Data Types

Below are the data types used in Collection+JSON documents. The reference for most of the data type definitions below is the JSON Grammar section of the RFC 4627.

ARRAY

An `ARRAY` structure is represented as square brackets surrounding zero or more values (or elements). Elements are separated by commas.

NUMBER

A `NUMBER` contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part.

Octal and hex forms are not allowed. Leading zeros are not allowed. A fraction part is a decimal point followed by one or more digits.

An exponent part begins with the letter E in upper or lowercase, which may be followed by a plus or minus sign. The E and optional sign are followed by one or more digits.

OBJECT

An `OBJECT` structure is represented as a pair of curly brackets surrounding zero or more name/value pairs (or members). A name is a string. A single colon comes after each name, separating the name from the value. A single comma separates a value from a following name. The names within an object SHOULD be unique.

STRING

A `STRING` begins and ends with quotation marks. All Unicode characters may be placed within the quotation marks except for the characters that must be escaped: quotation mark, reverse solidus, and the control characters (U+0000 through U+001F).

URI

A URI is defined by RFC 3986.

VALUE

A `VALUE` data type MUST be a `NUMBER`, `STRING`, or one of the following literal names: `false`, `null`, or `true`.

This release of Collection+JSON does not support `OBJECT` or `ARRAY` as a valid `VALUE`.

Extensibility

This document describes the Collection+JSON markup vocabulary. Markup from other vocabularies (i.e. foreign markup) can be used in a Collection+JSON document. Any extensions to the Collection+JSON vocabulary MUST not redefine any objects (or their properties), arrays, properties, link relations, or data types defined in this document. Clients that do not recognize extensions to the Collection+JSON vocabulary SHOULD ignore them.

The details of designing and implementing Collection+JSON extensions is beyond the scope of this document.

It is possible that future forward-compatible modifications to this specification will include new objects, arrays, properties, link relations, and data types. Extension designers should take care to prevent future modifications from breaking or redefining those extensions.

APPENDIX E

Microblogging HTML Semantic Profile

The purpose of a Semantic Profile is to document the application-level semantics of a particular implementation. This is accomplished by describing elements of response representations for a target media type. For example, identifying markup elements returned (i.e. semantic HTML with Microformats) and state transitions (i.e. HTML.A and HTML.FORM elements) that advance the state of the current application.

It should be noted that this documentation does not contain any of the following:

- URI construction rules
- Suggested resource names
- HTTP request/response samples
- Example resource representations

This work was inspired by Roy T. Fielding's "REST APIs must be hypertext driven" blog post. The implementation approach is based on Tantek Çelik's HTML Meta Data Profiles.

General Concepts

The profile here contains details on customizing the HTML media type for a specific application domain: microblogging. It contains descriptions of valid `@class`, `@id`, `@name`, and `@rel` values that can appear within HTML resource representations. The identified base media type (HTML) along with the list of attributes, values, and their meaning describes a hypermedia interface. This document is presented as a complete blueprint for implementing a compliant client or server that supports the basic features of the target application domain (microblogging).

Compliance

An implementation is not compliant if it fails to satisfy one or more of the MUST or REQUIRED level requirements. An implementation that satisfies all of the MUST or

REQUIRED and all of the SHOULD level requirements, is said to be "unconditionally compliant." One that satisfies all of the MUST level requirements but not all the SHOULD level requirements is said to be "conditionally compliant."

> The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

Design Characteristics

Base Format
: HTML

Domain Semantics
: Specific (via Semantic Profile)

State Transfer
: Ad hoc (via HTML FORMs)

Application Flow
: Applied (via Semantic Profile)

H-Factors
: LO, LE, LT, LN, CM, CL

Additional Constraints

- All resource representations MUST be valid HTML documents.
- Servers MAY require clients to support HTTP Authentication (BASIC or DIGEST) for some requests.
- Servers MAY provide additional markup or features not covered in this profile, but these additions MUST NOT contradict the semantics outlined here.
- Servers MAY supply code-on-demand elements with their resource representations (JavaScript, CSS, XSLT, etc.), but servers SHOULD NOT assume clients will support them.

Semantic Profile

What follows is a list of HTML attributes and their possible values. Servers SHOULD send resource representations that contain these values along with appropriate markup and data. Servers are free to decide which elements are appropriate for each resource representation. Servers are also free to determine which of the elements below are to be supported.

Clients SHOULD be prepared to properly handle all of the attributes and elements described here. Clients SHOULD also be prepared to provide state transfers (FORMs) back to the server as indicated. Servers MAY provide additional semantics and clients MAY support those additional semantics.

> The phrase "designated user" means: 1) the currently authenticated (logged-in) user; 2) a user identified via other state information such as cookies; or 3) a user selected by some other means such as following a link.

Class Attribute Values

The following `class` values MAY appear within the response representation:

all
: Applied to a UL tag. A list representation. When this element is a descendant of DIV.id="messages" it MAY have one or more LI.class="message" descendant elements. When this element is a descendant of DIV.id="users" it MAY have one or more LI.class="user" descendant elements.

date-time
: Applied to a SPAN tag. Contains the UTC date and time the message was posted. When present, it SHOULD be valid per RFC 3339.

description
: Applied to a SPAN tag. Contains the text description of a user.

friends
: Applied to a UL tag. A list representation. When this element is a descendant of DIV.id="messages" it contains the list of messages posted by the designated user's friends and MAY have one or more LI.class="message" descendant elements. When this element is a descendant of DIV.id="users" it contains the list of users who are the friends of the designated user and MAY have one or more LI.class="user" descendant elements.

followers
: Applied to a UL tag. A list representation of all of the users from the designated user's friends list. MAY have one or more LI.class="user" descendant elements.

me
: Applied to a UL tag. When this element is a descendant of DIV.id="messages" it contains the list of messages posted by the designated user and MAY have one or more LI.class="message" descendant elements. When this element is a descendant of DIV.id="users" it SHOULD contain a single descendant LI.class="user" with the designated user's profile.

mentions
: Applied to a UL tag. A list representation of all of the messages that mention the designated user. It MAY contain one or more LI.class="message" descendant elements.

message
: Applied to an LI tag. A representation of a single message. It SHOULD contain the following descendant elements:

    SPAN.class="user-text"
    A.rel="user"
    SPAN.class="message-text"
    A.rel="message"

  It MAY also contain the following descendant elements:

    IMG.class="avatar"
    SPAN.class="date-time"

message-post
: Applied to a FORM tag. A link template to add a new message to the system by the designated, or logged-in, user. The element MUST be set to FORM.method="post" and SHOULD contain a descendant element:

    TEXTAREA.name="message"

message-reply
: Applied to a FORM tag. A link template to reply to an existing message. The element MUST be set to FORM.method="post" and SHOULD contain the following descendant elements:

    INPUT[hidden].name="user" (the author of the original post)
    TEXTAREA.name="message"

single
: When this element is a descendant of DIV.id="messages" it contains the message selected via a message link. SHOULD have a single LI.class="message" descendant element. When this element is a descendant of DIV.id="users" it contains the user selected via a user link. SHOULD have a single LI.class="user" descendant element.

messages-search
: Applied to a FORM tag. A link template to search all of the messages. The element MUST be set to FORM.method="get" and SHOULD contain the following descendant elements:

    INPUT[text].name="search"

message-text
: Applied to a SPAN tag. The text of a message posted by a user.

search
: Applied to a UL tag. A list representation. When this element is a descendant of DIV.id="messages" it contains a list of messages and MAY have one or more LI.class="message" descendant elements. When this element is a descendant of DIV.id="users" it contains a list of users and MAY have one or more LI.class="user" descendant elements.

shares
: Applied to a UL tag. A list representation of all the messages posted by the designated user that were shared by other users. It MAY contain one or more LI.class="message" descendant elements.

user
: Applied to an LI tag. A representation of a single user. It SHOULD contain the following descendant elements:

    SPAN.class="user-text"
    A.rel="user"
    A.rel="messages"

  It MAY also contain the following descendant elements:

    SPAN.class="description"
    IMG.class="avatar"
    A.rel="website"

user-add
: Applied to a FORM tag. A link template to create a new user profile. The element MUST be set to FORM.method="post" and SHOULD contain the following descendant elements:

    INPUT[text].name="user"
    INPUT[text].name="email"
    INPUT[password].name="password"

  It MAY also contain the following descendant elements:

    TEXTAREA.name="description"
    INPUT[file].name="avatar"
    INPUT[text].name="website"

user-follow
: Applied to a FORM tag. A link template to add a user to the designated user's friend list. The element MUST be set to FORM.method="post" and SHOULD contain the descendant element:

    INPUT[text].name="user"

avatar
: Applied to an IMG tag. A reference to an image of the designated user.

user-text
: Applied to a SPAN tag. The user nickname text.

user-update
: Applied to a FORM tag. A link template to update the designated user's profile. The element MUST be set to FORM.method="post" and SHOULD contain the following descendant elements:

    INPUT[hidden].name="user"
    INPUT[hidden].name="email"
    INPUT[password].name="password"

  It MAY also contain the following descendant elements:

    TEXTAREA.name="description"
    INPUT[file].name="avatar"
    INPUT[text].name="website"

users-search
: Applied to a FORM tag. A link template to search all of the users. The element MUST be set to FORM.method="get" and SHOULD contain the descendant element:

    INPUT[text].name="search"

ID Attribute Values

The following `id` values SHOULD appear within response representations:

messages
: Applied to a DIV tag. The list of messages in this representation. MAY have one or more of the following descendant elements:

    UL.class="all"
    UL.class="friends"
    UL.class="me"
    UL.class="mentions"
    UL.class="search"
    UL.class="shares"
    UL.class="single"

queries
: Applied to a DIV tag. The list of valid queries in this representation. MAY have one or more FORM and/or A descendant elements (see "rel" and "class" section for details).

users
: Applied to a DIV tag. The list of users in this representation. MAY have one or more of the following descendant elements:

UL.class="all"
UL.class="friends"
UL.class="followers"
UL.class="me"
UL.class="search"
UL.class="single"

Name Attribute Values

The following `name` values MAY appear within response representations:

description
: Applied to a TEXTAREA element. The description of the user.

email
: Applied to an INPUT[text] or INPUT[hidden] element. The email address of a user. When supplied, it SHOULD be valid per RFC 5322.

message
: Applied to a TEXTAREA element. The message to post (for the designated user).

name
: Applied to an INPUT[text] element. The full name of a user.

password
: Applied to an INPUT[password] element. The password of the user login.

search
: Applied to an INPUT[text]. The search value to use when searching messages (when applied to FORM.class="message-search") or when searching users (when applied to FORM.class="users-search").

user
: Applied to an INPUT[text] or INPUT[hidden] element. The public nickname of a user.

avatar
: Applied to an INPUT[file] element. The image for the user.

website
: Applied to an INPUT[text]. The URL of a website associated with the user profile. When supplied, it SHOULD be valid per RFC 3986.

Rel Attribute Values

The following `rel` values MAY appear within response representations.

first
: Applied to an A tag. A reference to the first page in a list of data (messages, users).

index
: Applied to an A tag. A reference to the starting URI for the application.

last
: Applied to an A tag. A reference to the last page in a list of data (messages, users).

message
: Applied to an A tag. A reference to a message representation.

message-post
: Applied to an A tag. A reference to the message-post FORM.

message-reply
: Applied to an A tag. A reference to the message-reply FORM.

message-share
: Applied to an A tag. A reference to the message-share FORM.

messages-all
: Applied to an A tag. A reference to a list representation of all the messages in the system.

messages-friends
: Applied to an A tag. A reference to a list representation of all the messages from the designated user's friends list.

messages-me
: Applied to an A tag. A reference to a list representation of all the messages posted by the designated user.

messages-mentions
: Applied to an A tag. A reference to a list representation of all the messages that mention the designated user.

messages-shares
: Applied to an A tag. A reference to a list representation of all the messages posted by the designated user that were shared by other users.

messages-search
: Applied to an A tag. A reference to the messages-search FORM.

next
: Applied to an A tag. A reference to the next page in a list of data (messages, users).

previous
: Applied to an A tag. A reference to the previous page in a list of data (messages, users).

self
: Applied to an A tag. A reference to the currently loaded resource representation.

user
: Applied to an A tag. A reference to a user representation.

user-add
: Applied to an A tag. A reference to the user-add FORM.

user-follow
: Applied to an A tag. A reference to the user-follow FORM.

user-me
: Applied to an A tag. A reference to the designated user's representation.

user-update
: Applied to an A tag. A reference to the user-update FORM.

users-all
: Applied to an A tag. A reference to a list representation of all the users in the system.

users-friends
: Applied to an A tag. A reference to list representation of the designated user's friend users.

users-followers
: Applied to an A tag. A reference to list representation of the users who follow the designated user.

users-search
: Applied to an A tag. A reference to the users-search FORM.

website
: Applied to an A tag. A reference to the website associated with a user.

APPENDIX F
IANA Media Type Registration Document

Below is a completed Internet Assigned Numbers Authority (IANA) media type registration document. This is the result of registering a media type using the IANA's "Application for Media Types" online form (found at *http://www.iana.org/cgi-bin/mediatypes.pl*). This results in the official registration of the `application/vnd.amundsen.maze+xml` media type.

The most recent version of this document can be found online at *http://www.iana.org/assignments/media-types/application/vnd.amundsen.maze+xml*.

```
(last updated 2011-02-01)

Name : Mike Amundsen

Email : mca&amundsen.com

MIME media type name : Application

MIME subtype name : Vendor Tree - vnd.amundsen.maze+xml

Required parameters : none


Optional parameters :
charset
This parameter has identical semantics to the charset parameter of
the "application/xml" media type as specified in RFC 3023.

Encoding considerations : binary
Same as encoding considerations of application/xml as specified in RFC 3023.


Security considerations :
Maze+XML shares security issues common to all XML content types. It does
not provide executable content. Information contained in Maze+XML
documents do not require privacy or integrity services.
```

```
Interoperability considerations :
Maze+XML is not described by a DTD and applies only the well-formedness
rules of XML. It should only be parsed by a non-validating parser.


Published specification :
http://amundsen.com/media-types/maze/


Applications which use this media :
Various


Additional information :

1. Magic number(s) : none
2. File extension(s) : .xml
3. Macintosh file type code : TEXT
4. Object Identifiers: none


Person to contact for further information :

1. Name : Mike Amundsen
2. Email : mca&amunndsen.com

Intended usage : Common
The Maze+XML media type is an XML data format for sharing maze state
information between clients and servers. It can be used to implement
simple mazes, adventure games, and other related data.


Author/Change controller : Mike Amundsen

(file created 2011-02-01)
```

APPENDIX G

IETF Link Relations Internet Draft

Below is a complete Internet Engineering Task Force (IETF) Internet Draft (I-D) for registering Link Relations. This document (which is still a work in progress) was submitted through the IETF's Data Tracker (*http://datatracker.ietf.org/submit/*).

The most recent updates for this I-D can be viewed here: *http://datatracker.ietf.org/doc/draft-amundsen-item-and-collection-link-relations/*

```
Network Working Group                                          M. Amundsen
Internet-Draft                                               October 9, 2011
Intended status: Informational
Expires: April 11, 2012


                    The Item and Collection Link Relations
                draft-amundsen-item-and-collection-link-relations-04

Abstract

   RFC 5988 [RFC5988] standardized a means of indicating the
   relationships between resources on the Web. This specification
   defines a pair of reciprocal link relation types that may be used to
   express the relationship between a collection and its members.

Editorial Note (To be removed by RFC Editor)

   Distribution of this document is unlimited.  Comments should be sent
   to the IETF Apps-Discuss mailing list (see
   <https://www.ietf.org/mailman/listinfo/apps-discuss>).

Status of This Memo

   This Internet-Draft is submitted to IETF in full conformance with the
   provisions of BCP 78 and BCP 79.

   Internet-Drafts are working documents of the Internet Engineering
   Task Force (IETF), its areas, and its working groups.  Note that
   other groups may also distribute working documents as Internet-
   Drafts.
```

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on April 11, 2012.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the

document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the BSD License.

1. Introduction

RFC 5988 [RFC5988] standardized a means of indicating the relationships between resources on the Web. This specification defines a pair of reciprocal link relation types that may be used to express the relationship between a collection and its members.

These link relation types can be applied to a wide range of use cases across multiple media types. For example, the 'collection' and 'item' link relation types are used in these media types:

1. OpenSearch 1.1: see Section 4.5.4.1 of [OpenSearch]

2. Maze+XML: see Section 3 of [Maze]

3. Collection+JSON: see Section 5 of [CollectionJSON]

2. Link Relations

The following link relations are defined.

2.1. 'item'

When included in a resource which represents a collection, the 'item' link relation identifies a target resource that represents a member of that collection.

For example, if a resource represents a catalog of products, that same representation may include one or more links to resources which represent members of that catalog.


```
<html>
...
  <h1>Product Group X Listing</h1>
  ...
  <a href="..." rel="item">View Product X001</a>
  <a href="..." rel="item">View Product X002</a>
  ...
</html>
```

or, in the case of a Link Header field

```
Link: <...>; rel="item"; title="View Product X001"
Link: <...>; rel="item"; title="View Product X002"
```

2.2. 'collection'

When included in a resource which represents a member of a collection, the 'collection' link relation identifies a target resource that represents a collection of which the context resource is a member.

For example, if a resource represents a single product in a catalog, that same representation may include a link to a resource which represents a product group to which this single product belongs:

```
<a href="..." rel="collection">Return to Product Group X</a>
```

or, in the case of a Link Header field

```
   Link: <...>; rel="collection"; title="Return to Product Group X"

   Since it is possible that a resource could be a member of multiple
   collections, multiple 'collection' link relations may appear within
   the same representation:

   <a href="..." rel="collection">View other widgets</a>
   <a href="..." rel="collection">View all discontinued items</a>

   The target resource representation need not be restricted to
   representing a list.  It may simply be a document that provides
   details on the collection of which the context resource is a member:

   Link: <...>; rel="collection";
         title="Shakespeare's Collected Works - A History"
```


3. IANA Considerations

IANA is asked to register the 'collection' and 'item' Link Relations below as per [RFC5988].

3.1. 'item' Link Relation Registration

Relation Name:

item

Description:

The target IRI points to a resource that is a member of the collection represented by the context IRI.

Reference:

See Section 2

3.2. 'collection' Link Relation Registration

Relation Name:

collection

Description:

The target IRI points to a resource which represents a collection of which the context IRI is a member.

Reference:

See Section 2

4. Security Considerations

The two link relation types defined in this document are not believed to introduce any new security issues to those which are discussed in Section 7 of RFC5988 [RFC5988].

5. Internationalisation Considerations

The 'item' and 'collection' link relation types do not have any internationalization considerations other than those which are discussed in Section 8 of RFC5988 [RFC5988].

6. References

6.1. Normative References

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

6.2. Informative References

[OpenSearch] Clinton, D., "Open Search 1.1", Work in Progress , March 2011, <http://www.opensearch.org/Specifications/OpenSearch/1.1/>.

[Maze] Amundsen, M., "Maze+XML - Format", Web Page , December 2010, <http://amundsen.com/media-types/maze/format/>.

[CollectionJSON] Amundsen, M., "Collection+JSON - Document Format", Web Page , July 2011, <http://amundsen.com/media-types/collection/format/>.

Appendix A. Acknowledgements

The author gratefully acknowledges the contributions of Julian Reschke and Mykyta Yevstifeyev.

Author's Address

Mike Amundsen

```
EMail: mca@amundsen.com
URI:   http://amundsen.com
```

APPENDIX H

Source Code, Software, and Installation Notes

Below are notes on the source code contained in this book and the software used in its preparation including some hints on installing CouchDB and Node.js on Ubuntu.

Source Code

A number of source code artifacts were produced in the writing of this book. These include:

- HTML5 markup
- CSS documents
- Client-side JavaScript files
- Server-side JavaScript for Node.js
- JavaScript for CouchDB
- Bash scripts for installing the databases

All of these documents, along with any additional materials related to the book, can be found by visiting the Hypermedia page at amundsen.com.

Prerequisites

The items below were installed to prep Ubuntu for CouchDB and Node.js.

These were installed via the command line:

```
sudo apt-get install curl
sudo apt-get install git
sudo apt-get pkg-config
```

The following were installed using Ubuntu's Software Center:

- python (2.6)
- GNU C++ Compiler
- libssl-dev

CouchDB

Installing CouchDB from the command line was very easy:

```
sudo apt-get install couchdb
```

To test the install, you can bring up your browser and open the Futon helper app:

```
http://localhost:5984/_utils/
```

You can run the CouchDB Test Suite from Futon if you wish. In my case, all tests passed (using FF3.6), but a few scripts run long enough that FF popped up a dialog a few times asking if I wanted to kill the script. I always answered "No" and let the tests continue.

Node.js

I used a script from the Node.js git site (*https://gist.github.com/579814*) called `node-and-npm-in-30-seconds.sh`:

```
echo 'export PATH=$HOME/local/bin:$PATH' >> ~/.bashrc
. ~/.bashrc
mkdir ~/local
mkdir ~/node-latest-install
cd ~/node-latest-install
curl http://nodejs.org/dist/node-latest.tar.gz | tar xz --strip-components=1
./configure --prefix=~/local
make install # ok, this step probably takes more than 30 seconds...
curl http://npmjs.org/install.sh | sh
```

For the record, despite the script's name, it takes more than 30 seconds to build Node.js.

I also installed the following packages using `npm`:

```
npm install express
npm install ejs
npm install cradle
```

Cloud Services

I relied on a handful of cloud services throughout the writing, coding, and testing of the contents of this book. I did this primarily as an experiment to see just how easily I would be able to complete tasks while relying only on the cloud. I must say that I was pleasantly surprised by the sophistication of available services and their overall reliability.

The following services were used in the creation of the book:

GitHub
: I used GitHub's site to host several Git repositories, share code and examples with others, and as a staging area for my online editing (see the next item).

Cloud9 IDE
: This development-as-a-service offering allowed me to grab content from my GitHub repos, edit, debug, and deploy implementations all without the need for a full-featured desktop or laptop machine. I was even able to deliver extended presentations and participate in coding sessions all from my Chromebook device connected to Cloud9's servers.

Joyent Node hosting
: I used Joyent's Node hosting services to run my sample applications on publicly available servers (see "Source Code" on page 217 for more details on how to access publicly available content related to this book). I was also able to deploy my code directly from the Cloud9 IDE to Joyent's servers in a single step. This made for a very enjoyable working experience from start to finish.

Cloudant CouchDB hosting
: I used Cloudant's servers to host my CouchDB data stores on a publicly available server. Cloudant's support includes access to the command line (via CURL) as well as support for the Futon UI to manage data stores. I was able to easily script data store creation and population directly from my workstation, too.

Authoring

I used several machines throughout the writing of this book:

- Meerkat Ion NetTop
- HP HDX X16-1040US Premium Notebook PC
- HP Mini Notebook 1030NR Notebook PC
- Google Chromebook

The software listed below was used to write the book. Since I often switch between physical machines and used two different operating systems in the process (Ubuntu and Windows), the products selected for the task all have installations for multiple platforms:

- XMLMind Editor (for writing the book text)
- gedit (for editing code the examples)
- RapidSVN (Source Control for the book)

Finally lots of notes, research, and initial drafts were written up (and shared) using Google Docs.

About the Author

Mike Amundsen, an internationally known author and lecturer, travels throughout the United States and Europe consulting and speaking on a wide range of topics including distributed network architecture, web application development, cloud computing, and other subjects. His recent work focuses on the role hypermedia plays in creating and maintaining applications that can successfully evolve over time. He has more than a dozen books to his credit and recently contributed to the *RESTful Web Services Cookbook* by Subbu Allamaraju. When he is not working, Mike enjoys spending time with his family in Kentucky, USA.