O'REILLY®

RESTful Web Clients

ENABLING REUSE THROUGH HYPERMEDIA

Mike Amundsen

www.it-ebooks.info

www.it-ebooks.info

RESTful Web Clients

Enabling Reuse Through Hypermedia

Mike Amundsen

Beijing · Boston · Farnham · Sebastopol · Tokyo O'REILLY®

www.it-ebooks.info

RESTful Web Clients
by Mike Amundsen

Copyright © 2016 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://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Meg Foley
Production Editor: FILL IN PRODUCTION EDITOR
Copyeditor: FILL IN COPYEDITOR
Proofreader: FILL IN PROOFREADER
Indexer: FILL IN INDEXER
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2016: First Edition

Revision History for the First Edition
2016-03-21: First Early Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491921906* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *RESTful Web Clients*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-92190-6

[LSI]

www.it-ebooks.info

Table of Contents

1. Our HTML Roots and Simple Web APIs. 7
- The Task Processing System (TPS) Web App 11
  - HTML from the Server 12
  - Common Web Browser as the Client 15
  - Observations 16
- The Task Services Web API 17
  - Web API Common Practice 18
  - Designing the TPS Web API 18
  - Implementing TPS Web API 25
  - Observations 31
- Summary 32
- References 34

2. JSON Clients. 35
- The JSON Web API Client 36
  - Objects 37
  - Addresses 40
  - Actions 42
  - Quick Summary 45
- The JSON SPA Client 45
  - The HTML Container 45
  - The Top-Level Parse Loop 47
  - Objects, Addresses, and Actions 48
- Dealing with Change 53
  - Adding a Field and Filter 54
- Coding a New Client 57
- Summary 59
- References 62

www.it-ebooks.info

3. The Representor Pattern. 63
XML or JSON: Pick a Side! 66
The New Crop of Hypermedia Formats 68
The Fallacy of *The Right One* 70
Re-Framing the Problem 71
The Representor Pattern 73
Separating Format from Functionality 74
The Selection Algorithm 75
A Solid STRATEGY 76
The TRANSFORM VIEW 78
A Server-Side Model 79
Handling the HTTP `Accept` Header 79
Implementing the STRATEGY Pattern 80
General Representor Modules 81
The WeSTL Format 81
A Sample Representor 85
Summary 88
References 90

4. Versioning and the Web. 93
Versioning for the Internet 96
TCP/IP's Robustness Principle 96
HTTP's MUST IGNORE 97
HTML's Backward Compatibility 100
Guidelines for Non-Breaking Changes 101
API Designers 101
Server Implementors 103
Client Implementors 110
Summary 118
References 120

5. Collection+JSON Clients. 121
The Collection+JSON Format 123
Links 126
Items 127
Queries 128
Template 129
Error 131
A Quick Summary 131
The Collection+JSON Representor 132
The Top-Level Processing Loop 132
Links 133

www.it-ebooks.info

Items 134
Queries 136
Template 137
Error 139
The Collection+JSON SPA Client 139
The HTML Container 140
The Top-Level Parse Loop 141
Links 142
Items 144
Queries 146
Template 148
Error 150
Quick Summary 151
Dealing with Change 151
Adding the NOTE Object to the TPS API 152
Extending Collection+JSON 158
Supporting Improved Input Types 159
The `suggest` Object 160
Summary 164
References 167

www.it-ebooks.info

www.it-ebooks.info

CHAPTER 1

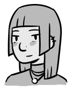
Our HTML Roots and Simple Web APIs

"What's so fascinating and frustrating and great about life is that you're constantly starting over, all the time, and I love that."

—Billy Crystal

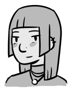
Bob and Carol

"Hello, Carol, I'm Bob. I wanted to stop by to talk."



"Right, Bob. I remember you from the acquisition party last month. Good to see you again. I hear you're helping with the team re-org, right?"

"Right, I just left a meeting with the leadership and they're really excited about the update your team recently released of the Task Progress System. They asked me to come talk to you about taking it to the next level here at BigCo."

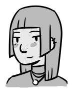


"That's great, Bob. I think the Web app has lots of potential and can help people all over the company better manage their time and resources."

"I agree. So, as a first step, we'd like you to form a new team to focus on the client-side while I take over your group to work exclusively on the server-side."

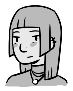
www.it-ebooks.info

“Sounds right, Bob. As long as I get to pick a few people from the current team so we keep some continuity, we’ll be fine.”



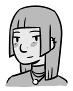
“No problem, you also need to bring on some new client-side developers since that’s a key target for the next phase of the TPS product.”

“So, I guess you’ll be in charge of the switch from HTML-only to a Web Service API, right?”



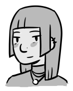
“Yes. I’ll pick of the server-side team and we’ll start work on a stand-alone Web API while you and your new team can focus on the client-side that comsumes the API.”

“There will be some challenges adpating the HTML app into an API but you’ll have a good team behind you, Bob.”



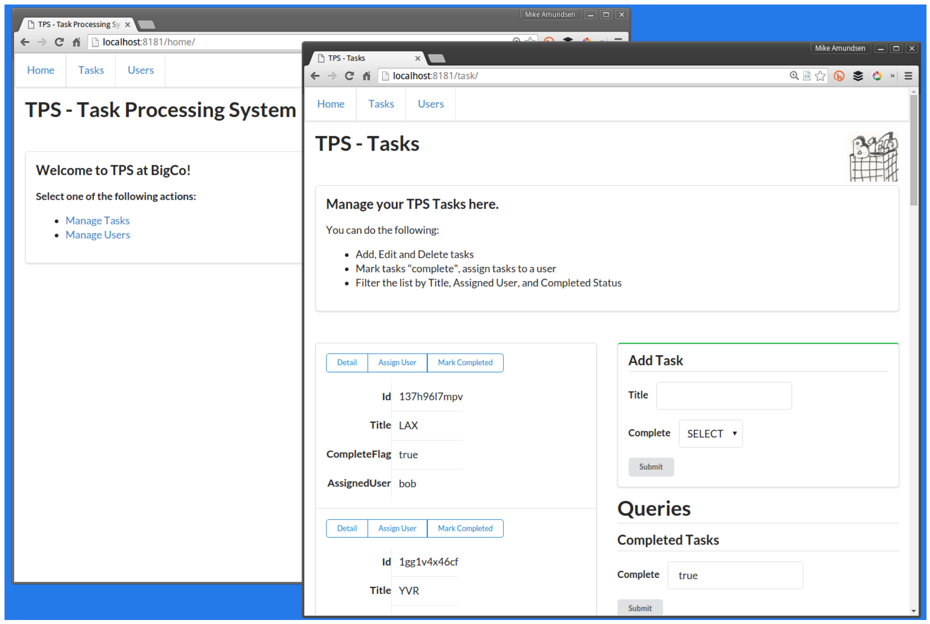
“I hope it’s not too challenging. We’ve got a window of about twelve weeks to pull this all together.”

“Well, I guess we should get started then.”



Before jumping right into the process of creating hypermedia client applications, let’s back up a bit and join the quest earlier on the curve — at the early end of the Web application’s history. Many Web applications began as Web sites — as HTML-based pages that were more than just a list of static documents. In some cases, the initial Web app was a pure HTML app. It had tables of data, forms for filtering and adding data, and lots of links to allow user to transition from one screen to the next.

www.it-ebooks.info



TPS - Task Processing Sy
localhost:8181/home/

Home | Tasks | Users

TPS - Task Processing System

Welcome to TPS at BigCo!

Select one of the following actions:

- Manage Tasks
- Manage Users

TPS - Tasks
localhost:8181/task/

Home | Tasks | Users

TPS - Tasks

Manage your TPS Tasks here.

You can do the following:

- Add, Edit and Delete tasks
- Mark tasks "complete", assign tasks to a user
- Filter the list by Title, Assigned User, and Completed Status

Detail | Assign User | Mark Completed

Id	137h96l7mpv
Title	LAX
CompleteFlag	true
AssignedUser	bob

Detail | Assign User | Mark Completed

Id	1gg1v4x46cf
Title	YVR

Add Task

Title:

Complete: SELECT

Submit

Queries

Completed Tasks

Complete: true

Submit

Figure 1-1. BigCo TPS Screens

One of the things that make pure HTML applications interesting is that they are written in a very *declarative* style. They work without the need for imperative programming code within the client application. In fact, even the visual styling of HTML applications is handled decaratively — via Cascading Style Sheets (CSS). It may seem unusual to attempt to create user experiences using only declarative markup. But it should be noted that many users in the early days of the Web were already family with this interaction style from the mainframe world. In many ways, the early Web applications looked and behaved much like the typical monochrome experiences users had with mainframe, mini-computers and the early personal computers. And this was seen as a "good thing."

www.it-ebooks.info

```
The IBM Personal Computer Basic
Version C1.10 Copyright IBM Corp 1981
62940 Bytes free
Ok
```

1LIST 2RUN 3LOAD" 4SAVE" 5CONT 6,"LPT1 7TRON 8TROFF 9KEY 0SCREEN

Figure 1-2. IBM Portable PC

Typically, at some point in the life of successful HTML-based Web app, someone gets the idea to convert it into a "Web API." There are lots of reasons this comes about. Some want to "unleash the value" locked inside a single application to make the underlying functionality available to a wider audience. There might be new opportunities in other UI platforms (e.g. mobile devices, rich desktop apps, etc.) that don't support a simple HTML+CSS experience. Maybe someone has a new creative idea and wants to try it out. Whatever the reasons, a Web API is born.

Usually, the process is seen as a straight-forward effort to expose an API that covers the internal workings of the existing Web app but without the baggeage of the existing HTML UI. And often the initial work is just that — taking away the UI (HTML) and exposing the data- or object-model already in use within the Web server code as *the* Web API.

Nex, it is assumed, a new team can build a 'better' user interface by consuming the server-side Web API directly from a client application. Often the goal is to build a 'native app' on a smartphone or an advanced Web app using one of the latest client-side frameworks. The first pass at the API is usually pretty easy to understand and building a set of client apps can go smoothly — especially if both the client- and

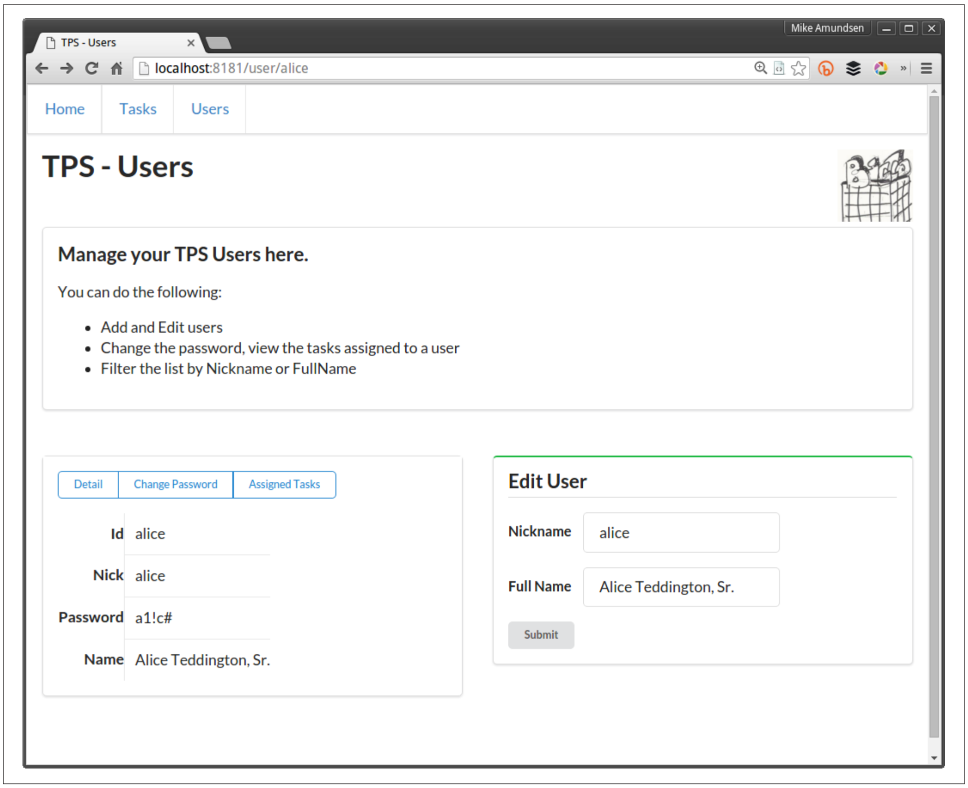
www.it-ebooks.info

server-side imperative code are built by the same team or by teams that share the same deep understanding of the original Web App.

And that's the part of the journey we'll take in this chapter. From HTML to API. That will lay the groundwork for the remainder of the book as we work through the process of building increasingly robust and adaptable client applications powered by the principles and practices of hypermedia.

The Task Processing System (TPS) Web App

For quick review, Carol's team built a Web App that delivers HTML (and CSS) from the Web server directly to common Web browsers. This app works in any brand and version of browser (there are no CSS "tricks", all HTML is standard, and there is no Javascript at all). It also runs quickly and 'gets the job done' by focusing on the key use cases originally defined when the app was first designed and implemented.



TPS - Users

Manage your TPS Users here.

You can do the following:

- Add and Edit users
- Change the password, view the tasks assigned to a user
- Filter the list by Nickname or FullName

Detail | Change Password | Assigned Tasks

Id	alice
Nick	alice
Password	a1!c#
Name	Alice Teddington, Sr.

Edit User

Nickname: alice

Full Name: Alice Teddington, Sr.

Submit

Figure 1-3. TPS User Screen

As you can see from Figure 1, the UI, while not likely to win any awards, is usable, practical, and reliable. All things we wish for in any application.

www.it-ebooks.info

The source code for this version of the TPS can be found in the associated github repo here: *https://github.com/RWCBook/html-only*. A running version of the app described in this chapter can be found here: *http://rwcbook01.herokuapp.com/* (TK: check URLs)

HTML from the Server

Part of the success of the TPS app is that it is very simple. The Web server delivers clean HTML that contains all the links and forms needed to accomplish the required use-cases.

```
<ul>
  <li class="item">
    <a href="https://rwcbook01.herokuapp.com/home/" rel="collection">Home</a>
  </li>
  <li class="item">
    <a href="https://rwcbook01.herokuapp.com/task/" rel="collection">Tasks</a>
  </li>
  <li class="item">
    <a href="https://rwcbook01.herokuapp.com/user/" rel="collection">Users</a>
  </li>
</ul>
```

For example, in the code listing above, you can see the HTML anchor tags (`<a>…</a>`) that point to related content for the current page. This set of 'menu links' appear at the top of each page delivered by the TSP app.

```
<div id="items">
  <div>
    <a href="https://rwcbook01.herokuapp.com/user/alice"
      rel="item" title="Detail">Detail</a> ❶
    <a href="https://rwcbook01.herokuapp.com/user/pass/alice"
      rel="edit" title="Change Password">Change Password</a> ❸
    <a href="https://rwcbook01.herokuapp.com/task/?assignedUser=alice"
      rel="collection" title="Assigned Tasks">Assigned Tasks</a> ❹
  </div>

  <table> ❷
    <tr>
      <th>id</th><td>alice</td>
    </tr>
    <tr>
      <th>nick</th><td>alice</td>
    </tr>
    <tr>
      <th>password</th>
      <td>a1!c#</td>
    </tr>
    <tr>
      <th>name</th><td>Alice Teddington, Sr.</td>
```

www.it-ebooks.info

```
      </tr>
    </table>
  </div>
```

Each user record rendered by the server contains a link pointing to the a single record which consists of a pointer (#1), a handful of data fields (#2), and some links that point to other actions that can be performed for this user record (#3 and #4). These links allow anyone viewing the page to get a detailed view of the record and also initiate updates or password changes (assuming they have the rights to perform these actions).

```
  <!-- add user form -->
  <form method="post" action="https://rwcbook01.herokuapp.com/user/">
    <div>Add User</div>
    <p>
      <label>Nickname</label>
      <input type="text" name="nick" value=""
        required="true" pattern="[a-zA-Z0-9]+" />
    </p>
    <p>
      <label>Full Name</label>
      <input type="text" name="name" value="" required="true" />
    </p>
    <p>
      <label>Password</label>
      <input type="text" name="password" value=""
        required="true" pattern="[a-zA-Z0-9!@#$%^&*-]+" />
    </p>
    <input type="submit"/>
  </form>
```

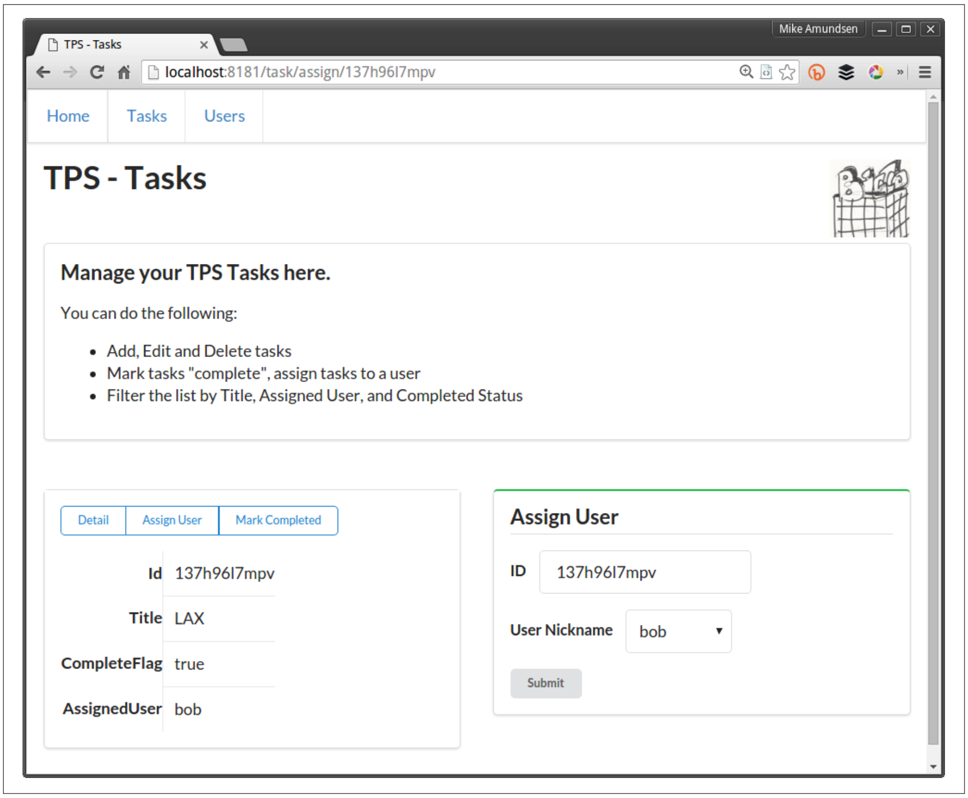
The HTML for adding a user record is also very simple (see above). A clean HTML `<form>` with associated `<label>` and `<input>` elements. In fact, all the input forms in this Web App look about the same. Each `FORM` used for queries (*safe* operations) has the `method` property set to `get` and each `FORM` used for writes (*unsafe* operations) has the `method` property set to `post`, but that is the only important difference in the `FORM` settings for this implementation.

www.it-ebooks.info

A Note about HTML and POST

In HTTP, the POST method defines a non-idempotent, unsafe operation (RFC7231). Some of the actions in the 'TPS' Web app could be handled by an idempotent, unsafe operation but HTML (still) does not support PUT or DELETE (the two idempotent, unsafe operations in HTTP). As Roy Fielding has pointed out in a 2009 blog post ("It is ok to use POST"), it is certainly possible to get everything done on the Web with only GET and POST. But it would be a bit easier if some operations were idempotent since that makes replaying failed requests much easier to deal with. As of this writing, the several attempts to bing PUT and DELETE to HTML have been given a chilly reception.

Along with the typical list, read, add, edit, and remove actions. The TPS web app includes actions like 'Change Password' for users, and 'Assign User' for tasks. Below is the 'Assign User' screen followed by the HTML that drives that screen.



TPS - Tasks

Manage your TPS Tasks here.

You can do the following:

- Add, Edit and Delete tasks
- Mark tasks "complete", assign tasks to a user
- Filter the list by Title, Assigned User, and Completed Status

Id	137h96l7mpv
Title	LAX
CompleteFlag	true
AssignedUser	bob

Assign User

ID: 137h96l7mpv

User Nickname: bob

Figure 1-4. Assign User Screen

www.it-ebooks.info

```
<!-- assign user form -->
<form method="post" action="//rwcbook01.herokuapp.com/task/assign/137h96l7mpv">
  <div>Assign User</div>
  <p>
    <label>ID</label>
    <input type="text" name="id" value="137h96l7mpv" readonly="readonly" />
  </p>
  <p>
    <label>User Nickname</label>
    <select name="assignedUser">
      <option value="">SELECT</option>
      <option value="alice">alice</option>
      <option value="bob" selected="selected">bob</option>
      <option value="carol">carol</option>
      <option value="mamund">mamund</option>
      <option value="ted">ted</option>
    </select>
  </p>
  <input type="submit" />
</form>
```

Note that this form uses the HTTP POST method. Since HTML only provides GET and POST, all unsafe actions (create, update, remove) are enabled using a POST form. We'll have to deal with this later when we convert this HTML-only Web app into a Web API.

Common Web Browser as the Client

The "client-side" of common Web browser applications like this one is pretty uninteresting. First, this app has no client-side javascript dependencies. It runs fine w/o any javascript running locally. The app does take advantage of a handful of HTML5 user experience features such as:

- HTML pattern to perform local input validations
- HTML required to guide the user in filling out important fields
- HTML readonly to prevent users from changing important FORM data

These, along with the use of a SELECT input control to supply users with valid input options, do a pretty good job of providing client-side interaction — all without relying on custom javascript. Some of the screens could be improved with a little client-side script to help hide and show forms, adjust the layout, etc. but to keep things simple in our example, Carol and her team have not yet implemented what is known as the "Unobtrusive Javascript" layer. And, even if they *had* added this enhancement, the app would have no direct dependence on it and would still function without it.

www.it-ebooks.info

Unobtrusive Javascript

The term “Unobtrusive Javascript” is about “the separation of behavior (JavaScript),content (HTML), and presentation(CSS)”. The goal is to create Web applications that function without a fatal dependency on Javascript or CSS. In 2007, the estimate was that up to 10% of browsers had Javascipt disabled. By 2010, stats published buy Yahoo indicated only about 6% of users worldwide had disabled Javascript. I found it difficult to find any more recent data on the subject. While there are developers who still talk about making Unobtrusive Javascript a goal, it is common to encounter Web site and client apps that fail when support for Javascript is turned off.

The CSS styling here is handled by a library called “Semantic UI”. It supports lots of UI design elements while still supporting reasonable HTML markup. Semantic UI libraries also support javascript-driven enhancements that may be used in future updates for this app.

Observations

It turns out that, at least for this Web app, the client-side experience is pretty boring to talk about. There just isn’t much here to cover! That’s actually good news. The common Web browser is designed to accept HTML markup and — based on the responses links and forms (and the added input validations) — provide a solid user experience *without* the requirement of writing imperative javascript code.

Here are a few other observations:

Very Few “Bugs”
: Since there is no custom Javascript code for this client there are almost “no bugs”, either. It is possible that the server will emit broken HTML, of course. And a poorly implemented CSS rule can cause the UI to become unusable. But the fewer lines of code invovled, the less likelihood a bug will be encountered. And this app has *no* imperative client code.

POST-Only Updates
: Since the app is limited to HTML-only responses, all data updates such as create, update, delete, along with the custom events like assign-user, and change-password are handled using HTML `POST` requests. This is, strictly speaking, not a *bug* but it does run counter to the way most Web developers think about actions on the Web and the use of the non-idempotent `POST` action does introduce a slight complication in edge-cases where users are not sure of a `POST` was successful and attempt it a second time.

www.it-ebooks.info

Links and Forms
One of the nice things about using HTML as a response format is that it contains support for a wide range of hypermedia controls: links and forms. The TPS responses include `<a>…</a>` tag to handle simple immutable links, `<form method="get">` elements to handle safe searches and queries, and `<form method="post">` controls to handle all the unsafe write operations. Each response contains all the details for passing arguments to the server and even include simple client-side validation rules to check before the data is sent to the server. Having all this data in the response makes is easy for the browser to enforce specific input rules without having any custom client-side code.

Limited User Experience
Despite the reality of a "bug-free" app, and fully-functional write operations via `POST`, the user experience for this Web App is still very basic. This might be acceptable within a single team or small company, but if BigCo ever plans to release this app to a wider public — even to other teams within the company — a more responsive UX would be a good idea.

So, now that we have a baseline Web App to start from, let's take a look at how BigCo's Bob and Carol can take this app to "the next level" by creating a server-side Web API that can be used to power a stand-alone Web client application.

The Task Services Web API

Often the "next logical step" in the life of an HTML-base Web application is to publish a stand-alone Web API — an *application programming interface* — that can be used by client applications directly. In the dialog at the start of this chapter Bob has taken on the task of leading the server-side team that will design, implement, and publish the 'Task System' API while Carol's team will build the client applications that consume that API.

The source code for the JSON-based RPC-CRUD Web API version of the TPS can be found in the associated github repo here: *https://github.com/RWCBook/json-crud*. A running version of the app described in this chapter can be found here: *http://rwcbook02.herokuapp.com/* (TK: check URLs)

Let's first do a quick rundown on the design process for a typical Web API server followed by a review of the changes needed to convert our existing TPS HTML-only Web app into a proper JSON-based RPC-CRUD Web API.

www.it-ebooks.info

Web API Common Practice

The common practice for creating Web APIs is to publish a fixed set of Remote Procedure Call (RPC) "end points" expressed as URLs that allow access to the important functionality of the original application. This practice also covers the design of those URLs, the serialized objects that are passed between server and client, and a set of guidelines on how to use HTTP methods, status codes, and headers in a consistent manner. For most Web developers today, this is the 'state of the art' for HTTP.

HTTP, REST, and Parkinson's Law

At this point in many discussions, someone starts mention the word 'REST' and a fight (literally or *actually*) may break out between people who want to argue about the 'proper' way to design URLs, which HTTP headers you should *not* use, why it is acceptable to ignore some HTTP Status codes, and so forth. Disputes about the content and meaning of IETF documents specifying the HTTP protocol and disagreements about the shape of URLs are all side-stories to the main adventure: building solid Web applications. Arguing about URLs instead of discussing which interactions are needed to solve a use-case is missing the point. HTTP is just 'tech', 'REST' is just a style (like punk rock or impressionism, etc.). Disagreeing on what is 'true REST' or 'proper HTTP' is a classic cases of Parkinson's Law of Triviality - debating the trivial points while ignoring the important issues.

It turns out designing and implementing reliable and flexible applications that live on the Web *is* non-trivial. It takes a clear head, an eye for the future, and a willingness to spend time engaged in systems-level thinking. Instead of focusing on those hard problems, some get caught up in disagreements on the characters in a URL or other silliness. I don't plan to do that here.

What follows in this chapter is the common practice for HTTP-based Web APIs. It is not, as I plan to illustrate in the ensuing chapters, the *only* way to implement services on the Web. Once we get beyond this particular design and implementation detail we can move on to explore additional approaches.

Designing the TPS Web API

Esentially, we need to *design* the Web API. The common approach is to identify a set of *objects* that will be manipulated via the API and arrange a fixed set of *actions* on those objects. The actions are **Create**, **Read**, **Update**, and **Delete** — the *CRUD* operations. In the case of the TPS example, the list of published objects and actions would look something like that shown in Table 1.

www.it-ebooks.info

Table 1-1. TPS API End Points

URL	Method	Returns Object	Accepts Object
`/task/`	GET, POST	TaskList	Task(POST)
`/task/{id}`	GET,PUT,DELETE	Task	Task(PUT)
`/user/`	GET,POST	UserList	User(POST)
`/user/{id}`	GET,PUT	User	User(PUT)

This looks fairly simple. Four endpoints and about ten operations (we'll handle the 'missing' one in a minute).

There are essentially two forms of the object URL: *list* and *item*. The *list* version of the URL contains the object name (`Task` or `User`) and supports 1) HTTP `GET` to return a list of objects and, 2) HTTP `POST` to create a new object and add it to the list. The *item* version of the URL contains both the object name (`Task` or `User`) and the object's unique `id` value. This URL supports 1) HTTP `GET` to return a single object, 2) HTTP `PUT` to support updating the single object, and 3) HTTP `DELETE` to support removing that object from the collection.

However, there are some 'exceptions' to this simple CRUD approach. Looking at the table, you'll notice that the TPS `User` object does not support the `DELETE` operation. This is a variant to the common CRUD model, but not a big problem. We'd need to document that exception and make sure the API service rejects any `DELETE` request for `User` objects.

Also, the TPS Web App offers a few specialized operations that allow clients to modify server data. These are:

TaskMarkCompleted
: Allows client apps to mark a single `Task` object with the `completeFlag="true"`

TaskAssignUser
: Allows client apps to assign a `User.nick` to a single `Task` object.

UserChangePassword
: Allow client apps to change the `password` value of `User` object.

None of the above operations falls neatly into the CRUD pattern. This complicates the API design a bit. Typically, these special operations are handled by creating a unique URL (e.g. `/task/assign-user` or `/user/change-pw/`) and executing an HTTP `POST` request with a set of arguments to pass to the server.

www.it-ebooks.info

Finally, the TPS Web API supports a handful of filter operations that need to be handled. They are:

TaskFilterByTitle
: Return a list of `Task` objects whose `title` property contains the passed-in string value

TaskFilterByStatus
: Return a list of `Task` objects whose `completeFlag` property is set to "true" (or set to "false")

TaskFilterByUser
: Return a list of `Task` objects whose `assignedUser` property is set to the passed-in `User.nick` value

UserFilterByNick
: Return a list of `User` objects whose `nick` property contains the passed-in string value

UserFilterByName
: Return a list of `User` objects whose `name` property contains the passed-in string value

The common design approach here is to make an HTTP `GET` request to the object's *list* URL (`/task/` or `/user/`) and pass query arguments in the URL directly. For example, to return a list of `Task` objects that have their `completeFlag` set to "true", you could use the following HTTP request: `GET /task/?completeFlag=true`.

So, we have the standard CRUD operations (nine in our case), plus the special operations (three), and then the filter options (five). That's a fixed set of 17 operations to define, document, and implement.

A more complete set of API Design URLs — one that includes the arguments to pass for the write operations (`POST`, and `PUT`) would look like the one in Table 2.

Table 1-2. Complete Set of TPS API End Points

Operation	URL	Method	Returns	Inputs
TaskList	`/task/`	GET	TaskList	none
TaskAdd	`/task/`	POST	TaskList	`title, completeFlag`
TaskItem	`/task/{id}`	GET	Task	none

www.it-ebooks.info

Operation	URL	Method	Returns	Inputs
TaskUpdate	`/task/{id}`	PUT	TaskList	`id`, `title`, `completeFlag`
TaskDelete	`/task/{id}`	DELETE	TaskList	none
TaskMarkComplete	`/task/completed/{id}`	POST	Task	none
TaskAssignUser	`/task/assign/{id}`	POST	Task	`id`, `nick`
TaskFilterByTitle	`/task/?Title={title}`	GET	TaskList	none
TaskFilterByStatus	`/task/?CompleteFlag={status}`	GET	TaskList	none
TaskFilterByUser	`/task/?AssignedUser={nick}`	GET	TaskList	none
UserList	`/user/`	GET	UserList	none
UserAdd	`/user/`	POST	UserList	`nick`, `password`, `name`
UserItem	`/user/{nick}`	GET	User	none
UserUpdate	`/user/{nick}`	PUT	UserList	`nick`, `name`
UserChangePassword	`/user/changepw/{nick}`	POST	User	`nick`, `oldPass`, `newPass`, `checkPass`
UserFilterByNick	`/user/?nick={nick}`	GET	UserList	none
UserFilterByName	`/user/?name={name}`	GET	UserList	none

www.it-ebooks.info

A Note about URL Design

The URLs in the Task System API example are just one of a number of ways to design URLs for a Web API. There are several books (Allamaraju, Masse) which devote pages to the 'proper' way to design a URL for human use. In truth *machines* don't care about the shape of the URL — they only care that it follows the standards for valid URLs (RFC3986) and that they contain enough information for the service to route the request to the right place for processing. In this book, you'll find a wide range of URL designs.

Documenting Data-Passing

By now, you've probably noticed that what we have done here is document a set of remote-procedure calls (RPCs). We've identifed the actions using URLs and listed the arguments to pass for each of them. The arguments are listed in the table, but it's is worth calling them out separately, too. We'll need to share these with API developers so that they know which data element to pass for each request.

Table 1-3. Arguments to Pass for the TPS Web API

Agument Name	Operation(s)
id	TaskItem, TaskUpdate, TaskDelete
title	TaskAdd, TaskUpdate, TaskFilterByTitle
completeFlag	TaskAdd, TaskUpdate, TaskMarkComplete, TaskFilterByStatus
assignedUser	TaskAssignUser, TaskFilterByUser
nick	UserAdd, UserChangePassword, UserFilterByNick
name	UserAdd, UserUpdate, UserFilterByName
password	TaskAdd, TaskChangePassword
oldPass	TaskChangePassword
newPass	TaskChangePassword
checkPass	TaskChangePassword

Notice that the last three arguments in the table (`oldPass`, `newPass`, and `checkPass`) do not belong to any TPS *objects* (e.g. `Task` or `User`). They only exist in order to complete the `UserChangePassword` operation. Usually, RPC-CRUD style APIs restrict

www.it-ebooks.info

data-passing to arguments that belong to some defined object. But, as we've seen already, there are exceptions to this general rule.

Some RPC-CRUD API designs will document an additional set of objects just for passing arguments. I'll not be covering that here, but it *is* an option you may encounter when working with other RPC-CRUD APIs

It is not enough to just document *which* data arguments are passed with each HTTP request. It is also important to document the *format* used to pass arguments from the client to the service. There is no set standard for data-passing with JSON-based APIs, but the typical option is to pass arguments as JSON dictionary objects. For example, the `TaskAdd` operation in Table 2-2 lists two inputs: `title` and `completeFlag`. Using a JSON dictionary to pass this data would look like this:

```
POST /task/ HTTP/1.1
content-type: application/json
...

{
  "title" : "This is my job",
  "completeFlag" : "false"
}
```

Even though the most common way to pass data from client to server on the WWW is using common HTML `FORM` media type (`application/x-www-form-urlencoded`), it is limited to sending simple name-value pairs from client to server. JSON is a bit more flexible than `FORM` data since it is possible to pass aribtarily nested *graphs* of data in a single request. For this implementation, we'll use the typical JSON dictionary approach.

That covers the endpoints, arguments, and format details for sending data from client to server. But there is another important interface detail missing here — the format of the responses. We'll pick that up in the next section.

Serialized JSON Objects

Another important element of this PRC-CRUD style of Web API practice is to identity the format and *shape* of the serialized objects passed from server to client and back again. In the case of the TPS Web API Bob has decided to use simple JSON-serialized objects to pass state back and forth. Some implementations will use nested object trees to pass between parties, but BigCo's serialized objects are rather simple for the moment.

Scanning the **Returns** column of Table 2 you'll notice there are four different return elements defined:

www.it-ebooks.info

1. TaskList
2. Task
3. UserList
4. User

These are the return collections/objects that need to be explicitly defined for API developers. Lucky for us, the TPS Web API has only two key objects as that will make our definition list rather short.

Tables 4 and 5 define the properties for the `Task` and `User` objects in our TPS Web API.

Table 1-4. Task Object Properties

Property	Type	Status	Default
id	string	required	none
title	string	required	none
completeFlag	“true” or “false”	optional	“false”
assignedUser	MUST match User.nick	optional	“”

All fields are defined as `"string"` types. This is just to simplify the implementation of the TPS API for the book. Also, the stored record layout includes `dateCreated` and `dateUpdated` fields that are not listed in our design here. These were left out of the tables for clarity.

Table 1-5. User Object Properties

Property	Type	Status	Default
nick	[a-zA-Z0-9]+	required	none
password	[a-zA-Z0-9!@#$%^&*-]+	required	none
name	string	required	none

For our TPS app, we’ll make things easy and define the `TaskList` and `UserList` return objects simply JSON arrays of the `Task` and `User` objects respectively. Below are examples of each object:

www.it-ebooks.info

```
/* TaskList */
{
  "task": [
    {
      "id": "dr8ar791pk",
      "title": "compost",
      "completeFlag": false,
      "assignedUser": "mamund"
    }
    ... more tasks appear here ...
  ]
}

/* UserList */
{
  "user": [
    {
      "nick": "lee",
      "name": "Lee Amundsen",
      "password": "p@ss"
    }
    ... more user records appear here ...
  ]
}
```

So we've defined the following for our TPS Web API:

- URLs and HTTP methods for each RPC endpoint
- Arguments and format for passing data to the service
- JSON objects returned to the clients

There are a few other implementation details that we'll skip over here (handling errors, HTTP return codes, etc.). These would all appear in a complete documentation set for RPC-CRUD APIs, however. For now, we'll make some assumptions and move on to some implementation details for creating the actually running TPS Web API.

Implementing TPS Web API

We need to make some changes to the existing TPS Web site/app in order to implement our JSON Web API. We don't need to start from scratch (although in some real-life cases that might be the way to go). For our example, we'll just 'fork' the existing implementation to create a new stand-alone codebase that we can alter and turn into a functioning JSON-based RPC-CRUD Web API.

www.it-ebooks.info

The source code for the JSON-based RPC-CRUD Web API version of the TPS can be found in the associated github repo here: *https://github.com/RWCBook/json-crud*. A running version of the app described in this chapter can be found here: *http://rwcbook02.herokuapp.com/* (TK: check URLs)

We have two tasks here. First, we need to modify the TPS Web site to get it to stop emitting HTML and start emitting valid JSON responses. That won't be too tough since the TPS server has some smart tech built in to make *representing* stored data in various media types relatively easy.

We'll dig into the tech for *representing* responses in an upcoming chapter (TK: ref?)

The second task is to add support for all the HTTP requests documented in Table 2-2 above. The good news is *most* of those operations are already supported by the TPS Web site app. We just need to add a few of them (three, actually) and clean up some of the server-side code to make sure we have all the operations working properly.

So, let's get started.

Defaulting to JSON Responses

The TPS Web Site/App emits HTML for all responses. Our Web API will need to change that. Instead of HTML (`text/html`), we will emit JSON (`application/json`) for all responses. Another important change we'll make is to limit the service responses to only send the actual stored `Task` and `User` objects and properties. This will follow along with the information documented in Table 2-2 (Complete set of TPS API End Points) and the details in Tables 2-4 (Task Object Properties) and 2-5 (User Object Properties).

Here is an example of the JSON output from a request to the `/task/` URL:

```
{
  "task": [
    {
      "id": "137h96l7mpv",
      "title": "Update TPS Web API",
      "completeFlag": "true",
      "assignedUser": "bob"
    },
    {
      "id": "1gg1v4x46cf",
      "title": "Review Client API",
```

www.it-ebooks.info

```
        "completeFlag": "false",
        "assignedUser": "carol"
      },
      {
        "id": "1hs5sl6bdv1",
        "title": "Carry Water",
        "completeFlag": "false",
        "assignedUser": "mamund"
      }
      ... more task records here
    ]
  }
```

Note that there are no links or forms in the JSON responses. This is typical for RPC-CRUD style API responses. The URLs and action details are included in the human-readable documentation (TK: ref?) and will be hard-coded into the client application calling this API.

As you would expect, the responses for calls to the `/user/` endpoint look similar to those from the `/task/` URL.

```
  {
    "user": [
      {
        "id": "alice",
        "nick": "alice",
        "password": "a1!c#",
        "name": "Alice Teddington, Sr."
      },
      {
        "id": "bob",
        "nick": "bob",
        "password": "b0b",
        "name": "Bob Carrolton"
      },
      .... more user records here
    ]
  }
```

So, that covers the service responses. Next, we need to make sure all the RPC operations documented in Table 2-2. (TK:check)

Updating the TPS Web API Operations

The TPS Web Site/App supported edit and remove operations via the HTML `POST` method. While this is perfectly fine from an HTML and HTTP point of view, it runs counter to the common practice that jhas grown up around the JSON-based RPC-CRUD pattern. Instead. *edit* operations are handled by the HTTP `PUT` method and *remove* operations are handled by the HTTP `DELETE` operations.

To make our TPS Web API compliant, we need to add two things:

www.it-ebooks.info

1. Support for PUT and DELETE on /task/{id} URLs
2. Support for PUT on the /user/{nick} URLs.

Since the TPS service *already* supports the actions of 'update' and 'remove' for Tasks (and 'update' for Users), the only thing we need to add to the service code is support for executing those actions via HTTP PUT and DELETE. A quick look at the code from our TPS server (with the functionality updated is below:

```
...
case 'POST':
  if(parts[1] && parts[1].indexOf('?')===-1) {
    switch(parts[1].toLowerCase()) {
      /* Web API no longer supports update and remove via POST ❶
      case "update":
        updateTask(req, res, respond, parts[2]);
        break;
      case "remove":
        removeTask(req, res, respond, parts[2]);
        break;
      */
      case "completed":
        markCompleted(req, res, respond, parts[2]);
        break;
      case "assign":
        assignUser(req, res, respond, parts[2]);
        break;
      default:
        respond(req, res,
          utils.errorResponse(req, res, 'Method Not Allowed', 405)
        );
    }
  }
  else {
    addTask(req, res, respond);
  }
break;
/* add support for update via PUT */ ❷
case 'PUT':
  if(parts[1] && parts[1].indexOf('?')===-1) {
    updateTask(req, res, respond, parts[1]);
  }
  else {
    respond(req, res,
      utils.errorResponse(req, res, 'Method Not Allowed', 405)
    );
  }
break;
/* add support for remove via DELETE */ ❸
case 'DELETE':
  if(parts[1] && parts[1].indexOf('?')===-1) {
```

www.it-ebooks.info

```
      removeTask(req, res, respond, parts[1]);
    }
    else {
      respond(req, res,
        utils.errorResponse(req, res, 'Method Not Allowed', 405)
      );
    }
  break;
  ...
```

As you can see from the code snippet above, the HTTP handler for `Task` data no longer supports the 'update' and 'remove' actions via `POST` (#1). They are now accessed via HTTP `PUT` (update) and `DELETE` (remove). A similar change was made to support 'update' for `User` data, too.

To be complete, the Web API service should also be updated to no longer serve up the `assignUser`, `markCompleted`, and `changePassword` pages. These were provided by the TPS Web site/app to allow users to enter data via HTML `FORMS`. Since our Web API doesn't support `FORMS`, we don't need these pages anymore.

Below is the TPS Web API `Task` handler with the `assignUser` and `markCompleted` `FORM` pages turned off:

```
  ....
  case 'GET':
    /* Web API no longer serves up assign and completed forms
    if(flag===false && parts[1]==="assign" && parts[2]) {
      flag=true;
      sendAssignPage(req, res, respond, parts[2]);
    }
    if(flag===false && parts[1]==="completed" && parts[2]) {
      flag=true;
      sendCompletedPage(req, res, respond, parts[2]);
    }
    */
    if(flag===false && parts[1] && parts[1].indexOf('?')===-1) {
      flag = true;
      sendItemPage(req, res, respond, parts[1]);
    }
    if(flag===false) {
      sendListPage(req, res, respond);
    }
  break;
  ....
```

Testing the TPS Web API with cURL

While we need a fully-functioning JSON CRUD client to test all of the TPS Web API, we can still run some basic tests using the `curl` command-line utility. This will con-

www.it-ebooks.info

firm that we have set up the TPS Web API correctly (per the API design above) and allow us to do some simple interactions with the running API service.

Below is a short `curl` session that shows running all the CRUD operations on the `Task` endpoint as well as the `TaskMarkCompleted` special operation.

To save space and stay within the page layout some of the command lines are printed on *two* lines. If you are running these commands yourself, you'll need to place each command on a single line.

```
// create a new task record
curl -X POST -H "content-type:application/json" -d '{"title":"testing"}'
  http://localhost:8181/task/

// fetch the newly created task record
curl http://localhost:8181/task/1z4yb9wjwi1

// update the existing task record
curl -X PUT -H "content-type:application/json" -d '{"title":"testing again"}'
  http://localhost:8181/task/1z4yb9wjwi1

// mark the record completed
curl -X POST -H "content-type:application/json"
  http://localhost:8181/task/completed/1z4yb9wjwi1

// delete the task record
curl -X DELETE -H "content-type:application/json"
  http://localhost:8181/task/1z4yb9wjwi1
```

So, we've made the implementation changes need to get the TPS Web API up and running:

- Set the API responses to all emit simple JSON (`application/json`) arrays
- Added support for `PUT`(update) and `DELETE`(remove) for `Task` objects
- Removed support for `POST`(update) and `POST`(remove) for `Task` objects
- Removed support for `GET`(assignUser) and `GET`(markCompleted) FORMS for `Task` objects
- Added support for `PUT`(update) for `User` objects
- Removed support for `POST`(update) for `User` objects
- Removed support for `GET`(changePassword) FORMS for `User` objects

As you can see from the list, we actually did more to *remove* support in the Web API than anything else. Rember that we also removed all the links and forms from the

www.it-ebooks.info

Web API responses. The work of knowing what it takes to filter and modify data on the TPS service will now need to be coded into the JSON client application. We'll see how that works in the next chapter.

Observations

Now that we have a working TPS Web API service up and running, its worth making a few observations on the experience.

Plain JSON Responses
A hallmark of Web APIs today is to emit 'plain JSON' responses. No more HTML, not even XML — just JSON. The advantage is that supporting JSON in browser clients Javascript is easier than dealing with XML or parsing HTML responses. Although we didn't get to see it in our simple example, JSON response can carry a large nest graph of data more efficiently than HTML, too.

API Design is All About URLs and CRUD
When we were designing out Web API we spent most of the time and effort crafting URLs and deciding on which methods and arguments to pass for each request. We also needed make sure the exposed URLs map the Create-Read-Update-Delete (CRUD) semantics against important JSON objects. There were a few actions (three for our use case) that didn't map well to CRUD and we had to create special URLs for them, too.

No More Links and Forms
Another common feature of Web APIs is the lack of links and forms in responses. Common Web browsers use the links and forms in HTML responses to render a user interface for humans to scan and activate. This works because the browser already *understands* links and forms in HTML. Since JSON doesn't have things like `<a>…</a>` and `<form method="get">` or `<form method="post">`, the information needed to execute actions from the UI will need to be baked into the API client code.

API Servers are Rather Easy
Since most of what we did to make our TPS Web app into a Web API is *remove* features, it seems building API servers is relatively easy to do. There are certainly challenges to it — our TPS Web API is pretty simple — but for the most part, we have less things to decide when creating these RPC-CRUD style APIs than when we are creating both the data responses *and* the UI rendering instructions from the standard Web site/app.

Completing the API is Only Part of the Story
We found out that once you have the Web API up and running, you still need to test it with some kind of client. We can't just point a Web browser at the API since browsers don't know about our CRUD and special operations. For now, we

www.it-ebooks.info

used the `curl` command-line utility to execute HTTP-level requests against the API to make sure it was behaving as expected. We still need to build a fully-functional JSON CRUD client before we'll know for sure, though.

Summary

In this chapter, we started our journey toward hypermedia clients by first stepping back a bit and reviewing a kind of 'early history' of typcial Web APIs — especially their roots in simple HTML-only Web site/apps. We were introduced to BigCo's Task Processing System (TPS) Web app and learned that the HTML5 app worked just fine without any javascript code at all. Sure, it was a bit limited but all the functionality needed was available in a responsive, visually acceptable (not stunning) UI.

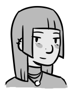
But we're interested in API services and API clients. So the first task was to convert this simple HTML-Only Web app into a pure JSON Web API service. And it was not too tough, either. We adopted the common RPC-CRUD design model by establishing a key URL for each API object (`Task` and `User`) and implementing the Create-Read-Update-Delete (CRUD) pattern against these objects and thier URLs. We had to create a few other special URLs to support unique operations (using `POST`) and documented a set of filter routines agains the Web API's 'collection' URLs (`/task/` and `/user/`). We then documented the JSON objects that were returned and established that all payloads sent from client to server should be formatted as JSON dictionary objects.

With the design completed, we needed to actually implement the API. We were able to 'fork' the existing Web site app and spent most of our efforts *removing* functionality, simplifying the format (we dropped all the links and forms in the JSON responses) and cleaning up the Web API code. Finally, we used the `curl` command-line utility to confirm our API was functioning as expected.

This gives us a great start on our TPS API service. The next challenge is building a fully-functional JSON CRUD client that understands the TPS Web API documentation. Since we spent a lot of our time eliminating information in the Web API responses, it seems likely, we'll need to add that information to the API client instead.

We'll take on that challenge in a future chapter.

Bob and Carol



So, Bob. Do you have the TPS Web API up and running yet?

www.it-ebooks.info

Actually, Carol, I do. It was not as hard as I thought to implement, but did take a bit more designing than I'd planned.

Right, this was what I thought would happen. With the links and forms no longer in the responses, you needed to re-design the API to support the CRUD pattern, right?

Yep. The Create-Read-Update-Delete pattern is what most JSON API developers expect. But we still had to support some other non-CRUD actions like our `Assign User', `Mark Completed', and `Change Password' use-cases.

Yeah, not all actions will fit into the pattern. So, I assume you've got some serious documentation for me now, too.

Well, the docs are not too big -- It's a simple app right now. But I've got a list of all the URLs, HTTP methods, payloads, arguments, and return object for you. You'll need to hard-code all that into your app now that we don't have any links or forms in the response to carry that information.

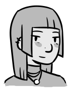
Oh, now I see what you meant when you said the implementation was not so hard. Your team took a bunch of stuff out of the server-side API and *my* team need to spend time putting it all back in on the client side.

Well, that's one way to look at it, I guess. Hadn't tought about it like that before.

No worries, Bob. So, you set for my team to start working on the JSON API client for the TPS API?

Yes, and the sooner the better. I can't use a browser to test this API anymore. The command-line utilities work but I need a fully functional client to help me figure out if there is anything we missed.

www.it-ebooks.info


OK, Bob. We'll get right on it and meet you back here in a few days.

References

1. The monochrome computer screen image is a photo of an IBM Portable PC from Hubert Berberich (HubiB) (Own work) [CC BY-SA 3.0 (*http://creativecommons.org/licenses/by-sa/3.0*)], via Wikimedia Commons
2. Roy Fielding's 2009 blog post It is ok to use POST points out that his dissertation never mentions CRUD and that it is fine to use GET and POST for Web apps.
3. The book *Unobtrusive Ajax* (2007) by Jesse Skinner was published by O'Reilly as part of their "Short Cut" series.
4. A Yahoo blog post *How many users have Javascript disabled?* (2010) shows that only about 2% of US users were visiting Yahoo properties with scripting turned off.
5. Two books I recommend when looking for guides in URL design are *RESTful Web Services Cookbook* (2010) by Subbu Allamaraju and *REST API Design Rulebook* (2011) by Mark Masse. There are quite a few more, but these are the books I find I use often.
6. Parkinson's Law of Triviality is sometimes referred to as "bikeshedding" was first described by C. Northcut Parkinson in his book *Parkinson's Law – and other studies in administration* (1957). When referring to the case of committees working through a busy agenda, Parkinson observed "The time spent on any item of the agenda will be in inverse proportion to the sum [of money] involved."

www.it-ebooks.info

CHAPTER 2

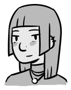
JSON Clients

"All my books started out as extravagant and ended up pure and plain."

—Annie Dillard

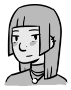
Bob and Carol

"OK, Carol. The TPS Web API is up and running and ready for your team to create a client."



"Sounds good, Bob. I see your team provided the API documentation, too. Wow, that seems like a lot of documentation for such a small API!"

"Well, when you get down to it, there are lots of details to work out when writing your client app. The objects to deal with, all the URLs. and even all the parameters for adding and updating records -- that's all part of the docs."



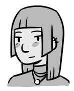
"Right. No problem. I hadn't thought about that level of detail. I thought the CRUD pattern would make most of this easy."

"Even though we used the Cread-Read-Update-Delete pattern where we could, not all the API actions fit neatly into those four actions. You'll see

www.it-ebooks.info

a couple things where we just use HTTP POST with arguments."

"That's fine. I'll to discuss this with the team to make sure we account for it all and still keep the client app small and limit the added complexity. Hopefully we can get our first client app into production by the end of next week."



"Ok, Carol. Keep me posted and we'll get together next week."

OK, now that we have a fully-functionl JSON Web API, we're ready to build the client apps. Since the Web API we're targeting is an RPC-CRUD API, we'll need to consult the documentation and be sure to build into the app all the rules for constructing URLs, handling unique response objects/collections, and knowing the full details on how to execute all the operations (almost twenty of them in this simple app) for filtering, displaying, and modifying service data.

After building and releasing our API client, we'll update the backend service and see how that affects our client in production. Ideally, we'd like that client to 'just work' and assume any new features of the updated backend. But, anyone who has built these RPC-CRUD clients knows that's not at all likely. At least we'd like the app to *not crash* when the backend changes and even that is an iffy proposition. We'll work through the changes needed to keep our Web API client up-to-date and close the chapter with some observations before we move on to our next project.

So, let's get started.

The JSON Web API Client

For many readers, the typical JSON Web API Client is nothing new — its the style that most Web APIs are designed to support right now. We'll review some of the basic elements of this client and then, after a short detour into the service code to explore JSON API output, we'll work through the coding needed to create a fully-functional JSON Web API client. Along the way we'll learn how the JSON client needs to handle important elements such as:

- Recognizing the OBJECTS in responses
- Constructing ADDRESSES (the URLs) for interacting with the service

www.it-ebooks.info

- Handling ACTIONS such as filtering, editing or deleting data

The OAA Challenge

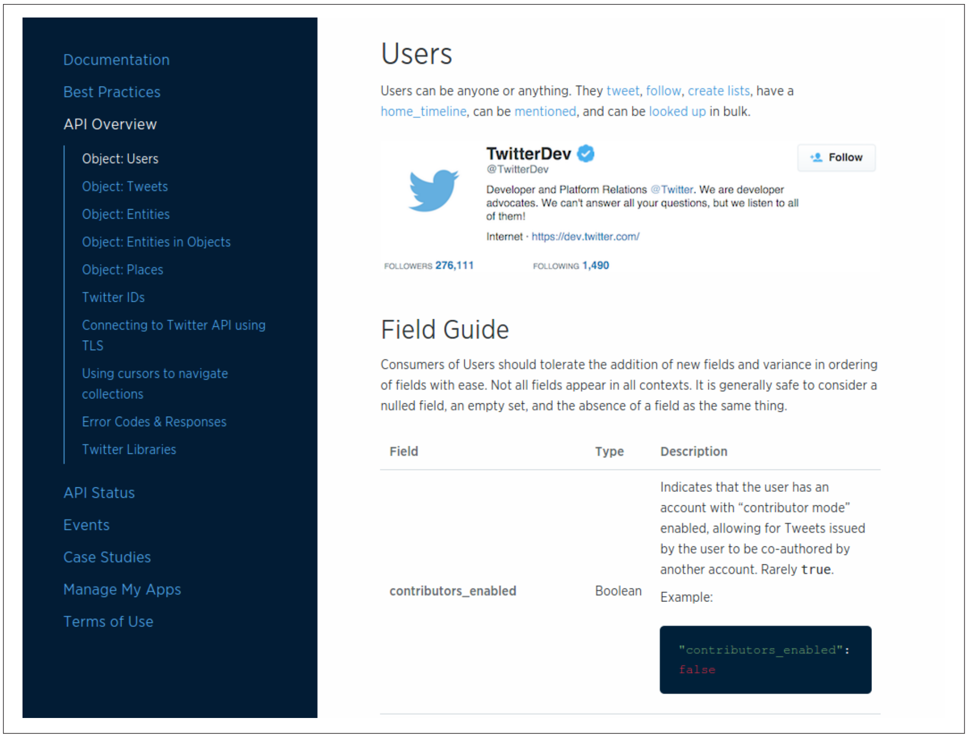
Throughout the book I'll refer to this as the OAA Challenge (as in OBJECTS, ADDRESSES, and ACTIONS). We'll see that every Web API client app needs to deal with them and, especially when we start looking at the hypermedia-style clients, there are varying ways to handle this challenge.

Let's take a minute to review each of these three elements before starting to look at the way client apps deal with the.

Objects

One of the most important things that JSON Web API clients need to deal with are the JSON objects that appear in responses. Most JSON Web APIs expose a unique object model via the responses. Before you can even *start* using the API in any important way, you need to know (and support) the object model. Sometimes, this can get rather elaborate, too.

www.it-ebooks.info


Documentation
Best Practices
API Overview
Object: Users
Object: Tweets
Object: Entities
Object: Entities in Objects
Object: Places
Twitter IDs
Connecting to Twitter API using TLS
Using cursors to navigate collections
Error Codes & Responses
Twitter Libraries
API Status
Events
Case Studies
Manage My Apps
Terms of Use

Users

Users can be anyone or anything. They tweet, follow, create lists, have a home_timeline, can be mentioned, and can be looked up in bulk.

TwitterDev
@TwitterDev
Follow
Developer and Platform Relations @Twitter. We are developer advocates. We can't answer all your questions, but we listen to all of them!
Internet · https://dev.twitter.com/
FOLLOWERS 276,111 FOLLOWING 1,490

Field Guide

Consumers of Users should tolerate the addition of new fields and variance in ordering of fields with ease. Not all fields appear in all contexts. It is generally safe to consider a nulled field, an empty set, and the absence of a field as the same thing.

Field	Type	Description
contributors_enabled	Boolean	Indicates that the user has an account with "contributor mode" enabled, allowing for Tweets issued by the user to be co-authored by another account. Rarely `true`. Example: `"contributors_enabled": false`

Figure 2-1. Screenshot of Twitter's JSON Object Documentation

As of this writing, the Twitter(c) API Overview page lists five baseline objects:

- Users
- Tweets
- Entities
- Entities in Objects
- Places

Many of these objects contain nested dictionary and array objects as well. And there are several complete sets of JSON objects for the Twitter API for their streaming service, Ad service, and others.

Recognizing Objects

Lucky for us, the TPS Web API has only two main objects (`Task` and `User`) and each of them are just a set of name-value pairs. This simple design makes our sample apps easy to work with and explore. However, most non-trivial production apps are likely to have several objects and tens (possibly hundreds) of properties.

www.it-ebooks.info

Currently, the TPS Web API responses are simple arrays:

```
{
  "task": [
    {
      "id": "137h96l7mpv",
      "title": "LAX",
      "completeFlag": "true",
      "assignedUser": "bob",
      "dateCreated": "2016-01-14T17:48:42.083Z",
      "dateUpdated": "2016-01-27T22:03:02.850Z"
    },
    {
      "id": "1gg1v4x46cf",
      "title": "YVR",
      "completeFlag": "false",
      "assignedUser": "carol",
      "dateCreated": "2016-01-14T18:03:18.804Z",
      "dateUpdated": "2016-01-27T17:45:46.597Z"
    },
    .... more TASK objects here
   ]
 }
```

Our client app will need to recognize the `"task"` array name and act accordingly at runtime. One good way to do this is to use the object identifier as a *context switch*. When our app 'sees' `"task":[…]` in the response, it will switch to 'task-mode' and display the data (and possible actions) related to tasks. When the server response contains `"user":[…]` the app can switch to 'user-mode'. Of course, our app won't know what to do if a service response contains `"note ":[…]` or some other unknown context value. For now, we'll need to ignore anything we don't recognize.

Displaying Data

And just knowing the objects and their properties is not enough. Client applications also need to know how to deal with them when they "show up" in an API response. For example, whether to show the data, which properties to display, the human prompts associated with the data, etc.

For example, the `Task` object emitted by the TPS Web API looks like this:

```
{
  "id": "137h96l7mpv",
  "title": "Review API Design",
  "completeFlag": "false",
  "assignedUser": "bob",
  "dateCreated": "2016-05-14T17:48:42.083Z",
  "dateUpdated": "2016-05-27T22:03:02.850Z"
},
```

www.it-ebooks.info

For our client app, we've decided to *not* display the `dateCreated` and `dateUpdated` fields. In fact, we'll need to keep track of which fields to hide and which to show for *all* the TPS objects.

We also need to decide which prompts to display for each property in the TPS `Task` and `User` objects. Most of the time, client-side developers need to keep an internal set of prompts (possibly even tagged for more than one language) and map those prompts to the property names at runtime. For our simple app, we'll just use a CSS trick to capitalize the property names when displaying them as prompts (see callout #1 below).

```
span.prompt {
  display:block;
  width:125px;
  font-weight:bold;
  text-align:right;
  text-transform:capitalize; ❶
}
```

This works because we only need to support a single language and we're just working with a simple demo app. Production apps will need more attention to detail on this point.

So, our client app will keep track of all the important JSON objects coming from the server, will know how to "handle" each of them, and will know which properties to display and what prompts are associated with them.

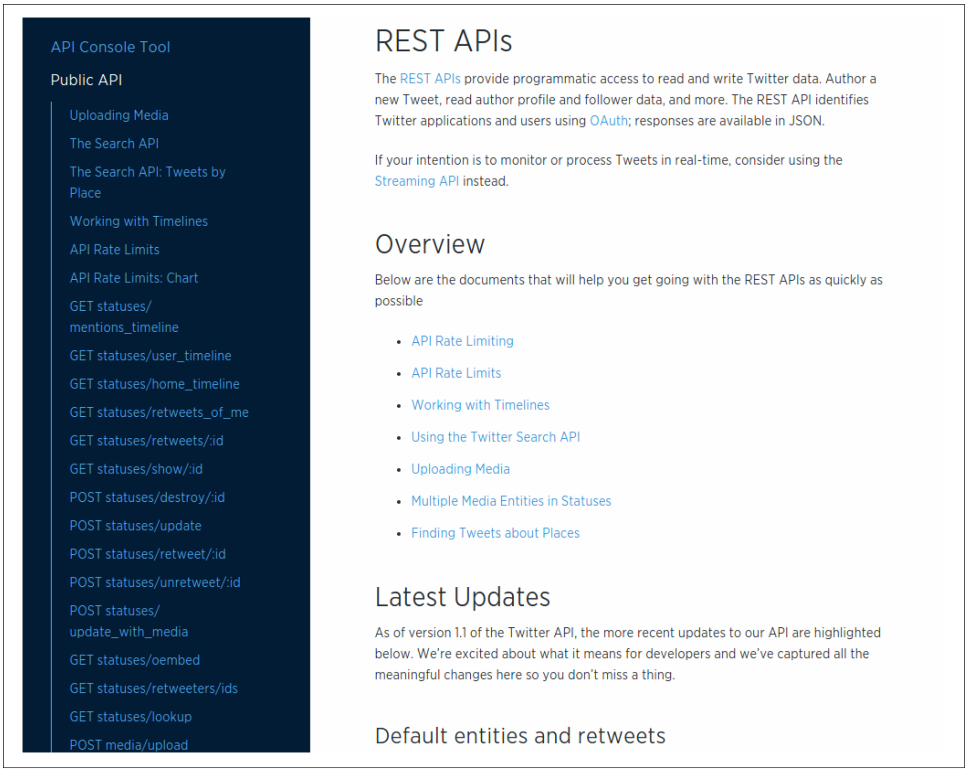
The next thing to deal with is the object's *addresses* — their URLs.

Addresses

Most JSON RPC-CRUD API responses don't include URLs — the addresses of the objects and arrays the client application is processing. Instead, the URL information is written up in human-readable documentation and it is up to the developer to work out the details. Often this involves hard-coding URLs (or URL templates) into the app, associating those addresses with objects and collections at runtime, and resolving any parameters in the URL templates before actually *using* them in the app.

The number of URLs and templates in an API can be very large. For example, using the Twitter API (mentioned above) as an example, there are close to 100 URL endpoints displayed on just one page of the Twitter API documentation (see screenshot below).

www.it-ebooks.info


API Console Tool

Public API

Uploading Media
The Search API
The Search API: Tweets by Place
Working with Timelines
API Rate Limits
API Rate Limits: Chart
GET statuses/mentions_timeline
GET statuses/user_timeline
GET statuses/home_timeline
GET statuses/retweets_of_me
GET statuses/retweets/:id
GET statuses/show/:id
POST statuses/destroy/:id
POST statuses/update
POST statuses/retweet/:id
POST statuses/unretweet/:id
POST statuses/update_with_media
GET statuses/oembed
GET statuses/retweeters/ids
GET statuses/lookup
POST media/upload

REST APIs

The REST APIs provide programmatic access to read and write Twitter data. Author a new Tweet, read author profile and follower data, and more. The REST API identifies Twitter applications and users using OAuth; responses are available in JSON.

If your intention is to monitor or process Tweets in real-time, consider using the Streaming API instead.

Overview

Below are the documents that will help you get going with the REST APIs as quickly as possible

- API Rate Limiting
- API Rate Limits
- Working with Timelines
- Using the Twitter Search API
- Uploading Media
- Multiple Media Entities in Statuses
- Finding Tweets about Places

Latest Updates

As of version 1.1 of the Twitter API, the more recent updates to our API are highlighted below. We're excited about what it means for developers and we've captured all the meaningful changes here so you don't miss a thing.

Default entities and retweets

Figure 2-2. Screenshot of Twitter's JSON API Documentation

While it is likely a single API client will not need to handle *all* 97 of the URLs listed on that page, any client wanting to do more than one or two things will need to deal with dozens of them. And, as I mentioned before, this list does not include the URLs for Twitter's streaming API, their Ad API, or other API sets they offer.

For the TPS Web API, there are 17 URLs and templates to deal with. They're listed in Table 2 of chapter XXX. (TK:fix). We'll need to sort out which addresses belong to each context object (`Task` or `User`) and which of them are more than simple actions (e.g. HTTP POST, PUT, and DELETE actions).

There are many different ways of handling URLs and templates for JSON Web APIs. The approach I'll use in our sample app is to create a JSON dictionary of all the URLs for each object. For example, the code below shows how I'll 'memorize' some of the `Task` operations. Note I included a `prompt` element for use when displaying these URLs as links in the client UI.

```
actions.task = {
  tasks:   {href:"/task/", prompt:"All Tasks"},
  active:  {href:"/task/?completeFlag=false", prompt:"Active Tasks"},
```

www.it-ebooks.info

```
      closed:  {href:"/task/?completeFlag=true", prompt:"Completed Tasks"},
    }
```

I'll also need to keep some information on *when* to display links. Should they appear on every page? Just on the pages associated with Tasks? Only when a *single* Task is displayed? My solution is to use an additional property of my address list called `target` which is set to values such as `"all"` or `"list"` or `"single-item"`, etc. We'll see that a bit later in this chapter.

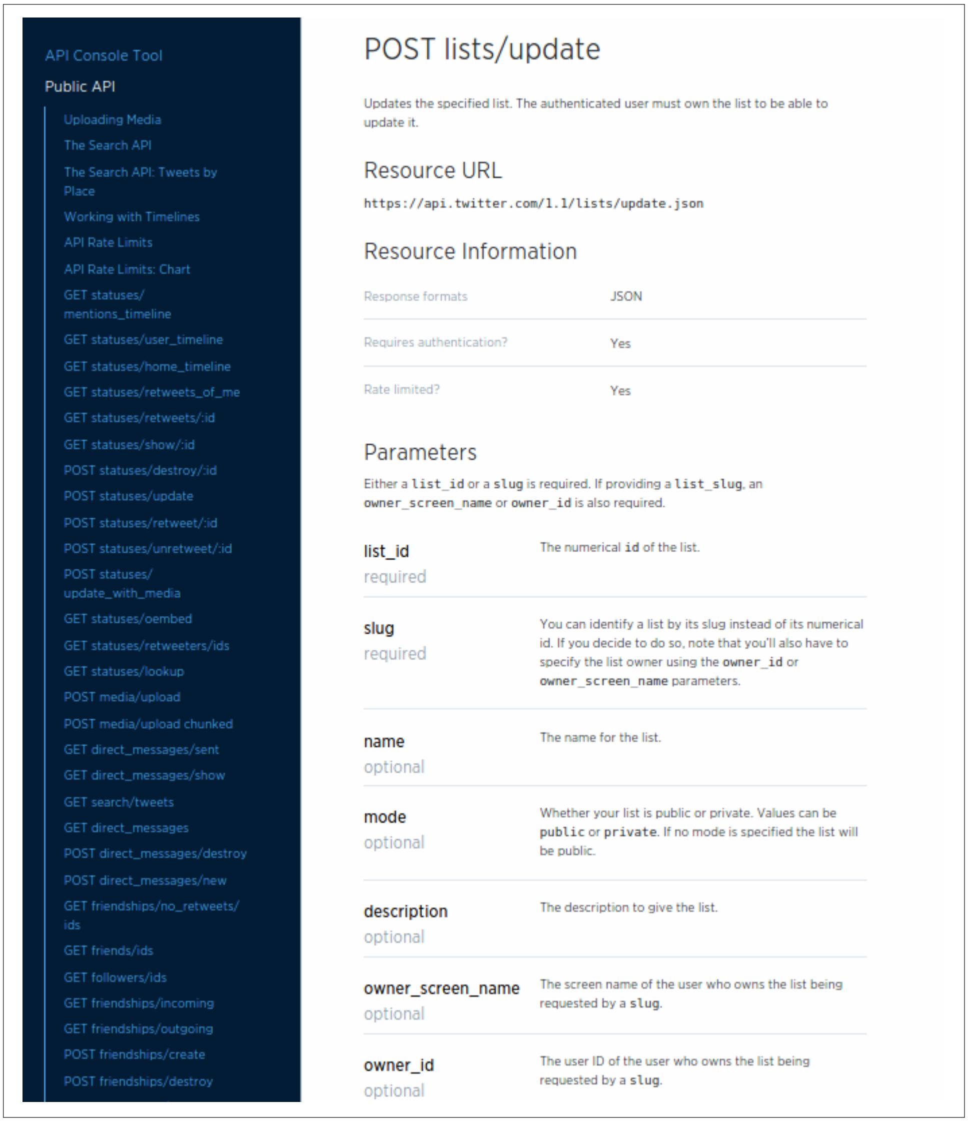
So, objects and addresses. That's pretty good, but that's not enough. We also need to know the details on actions that involve query parameters and actions that require construction of request bodies (e.g. `POST`, and `PUT`).

Actions

The third important element that Web API clients need to deal with are actions that include query parameters or require HTTP bodies — the filters and write operations of a Web API. Just like the URLs, this information is written up in human-readable documentation and needs to be translated into client-side code.

The common approach in documentation is to list the URL, the HTTP method, and parameters that can be passed including any validation rules for the inputs, etc. Keeping with the Twitter theme, Twitter's API for updating existing lists takes up to seven parameters and looks like this in the documentation:

www.it-ebooks.info



API Console Tool

Public API

- Uploading Media
- The Search API
- The Search API: Tweets by Place
- Working with Timelines
- API Rate Limits
- API Rate Limits: Chart
- GET statuses/mentions_timeline
- GET statuses/user_timeline
- GET statuses/home_timeline
- GET statuses/retweets_of_me
- GET statuses/retweets/:id
- GET statuses/show/:id
- POST statuses/destroy/:id
- POST statuses/update
- POST statuses/retweet/:id
- POST statuses/unretweet/:id
- POST statuses/update_with_media
- GET statuses/oembed
- GET statuses/retweeters/ids
- GET statuses/lookup
- POST media/upload
- POST media/upload chunked
- GET direct_messages/sent
- GET direct_messages/show
- GET search/tweets
- GET direct_messages
- POST direct_messages/destroy
- POST direct_messages/new
- GET friendships/no_retweets/ids
- GET friends/ids
- GET followers/ids
- GET friendships/incoming
- GET friendships/outgoing
- POST friendships/create
- POST friendships/destroy

POST lists/update

Updates the specified list. The authenticated user must own the list to be able to update it.

Resource URL

`https://api.twitter.com/1.1/lists/update.json`

Resource Information

Response formats	JSON
Requires authentication?	Yes
Rate limited?	Yes

Parameters

Either a `list_id` or a `slug` is required. If providing a `list_slug`, an `owner_screen_name` or `owner_id` is also required.

list_id required	The numerical **id** of the list.
slug required	You can identify a list by its slug instead of its numerical id. If you decide to do so, note that you'll also have to specify the list owner using the `owner_id` or `owner_screen_name` parameters.
name optional	The name for the list.
mode optional	Whether your list is public or private. Values can be `public` or `private`. If no mode is specified the list will be public.
description optional	The description to give the list.
owner_screen_name optional	The screen name of the user who owns the list being requested by a `slug`.
owner_id optional	The user ID of the user who owns the list being requested by a `slug`.

Figure 2-3. Screenshot of Twitter's Update List API Documentation

The documentation for our TPS Web API appears in Table 2 in chapter XX(TK:fix). Similar to the Twitter API documentation, the TPS docs show the URL, the method, and the set of parameters. This all needs to be "baked" into the Web API client app, too.

There are many different ways to encode action information into client apps; from hard-coding it into the app, keeping it as meta-data in a separate local file, or even as

www.it-ebooks.info

a remote configuration file sent with the JSON response. For our TPS client, I'll use an approach similar to the one I used when handling simple URLs (see section above).

For example, the `UserAdd` action looks like this in the TPS documentation:

Table 2-1. TPS UserAdd Action

Operation	URL	Method	Returns	Inputs
UserAdd	`/user/`	POST	UserList	`nick, password, name`

And the Web API client app will store that information in the `actions` element (see the code below). Note the `pattern` property information comes from another part of the TPS documentation (Table 4 (TK??)).

```
actions.user = {
  add:  {
    href:"/user/",
    prompt:"Add User",
    method:"POST",
    args:{
      nick: {
        value:"",
        prompt:"Nickname",
        required:true,
        pattern:"[a-zA-Z0-9]+"
      },
      password: {
        value:"",
        prompt:"Password",
        required:true,
        pattern:"[a-zA-Z0-9!@#$%^&*-]+"
      },
      name: {
        value:"",
        prompt:"Full Name",
        required:true
      }
    }
  }
}
```

And this kind of information needs to be handled for *all* the actions your client app needs to perform. In the TPS Web API, that is 17 actions. A non-trivial app will need to handle quite a few more.

www.it-ebooks.info

Quick Summary

So know we have a good sense of what a Web API client for a JSON RPC-CRUD style API will need to deal with. Along with the usual code for requests, parsing, and rendering responses, every JSON API client will also need to know how to handle:

- Key OBJECTS unique to each API
- ADDRESSES (URLs and URL templates) for all the actions of the API
- ACTION details including HTTP methods and arguments for all non-trivial actions including HTTP `POST`, `PUT`, and `DELETE` requests.

With this in mind, we can now dig into the actual code for the JSON Web API client.

The JSON SPA Client

Now we're ready to walk through the JSON Web API client app. We'll look at the HTML Container for the single-page app (SPA), the top-level request, parse, render loop, and check out how this client app handles the three things we reviewed above: Objects, Addresses, and Actions. Along the way we'll see the JSON Client in action and look ahead to see how it will deal with backend API changes.

The source code for the TPS JSON Web API client can be found in the associated github repo here: *https://github.com/RWCBook/json-client*. A running version of the app described in this chapter can be found here: *http://rwcbook03.herokuapp.com/files/json-client.html* (TK: check URLs)

Throughout the book, I'll be showing examples of Single-Page Apps or SPAs hosted within a browser. Also, I chose to build all the apps for this book without using one of the many javscript frameworks in order to make it easier for you to see the code that matters. So, the code here is not production ready because I wrote it for this book. But making it production-ready is just a matter of making it bullet-proof and you don't need any fancy frameworks for that.

The HTML Container

The SPA client created for the TPS Web API starts with a single HTML document. This document acts as the *container* for the entire API client application. Once the intial HTML is loaded, all other requests and responses will be handled by the runing javascript code parsing and rendering the JSON objects returned from the TPS Web API service.

The HTML container looks like this:

www.it-ebooks.info

```
<!DOCTYPE html>
<html>
  <head>
    <title>JSON</title>
    <link href="json-client.css" rel="stylesheet" /> ❷
  </head>
  <body>
    <h1 id="title"></h1> ❶
    <div id="toplinks"></div>
    <div id="content"></div>
    <div id="actions"></div>
    <div id="form"></div>
    <div id="items"></div>
    <div>
      <pre id="dump"></pre>
    </div>
  </body>
  <script src="dom-help.js">//na </script> ❸
  <script src="json-client.js">//na </script> ❹
  <script>
    window.onload = function() {
      var pg = json();
      pg.init("/home/", "TPS - Task Processing System"); ❺
    }
  </script>
</html>
```

As you can see from the HTML listing above, there is not much to talk about in this document. The part that all the *code* will be paying attention to starts at callout #1 — the seven `DIV` elements. Each of them has a unique identifier and purpose at runtime. You can figure most of that out just by reading the names. The last `DIV` actually encloses a `<pre>` tag that will hold the full 'dump' of the JSON responses at runtime. This is a handy kind of 'debug display' in case you need it.

Along with the HTML, there is a single CSS file (callout #2) and two Javascript references; a simple DOM library (callout #3) and the complete client-side code (callout #4). We'll inspect the `json-client.js` library throughout this section of the chapter.

Finally, once the page is loaded, a single function is executed (see callout #5). This initializes the client with a starting URL and (optionally) a title string). The URL shown here works fine when the client app is hosted in the same web domain as the TPS Web API. If you want to run the client app from a separate domain, all you need to do is update this intial URL and app will work just fine.

www.it-ebooks.info

The source code for the TPS JSON Web API client can be found in the associated github repo here: *https://github.com/RWCBook/json-client*. A running version of the app described in this chapter can be found here: *http://rwcbook03.herokuapp.com/files/json-client.html* (TK: check URLs)

So, let's look inside the `json-client.js` library and see how it works.

The Top-Level Parse Loop

The client app is designed to act in a simple, repeating loop that looks like this:

1. Execute an HTTP request
2. Store the JSON response in memory
3. Inspect the response for *context*
4. Walk through the response and render the context-related information on screen

We talked about *context* earlier in the chapter. This client is expecting multiple custom object models from the TPS Web API (`Task` and `User`) and uses the returned object model as *context* in deciding how to parse and render the response.

```
// init library and start ❶
function init(url, title) {
  if(!url || url==='') {
    alert('*** ERROR:\n\nMUST pass starting URL to the library');
  }
  else {
    g.title = title||"JSON Client";
    g.url = url;
    req(g.url,"get"); ❷
  }
}

// process loop ❸
function parseMsg() {
  setContext();
  toplinks();
  content();
  items();
  actions();
}
```

When the app first loads, the `init` function is called (see callout #1). That validates the initial URL, stores it, and eventually makes the first HTTP request to that URL (callout #2). Once the response returns (not shown here) the `parseMsg` function is called (clalout #3) and that starts the parse/render loop.

www.it-ebooks.info

The `parseMsg` function does a handful of things. First, it calls `setContext` to inspect the response and set the app's current *context* so that it knows how to interpret the the response. For our app, a global `context` variable is set to `"task"`, `"user"`, or `"home"`. Next, the page's top links are located and rendered (`toplinks`) and any HTML content is displayed (`content`). The `items` function finds all the objects in the response (`Task` or `User`) and renders them on the screen and the `actions` function constructs all the links and forms appropriate for the current *context*.

That's quite a bit in a single function and we'll get into some details of that in just a bit. But first, let's look at how the JSON client is keeping track of the TPS objects, addresses, and actions that were written up in the human documentation.

Objects, Addresses, and Actions

Since the TPS Web API is just returning custom JSON objects and arrays, our client app needs to know what those objects are, how to address them, and what actions are possible with them.

TPS Objects

The TPS objects (`Task` and `User`) are simple name-value pairs. So, all our client app needs to know are the properties of each object that need to be rendered on screen. For example, all TPS objects have `dateCreated` and `dateUpdated` properties, but our client doesn't need to deal with them.

This app uses a simple array to contain all the object properties it needs to know about:

```
g.fields.task = ["id","title","completeFlag","assignedUser"];
g.fields.user = ["nick","name","password"];
```

Now, whenever parsing incoming objects, the client app will compare the property on the object with its own list of properties and ignore any incoming property it doesn't already 'know' about.

An example of how this works can be seen in the code that handles on-screen rendering of objects in the response.

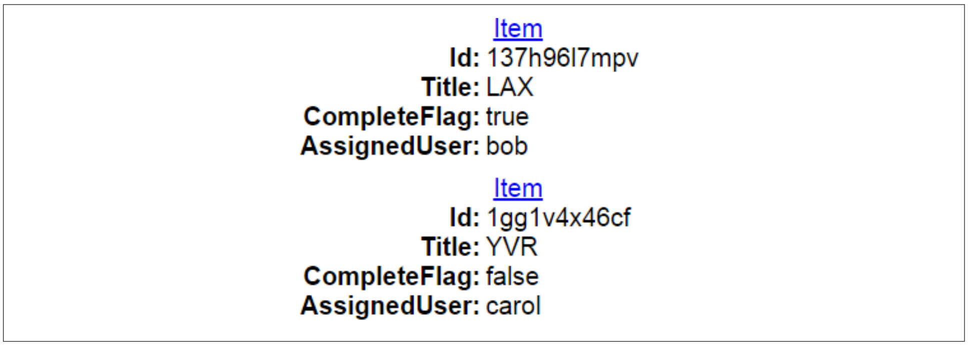
```
// handle item collection
function items() {
  var rsp, flds, elm, coll;
  var ul, li, dl, dt, dd, p;

  rsp = g.rsp[g.context]; ❶
  flds = g.fields[g.context];

  elm = d.find("items");
  d.clear(elm);
  ul = d.node("ul");
```

www.it-ebooks.info

```
      coll = rsp;
      for(var item of coll) {
        li = d.node("li");
        dl = d.node("dl");
        dt = d.node("dt");

        // emit the data elements
        dd = d.node("dd");
        for(var f of flds) { ❷
          p = d.data({text:f, value:item[f]});
          d.push(p,dd);
        }

        d.push(dt,dl);
        d.push(dd,dl);
        d.push(dl,li);
        d.push(li,ul);
      }

      d.push(ul,elm);

    }
```

The code examples for this book use the ES6 `for..of` iterator. When this book first went to press in early 2016 `for..of` was supported in *some* browsers, but not all. I used the Chrome browser (both the Google release and the Chromium open source release) while writing the examples and they all ran fine. Be sure to check your browser's support for the `for..of` iterator.

Note (in callout #1) the first step in the routine is to use the shared context value to locate the data in the response (`rsp`) and select the internal properties to use when inspecting the data (`flds`). This information is used (in callout #2) to make sure to only render the fields the client decides is appropriate.


Item
Id: 137h96l7mpv
Title: LAX
CompleteFlag: true
AssignedUser: bob

Item
Id: 1gg1v4x46cf
Title: YVR
CompleteFlag: false
AssignedUser: carol

Figure 2-4. Rendering Task Items in the JSON Client

www.it-ebooks.info

Addresses and Actions

This client app stores both the addresses (URLs) and actions (HTTP method and parameter information) in a single internal collection called `actions`. There is additional metadata about each action that indicates *when* it should be rendered (based on context information) and *how* it should be executed (e.g. as a simple link, via, a form, or a direct HTTP method call).

The list of actions for the TPS Web API is rather long (17 separate actions) but the code snippet below gives you a good idea of how they are stored in the client app.

```
// task context actions
g.actions.task = {
  tasks:   {target:"app", func:httpGet, href:"/task/", prompt:"Tasks"}, ❶
  active:  {target:"list", func:httpGet, href:"/task/?completeFlag=false",
             prompt:"Active Tasks"
           },
  byTitle: {target:"list", func:jsonForm, href:"/task", ❷
             prompt:"By Title", method:"GET",
             args:{
               title: {value:"", prompt:"Title", required:true}
             }
           },
  add:     {target:"list", func:jsonForm, href:"/task/", ❸
             prompt:"Add Task", method:"POST",
             args:{
               title: {value:"", prompt:"Title", required:true},
               completeFlag: {value:"", prompt:"completeFlag"}
             }
           },
  item:    {target:"item", func:httpGet, href:"/task/{id}", prompt:"Item"},
  edit:    {target:"single", func:jsonForm, href:"/task/{id}", ❹
             prompt:"Edit", method:"PUT",
             args:{
               id: {value:"{id}", prompt:"Id", readOnly:true},
               title: {value:"{title}", prompt:"Title", required:true},
               completeFlag: {value:"{completeFlag}", prompt:"completeFlag"}
             }
           },
  del:     {target:"single", func:httpDelete, href:"/task/{id}", ❺
             prompt:"Delete", method:"DELETE", args:{}
           },
};
```

In the code snippet above, you can see a simple safe, read-only action (callout #1) as well as a safe action that required user input (callout #2). There are also the classic CRUD actions (#3, #4, and #5) with the expected HTTP method names, prompts, and (where appropriate) argument lists. These action definitions are selected based on runtim context information. For example the `target` property indicates which

www.it-ebooks.info

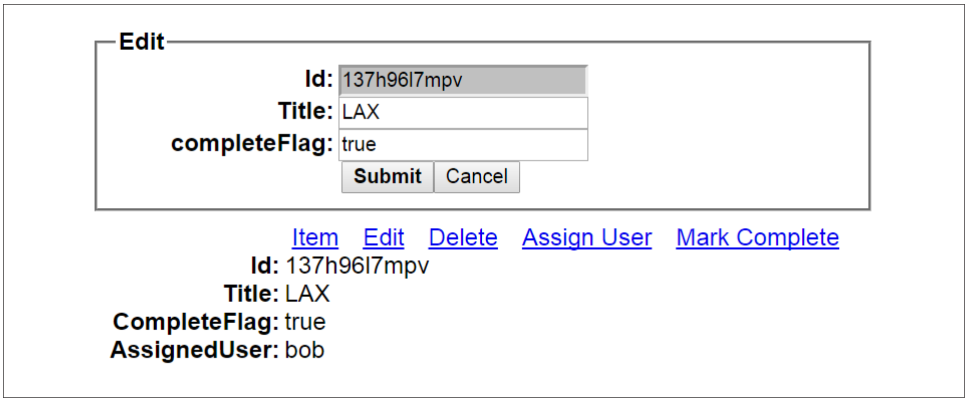
actions are appropriate for app-level, list-level, item-level, and even single-item level context.

Here's an example of the code that uses context information and scans the list of actions to render 'list-level' links.

```
// handle list-level actions
function actions() {
  var actions;
  var elm, coll;
  var ul, li, a;

  elm = d.find("actions");
  d.clear(elm);
  ul = d.node("ul");

  actions = g.actions[g.context]; ❶
  for(var act in actions) {
    link = actions[act];
    if(link.target==="list") { ❷
      li = d.node("li");
      a = d.anchor({
        href:link.href,
        rel:"collection",
        className:"action",
        text:link.prompt
      });
      a.onclick = link.func;
      a.setAttribute("method",(link.method||"GET"));
      a.setAttribute("args",(link.args?JSON.stringify(link.args):"{}"));
      d.push(a,li);
      d.push(li, ul);
    }
  }
  d.push(ul, elm);
}
```

You can see (in the code above) that both the object context (callout #1) and the internal render context (#2) are used to select only the links appropriate for display at the moment.

www.it-ebooks.info



Id: 137h96l7mpv
Title: LAX
completeFlag: true

Id: 137h96l7mpv
Title: LAX
CompleteFlag: true
AssignedUser: bob

Figure 2-5. JSON Client Rendering a Form

The actions that contain argument details will be rendered at runtime using HTML `<form>` elements. The code that handles this is listed below:

```
// render inputs
coll = JSON.parse(link.getAttribute("args")); ❶
for(var prop in coll) {
  val = coll[prop].value;
  if(rsp[0][prop]) {
    val = val.replace("{"+prop+"}",rsp[0][prop]); ❷
  }
  p = d.input({ ❸
    prompt:coll[prop].prompt,
    name:prop,
    value:val,
    required:coll[prop].required,
    readOnly:coll[prop].readOnly,
    pattern:coll[prop].pattern
  });
  d.push(p,fs);
}
```

In the small snippet above, you can see the collection of `args` (callout #1) from the `action` definition is used to create HTML `form` inputs. At callout #2 you can see that any current object is used to populate the value of the inputs before the actual HTML `input` element is created (callout #3). Note the inclusion of the HTML5 properties `required`, `readonly`, and `pattern` in order to improve the client-side user experience, too.

Quick Summary

There is more to the `json-client.js` library that we won't cover here including all the Ajax-related code to handle HTTP requests and responses. Even with all the low-level HTTP code, the total size of the library is around 500 lines of Javascript — and

www.it-ebooks.info

that includes extensive comments. In fact, the breakdown of the various parts of the client are worth noting.

HTTP-level Handlers
: For example, the Ajax-related low-level HTTP code takes up about 100 lines. This code would typically be handled by jQuery or other framework libraries. Also, this code will not grow or shrink as the number of unique objects, addresses, and actions changes within the Web API. The HTTP-handling code is a fixed size.

Parsing and Rendering
: The heart of the JSON Client code is about 300 lines that handle the parsing and rendering of the user interface. That content is also unlikely to change as the Web API objects and functionality changes. However, this part of the codebase *could* grow as more UX features are added to the library.

Web API Objects, and Actions
: Finally, the other large portions of the code library is the 150 or so lines of Javascript used to declare the TPS web API objects, address, and action configuration. This is the part of the library that will be directly affected by the Web API objects and actions. As the API functionality grows, this code MUST grow, too.

This last item points to an important aspect of JSON API clients: changing the backend API will *force* the client frontend to change, too. I've done my best to isolate those changes to a single section of the library, but I can't eliminate the need for these changes to occur since the client code has all the server objects, addresses, and actions 'baked' directly into the code.

So, let's make some changes to the backend Web API and see how this JSON Client handles it.

Dealing with Change

Throughout the book, I'll be introducing backend changes to service APIs *after* the client application has been completed (and presumably released into production). I'm doing this to explore how various API client implementations deal with change and to see what it takes to create API clients that can adapt to selected changes in the API at runtime.

Change is a fundamental part of the Web. Several key design aspects of the HTTP protocol and HTML make change not only easy to do but also easy to support without breaking existing HTTP clients. The content of HTML pages can change without the need to re-code and release new HTML browsers. The HTTP protocol has undergone a handful of updates over the last 25 years without crashing existing

www.it-ebooks.info

Web servers or requiring them to all be updated at the sime time. Change is essentially 'designed-in' to the WWW.

Unfortunately, most Web APIs today do not share this fundamental ability to support change over time. Changes on the server-side of a Web API usually requires changes on the client-side. Sometimes existing production API clients will even crash or operate improperly once the server-side code has been changed. The problem has become so common that most Web developers resort to using explicit version numbers on Web APIs in order to make it easy for developers to know when something has changed — usually so developers and turn around and re-code and deploy their production client apps in order to maintain compatibility with the service.

So, let's make some changes to the TPS Web API and see how our JSON Client reacts.

The source code for the TPS JSON Web API service can be found in the associated github repo here: *https://github.com/RWCBook/json-crud-v2*. A running version of the service can be found here: *http://rwcbook04.herokuapp.com/task/* (TK: check URLs)

Adding a Field and Filter

A common change that can occur in a production API is adding a new field to the data storage. For example, the team at BigCo working on the Web API might decide to add a `tag` field to the `Task` storage object. This will allow users to 'tag' tasks with common keywords and then recall all the tasks with the same keyword.

Adding the `tag` field means we'll probably need a new search option, too: `TaskFilterByTag`. It would take a single parameter (a string) and use that to search all the task record's `tag` fields, returning all `Task` objects where the search value is contained in the `tag` field.

Changing the TPS Web API

The process of changing the TPS Web API to support the new `tag` functionality is not too complex. We'll need to:

- Add the `tag` property to the `Task` storage definition
- Introduce a new query for `Task` storage and expost that via HTTP

To add the new `tag` field to the server's storage support, we first need to update line of code that defines the valid fields for the `task` object (see callout #1).

```
// task-component.js
// valid fields for this record
props = [
```

www.it-ebooks.info

```
    "id",
    "title",
    "tag", ❶
    "completeFlag",
    "assignedUser",
    "dateCreated",
    "dateUpdated"
  ];
```

Next, we need to modify te validation rules for adding and updating `task` objects on the server. The code snippet below shows the `addTask` routine with the new `tag` field (see callout #1 below). A similar change was made to the `updateTask` routine, too.

```
function addTask(elm, task, props) {
  var rtn, item;

  item = {}
  item.tags = (task.tags||""); ❶
  item.title = (task.title||"");
  item.assignedUser = (task.assignedUser||"");
  item.completeFlag = (task.completeFlag||"false");
  if(item.completeFlag!=="false" && item.completeFlag!=="true") {
    item.completeFlag="false";
  }
  if(item.title === "") {
    rtn = utils.exception("Missing Title");
  }
  else {
    storage(elm, 'add', utils.setProps(item, props));
  }

  return rtn;
}
```

Testing the Updated TPS Web API

With these changes in place, we can use the `curl` command-line app to validate our changes. A command to create a record with the `tag` value set to `"test"` looks like this:

```
curl -X POST -H "content-type:application/json" -d
  '{"title":"Run remote client tests","tags":"test"}'
  http://localhost:8181/task/
```

and creates a new `task` record that looks like this:

```
{
  "id": "1sog9t9g1ob",
  "title": "Run server-side tests",
  "tags": "test",
  "completeFlag": "false",
  "assignedUser": "",
```

www.it-ebooks.info

```
      "dateCreated": "2016-01-28T07:16:53.044Z",
      "dateUpdated": "2016-01-28T07:16:53.044Z"
    }
```

Assuming several new records were created, executing the filter query would look like this:

```
curl http://localhost:8181/task/?tags=test
```

would returns one or more `task` records with the `tag` value that contains `"test"`

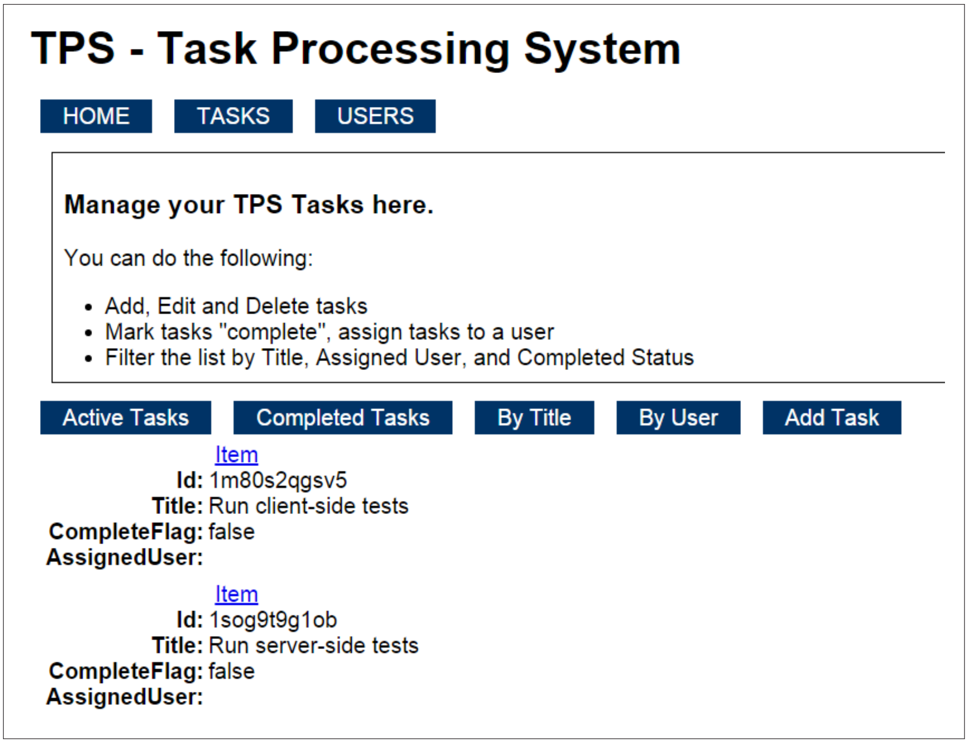
With the TPS Web API updated and validated, we next need to see how the JSON API Client handles the change in production.

For completeness, we should also update the TPS Web API documentation. We'll skip that step right now, though.

Testing the JSON API Client

The easiest way to test the JSON API Client's support for the new `tag` field and filter option is to simply run the client and check the results. Below is a screenshot from the JSON API Client making a request to the new TPS Web API server.

www.it-ebooks.info


TPS - Task Processing System

Manage your TPS Tasks here.

You can do the following:

- Add, Edit and Delete tasks
- Mark tasks "complete", assign tasks to a user
- Filter the list by Title, Assigned User, and Completed Status

Item
Id: 1m80s2qgsv5
Title: Run client-side tests
CompleteFlag: false
AssignedUser:

Item
Id: 1sog9t9g1ob
Title: Run server-side tests
CompleteFlag: false
AssignedUser:

Figure 2-6. JSON API Client w/o Tag Support

As you can see from the screenshot, even though the new `task` records appear in the client, the `tag` field is missing from the display as well as the new filter option. The good news is our JSON client didn't *crash* when the new feature was added. The bad news is our client simply ignored the new functionality.

The only way our JSON client will be able to take advantage of this new option is to re-code and redeploy a new version of the app into production.

Coding a New Client

To get the JSON API Client to reflect the new `tag` support, we need to update the client's object and action data. The client needs to know about the tag feature before it can use it. Because our JSON Client was designed to keep the object and action data separate from the rest of the library, adding the new feature is relatively easy.

First, we need to update the client's object properties (callout #1):

```
// task fields
g.fields.task = [
  "id",
```

www.it-ebooks.info

```
    "title",
    "tags", ❶
    "completeFlag",
    "assignedUser"
  ];
```

Next, we need to add the new filter option to the list of the client's `task.actions`:

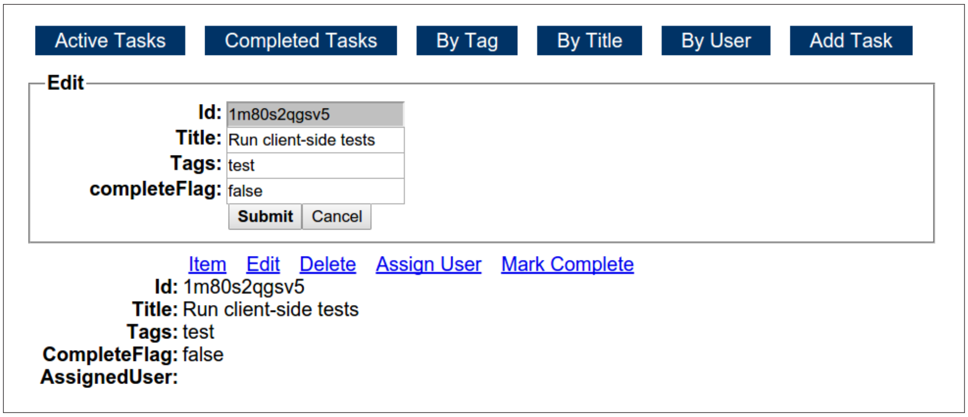
```
byTags: {
          target:"list",
          func:jsonForm,
          href:"/task",
          prompt:"By Tag",
          method:"GET",
          args:{
            tags: {value:"", prompt:"Tags", required:true}
          }
        }
```

and update the `addTask` and `updateTask` action definitions (callout #1):

```
add:  {
        target:"list",
        func:jsonForm,
        href:"/task/",
        prompt:"Add Task",
        method:"POST",
        args:{
          title: {value:"", prompt:"Title", required:true},
          tags: {value:"", prompt:"Tags"}, ❶
          completeFlag: {value:"", prompt:"completeFlag"}
        }
      },
edit: {
        target:"single",
        func:jsonForm,
        href:"/task/{id}",
        prompt:"Edit",
        method:"PUT",
        args:{
          id: {value:"{id}", prompt:"Id", readOnly:true},
          title: {value:"{title}", prompt:"Title", required:true},
          tags: {value:"{tags}", prompt:"Tags"}, ❶
          completeFlag: {value:"{completeFlag}", prompt:"completeFlag"}
        }
      }
```

With these changes in place, we can now see the JSON Client supports the new `tag` features of the TPS Web API.

www.it-ebooks.info



Edit

Id:	1m80s2qgsv5
Title:	Run client-side tests
Tags:	test
completeFlag:	false

Id: 1m80s2qgsv5
Title: Run client-side tests
Tags: test
CompleteFlag: false
AssignedUser:

Figure 2-7. JSON Client V2 Supports Tagging

You may have noticed that the changes we made to the client app look similar to the changes we made in the server API. That's not by accident. Typical JSON APIs require the client app and the server-side code share the same object/address/action profiles in order to keep in step with each other. That means every new feature on the service requires a new release for the client.

The original JSON Client app we created was able to keep offering the same functionality without an update because the TPS Web API didn't introduce breaking changes. If, for example, the service had changed the `addTask` and `updateTask` operations to make the `tag` field *required* when adding/editing `Task` objects, the original client app would no longer be able to save `Tasks`. Since the service made the `tag` field an optional input, the initial client app was still able to function; it just couldn't take advantage of the new feature. Essentially, when API services change, the best client apps can hope for is that services will not make *breaking* changes to the API (e.g. removing fields/functions, changing existing functions, or adding new required fields, etc.).

However, there are ways to design service APIs that allow clients to adapt to changing responses — even have the ability to expose new features (like the `tag` field and filter) w/o the need for re-coding the client. We'll explore that in future chapters.

Summary

In this chapter we reviewed the key aspects of JSON API Clients including the need for them to deal with:

- Handle key API OBJECTS in the service model
- Construct and manage service ADDRESSES or URLs

www.it-ebooks.info

- Know all the ACTIONS metadata such as parameters, HTTP methods, and input rules

We then did a walk-through of our sample JSON Client and saw how it handled the objects/addresses/actions challenge. We also noted that the client app was written using a simple loop pattern that made a request, parsed the response, and (based on context information) rendered information to the screen for a human to deal with. The fully-functional SPA client requires just over 500 lines of Javascript — even with all the low-level HTTP routines.

JSON Clients and the OAA Challenge

It turns out clients that receive only plain JSON responses, don't do well on our OAA Challenge. These clients either break or ignore changes to any of the three elements. They need to be re-coded and redeployed any time an OBJECT, ADDRESS, or ACTION is added, removed, or changed.

Finally, we introduced a new feature on the TPS Web API service and saw that our JSON client ignored the new feature. Luckly, the service was updated in a backward-compatible way and our client didn't crash or *lose* functionality. But we had to re-code and redeploy the client in order to take advantage of the new `tag` feature.

Bob and Carol

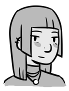
"Hey, Bob. Just stopping by to review our experiences with the TPS Web API Project."

"Hi, Carol. Good to see you. Let's get started."

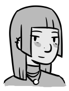
"First, we certainly learned a lot in the last week. It took some doing, but even with handling close to 20 operations in this API, we were able to get our JSON client SPA implementation down to around 500 lines of Javascript."

"That's pretty impressive. You were defintiely able to handle all the documented features details, too. Testing the functionality of the initial client went really well."

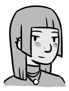
www.it-ebooks.info


"Right, but that brings up our major disappointment for this project, too. Changes to the backend made our client app obsolete."

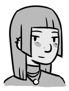
"Yeah, sorry about that, Carol. After we released the API into production some of the stakeholders came back to us and asked for that tagging feature. We had to add that later."


"Well, at least you made sure to add the new feature in a way that didn't *break* our JSON client. But, as you know, none of the customers could see the new tag field or filter until we re-coded and redeployed our JSON client."

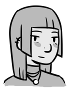
"Yep. Hopefully, that won't happen too often."


"Well, Bob, I'm a bit concerned that it *will* keep happening. I mean, change is inevitable, right?"

"I guess that's right. But we can't stop people asking for new features. I guess this is just they way the Web works. We keep updating and re-releasing from now on."


"I'm not so sure about that, Bob. I think we need to look into another way to build Web APIs and client apps. One that does a better job of supporting changes over time."

"Well, Carol, I'd like to hear more about that. But right now, I need to meet w/ my server-side team to review some more changes we need to make to the API."


"More changes? OK, Bob. Talk to you later."

www.it-ebooks.info

References

1. Twitter has poublished blog posts on both their initial SPA release and their subsequent redesign to improve then performance and reliability their web-based client app.
2. You can learn more about Twitter's API by reviewing the online Documentation
3. There is a nice ECMAScript compatibility chart hosted at *https://kangax.github.io/compat-table/es6/*

www.it-ebooks.info

CHAPTER 3

The Representor Pattern

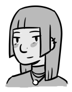
"But it is a pipe."

"No, it's not," I said. "It's a drawing of a pipe. Get it? All representations of a thing are inherently abstract. It's very clever."

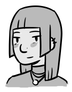
—John Green, *The Fault in Our Stars*

Bob and Carol

"Hi, Carol. Wanted to drop by your office here and see how things are going here in your new role."


"Oh, hello Bob. I'm settling into your old office rather nicely, I think. And the client-side work is going well, too."

"Great. I actually wanted to talk to you about some server-side items that you and your team had been working on before we switched roles. Specifically, about implementing support for all these output formats. See, while you had been doing a great HTML rendering, we now need to start adding support for JSON."


"Right, and some of the other teams were starting to ask about support for additional registered formats like HAL, Collection+JSON, Siren, and so forth. We were about to dig into that when the re-org happened."

www.it-ebooks.info

"Yep, your old colleagues have been bringing me up to speed on this and, before we got too deep into it, I wanted to check back with you and go over your initial notes."

"Well, I don't know how much I can offer since we had just begun our discovery. But you probably have that material already, right?"

"Yes, I have some solid info on the available formats, and lots of feedback on which one is considered `best', etc. But that's not really important right now. What I really wanted to review with you is some implementation ideas -- how we're actually going to *do* this work."

"Ok, Bob. Makes sense. What do you have so far?"

"Well, I was in this meeting yesterday where people were getting into heated debates about which of these formats to support. There seems to be a wide concensus on adding support for plain JSON objects and there are also some people very adamant about supporting Collection+JSON or HAL or Siren -- some of the so-called hypermedia formats. It got me to thinking that this could be a real mess."

"Well, that's possible but, as I see it, you don't really need to decide *which* format to support. In fact, I think that's a losing strategy."

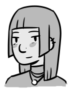
"Wait, Carol, you lost me there. I don't need to decide on a format? That doesn't sound right. I mean, I need to implement an API format, right?"

"Well, you certainly have to implement a format, but I don't think you need to implement _just one_ format. That's the point I am making here. Instead of trying to get everyone to agree on a single format, the

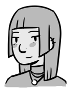
www.it-ebooks.info

better approach is to implement the service in a way that makes supporting one or more formats *trivial*."

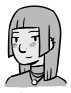
"Hmmm. You mean diffuse the argument by saying `yes' to everyone? That sounds like making no decision at all. That sounds even worse to me than picking one."


"No, not at all. One of the things I was exploring before the re-org was the notion of separating the format from the internals of the API -- a kind of loose-coupling approach to the output formats."

"Oh, I get it. You were thinking of isolating the format details from the rest of the implementation. That way, the actual resolution of the format question doesn't adversely affect the rest of the system."


"Right, Bob. I got the idea from an old paper on software modularity. I can't recall which one, but I think that's in the notes you already have."

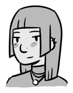
"Ah, good. I'll dig into those again. So, decoupling the output format from the internal model of the API itself means we can forge ahead with the internal model even if we don't yet have a single format selected."


"Yep, and -- if you do it right -- I bet you can come up with an implementation pattern that makes is relatively easy to support more than one output format for the same API. Now that would be something!"

"Huh, support multiple output formats cheaply and easily? Yes, Carol, that would be something. I'll need to think about that, too. Ok, I need to get back to the team and see what they think about all this. We'll let you know what we find."

www.it-ebooks.info

"Sounds good, Bob. Talk to you soon."



Almost every team starts out on the journey of implement APIs for the Web runs up against the decision of which output format(s) to support. Most often, this decision is a matter of accepting the current norms rather than any elaborate set of experiments and research efforts. And that's fine. Usually teams don't have the time or energy to go wandering through decades of material on software implementation and systems engineering in order to decide which output format the API will use. Instead the current custom or fad is the one that wins the day.

And selecting a format is only part of the challenge. A more important consider is just *how* the write services that implement output format support. Sometimes services are implement in a way that tightly binds the internal object model to the external output format. That means changes in the interal model leak out into the output format and are likely to break client applications consuming the service.

That leads to another important challenge to face when dealing with messages passed between API client/consumer and API service/provider: protecting against breakage. Long ago, writers on softrware modularity offered clear advice on how to isolate parts of a system that are likely to change often and impelment them in ways that made changes to that aspect of the system relatively cheap, safe, and easy. keeping this in mind is essential for building healthy and robust API programs.

So, there are a number of things to cover here and the first challenge to face is the constant question 'Which output format should we use for our API?'

XML or JSON: Pick a Side!

So you want to impelment an API, eh? Well, one of the first decisions you will face is which output format to use. Today, almost everyone decides on JSON. Often with little to no discussion. That's the power of current popularity — that the decision is made without much contemplation. And it turns out selecting JSON may not be the best choice or the *only* choice when it comes to your API output format.

Not surprisingly, the *de facto* choice for API output was not always JSON. All through the late ninties and early 2000s, the common custom was to rely on output formats based on the XML standard. At that time, XML had a strong history — HTML and XML both had the same progenitor in SGML(ISO 8879:1986) — and there were lots of tools and libraries geared to parsing and maniuplating XML documents. Both the Simple Object Access Protocol (SOAP) specification and much of what would later be known as the SOA (Service-Oriented Architecture) style started as XML-based efforts for business computing.

www.it-ebooks.info

XMLHttpRequest

One of the most important API-centric additions to the common Web browser — the ability to make direct calls to services *within* a single Web page — was called the `XMLHttpRequest` object *because* it was assumed that these browser-initiated inline requests would be returning XML documents. And they did in the beginning. But by the mid 2000s the Javascript Object Notation (JSON) format would overtake XML as the common way to transfer data between services and Web browsers. The format has changed, but the object name never has.

But, as we all know, selecting XML for passing data between services and clients did not *end* the format debate at all. Even while the XML-based Simple Object Access Protocol (SOAP) document was being published as a W3C Note in May 2000, there was another effort underway to standardize data-passing documents - the Javascript Object Notation or JSON format.

Douglas Crockford is credited with specificying JSON in early 2001. Even though the JSON RFC document (RFC627) was not published until 2006, the format had experienced wide use by that time and was gaining in popularity. As of this writing, JSON is considered the default format for any new API. Recent informal polls and surveys indicate few APIs today are being published using the XML output format and — at least for now — there is no new format likely to undermine the current JSON popularity.

"I did not invent JSON"

In 20011, Douglas Crockford gave a talk he dubbed "The True Story of JSON" and, in it, he said "I do not claim to have invented JSON. … What I did was I found it, I named it, I described how it was useful. … So, the idea's been around there for a while. What I did was I gave it a specification, and a little Web site." He even states that he saw an early example of JSON-like data-passing as early as 1996 from the team that was working on the Netscape Web browser.

Of course, XML and JSON are not the only formats to consider. For example, another valuable format for passing data between parties is the Comma-Separated Value (CSV) format. It was first standardized by the IETF in 2005 (RFC4180) but dates back to the late 1960s as a common interchange format for computers. There are likely going to be cases where an API will need to output CSV, too. For example, almost all spreadsheet software can easily consume CSV documents and place them in columns and rows with a high degreee of fidelity.

www.it-ebooks.info

Clearly the problem is not just deciding between XML and JSON.

The New Crop of Hypermedia Formats

Starting in the early 2010s, a new crop of formats emerged that offered more than just structure for data, they included instructions on how to manipulate the data as well. These are a set of formats I refer to as 'Hypermedia Formats'. These formats represent another trend in APIs and, as we will see later in the book, an valuable tool in creating API-based services that can support a wide range of changes without breaking existing clients. In some cases, they even allow client applications to 'auto-magically' acquire new features and behaviors without the need for re-writing and redeploying client-side code.

But that's getting ahead of the story a bit.

Atom Syndication and Publishing

Although most of the new hypermedia formats appeared on the sceen around 2010 and later, one format (the Atom Syndication Format) was standardized in 2005 as RFC4287, it has similar roots as the SOAP initiative and is an entirely XML-based specification. The Atom Format, along with the Atom Publishing Protocol (RFC5023) in 2007, outline a system of publishing and editing Web resources that is based on the common Create-Read-Update-Delete (CRUD) model of simple object manipulation.

Atom documents were mostly used in read-only mode for news feeds and other simple record-style output. However, several blog engines supported editing and publishing entries using Atom documents. There are also a number of registered format extensions to handle things like paging and archiving (RFC5005), threads (RFC4685), and licensing content (RFC4946). I don't often see Atom used to support read/write APIs on the WWW but still see it used in enterprise cases for handling outputs from queues and other transaction-style APIs.

Atom is interesting because it is an XML-based format that was designed specifically to add read/write semantics to the format. In other words, like HTML, it describes rules for adding, editing, and deleting server data.

And, since the release of the Atom spcifications, a handful of other formats have been published.

Other Hypermedia Formats

Starting in 2011, there was a rush of hypermedia-style formats published and registered with the IANA. They all share a similar basic set of assumptions even though each has unique strengths and focus on different challenges for API formats. I'll cover some of these at length later in the book but wanted to mention them here to provide

www.it-ebooks.info

a solid background for dealing with the challenge of selecting and supporting formats for APIs.

Hypermedia Application Lanugage (HAL)
: The HAL format was registered with the Internet Authority for Names and Addresses (IANA) in 2011 by Mike Kelly. Descrbied as "…a simple format that gives a consistent and easy way to hyperlink between resources…", HAL's design focus is on standardizing the way links are described and shared within messages. HAL does not describe write semantics but does leverage the URI Templates specification (RFC6570) to describe query details inline. We'll spend an entire chapter exploring (and using) this very popular hypermedia type.

Collection+JSON Format (Cj)
: I published the Collection+JSON hypermedia format the same year as Mike Kelly's HAL (We had been sharing ideas back and forth for quite a while before that). Unlike HAL, Cj supports detailed descriptions of the common Cread-Read-Update-Delete (CRUD) semantics inline along with a way to describe input metadata and errors. It is essentially a JSON-formatted fork of the Atom Publishing Protocol taht is focused on common list-management use cases. We'll spend time coding for Cj formats later in the book.

The Siren Format
: The Siren format was created by Kevin Swiber and registered at the IANA in 2012. Siren"is a hypermedia format for representing entities with their associated properties, children, and actions." It has a very rich semantic model that supports a wide range of HTTP verbs and it is currently used as the default format for the Zetta Internet of Things platform. We'll get a chance to dig into Siren later in the book.

The Universal Basis for Exchanging Representations (UBER)
: I released a working draft of the UBER format in 2014. Unlike the other hypermedia formats listed here, UBER does not have a strong message structure but, instead, has just one element (called "data") used for representing all types of content in a document. It also has both a JSON and XML variant. UBER has not yet been registered with the IANA and will not be covered in this book.

Other Formats
: There are a number of other interesting hypermedia-style formats that have recently appear that won't be covered in this book. They include Jorn Wildt's Mason, the JSON API spec from Yehuda Katz, Cross-Platform Hypertext Language by Mike Stowe, and several others. I suspect that by the time you are reading this book there are new formats available, the status of some of the ones I list here has changed, and possibly some of them listed here have disappeared.

www.it-ebooks.info

Currently none of these new formats are a clear leader in the market and that, I think, is a good thing. In my experience it is not common that an important universally valuable message format appears "out of the blue" from a single author. It is more likely that many formats from several design teams will be created, published, and tested in real-world scenarios before any possible "winner" will emerge. And the eventual solution will likely take several years to evolve and take several twists and turns along the way.

So, even though many people have said to me "I just wish someone would pick *just one* format so I would know what to use", I don't think that will happen any time soon. It may seem like a good thing that you don't have a choice to make, but that's rarely true in the long run.

We need to get used to the idea that there is not a "one API format to rule them all."

The Fallacy of *The Right One*

So, despite all the new hypermedia formats out there and the continued use of XML in enterprises — with the current trend pointing toward JSON as the common output format — it would seem an easy decision, right? Any time you start implementing an API, just use JSON and you're done. Unfortunately, that's almost never the way it goes.

First, some industry verticals still rely only on XML and SOAP-based formats. If you want to interact with them, you'll need to drop your "JSON-only" approach and support SOAP or some other custom XML-based formats. Examples might be partner APIs that you work with on a regular basis, government or other standards-led efforts that continue to focus on XML as their preferred format, and even third-party APIs that you use to solve important business goals.

Second, many companies invested heavily in XML-based APIs over the last decade and are often unwilling to rewrite these APIs *just* to change the output format from then then-popular XML to the now-popular JSON format. Unless there is a clear advantage to changing the output format (e.g. increased speed, new functionality, or some other business metric), these XML-based services are not likely to change any time soon.

Finally, some data storage systems are 'XML-Native' or default to outputing data as XML documents (e.g. dbXML, MarkLogic, etc). While some of these services may offer an option to output the data as JSON, many continue to focus on XML and the only clear way of converting this data to JSON is to move it to other 'JSON-Native' data storage systems like MongoDB, CouchDB and others.

So, deciding on a single format for your team's service may not be feasible. And, as your point of view widens from a single team to multiple teams within your company, multiple products within an enterprise, on up to the entire WWW itself, getting

www.it-ebooks.info

everyone to agree to both produce and consume a single output format is not a reasonable goal.

As frustrating that this may be for team leaders and enterprise-level software architects, there is no single "Right One" when it comes to output formats. While it may be possible to control a single team's decision (either through concensus or fiat), one's ability to exert this control wanes as the scope of the community grows.

And that means the way forward is to re-think the problem, not work harder at implementing the same solution.

Re-Framing the Problem

One way to face a challenge that seems insurmountable is to apply the technique of *re-framing* — to put the problem in a different light or from a new point of view. Instead of working harder to come up with a solution to the percieved problem, reframing encouages us to step 'outside the frame' and try change our perspectrive. Sometimes this allows us to recognize the scenario as a completely different problem — one that may have an easier or simpler solution.

Cognitive Reframing

The current use of the term *reframing* came from the cognitive therapy work of Aaron T. Beck in the 1960s. As he was counseling patients experiencing depression he hit upon the idea that patients could be taught to become aware of negative thoughts as they arose and to "examine and evaluate them", even turn them into positive thoughts. Intially called cognitive reframing, now the term is used to describe any technique that helps us reflect on our thoughts and situation and take a new perspective.

In our case (the challenge of selecting a single format for your APIs), it can help to ask "Why do we need to decide on a single format?" or, to put it another way "Why not support many formats for a single API?" Asking these questions gives us a chance to lay out some of the reasons for and against supporting multiple formats. In this way, we've side-stepped the challenge of picking *one* format. Now we're focused on a new aspect of the same problem. Why not support *multiple* formats?

Why is Supporting One Format 'Better'?

The common pattern is to *assume* that selecting a single format is the preferred solution. To that end, there are some typical justifications for this point of view:

One Format is Easier
: Usually people make the case for supporting a single format for API output because it is thought to be easier than supporting multiple formats. It may not be

www.it-ebooks.info

ideal to select just one format, but it is preferable to the cost of supporting more than one. And this is a valid consideration. Often we work with programming tools and libraries that make supporting multiple output formats costly in some way (additional programming time, testing difficulty, runtime support issues, etc).

Multiple Formats is Anarchy
There are other times when making the case for supporting more than one format is precieved as making the case for supporting *any* format. In other words, once you open the door for one additional format, you MUST support any format that might be thought of at some point in the future.

The Format You Prefer is 'Bad'
Sometimes, even in cases where multiple formats might be possible, some start to offer value judgements for one or more of the suggested formats saying they are (for any number of reasons) 'bad' or in some other way insufficient and should not be included. This can turn into a 'war of attrition' that can prompt leadership to just pick one and be done with the squabbling.

We Can't Know What People Will Like in the Future Anyway
Another reason to argue for just one format is that selecting any group of formats is bound to result in *not* select one or more formats that, at some future point, will become very popular. If you can't accurately predict which ones to pick for the future, it's a waste of time to pick any of them.

The list goes on with any number of variations. But the theme is usually the same. You won't be sure to pick the right formats, so just pick a single one and avoid the costly mistakes of adding other formats no one will like or use in the future. The underlying assumptions in these arguments is also generally the same. They look something like this:

1. Supporting multiple formats is hard
2. There is no way to safely add formats over time without disrupting production code
3. Selecting the right formats *today* is required (and is impossible)
4. Suppoting multiple formats to too costly when you don't have guaranteed uptake.

And it turns out, these assumptions are not always true.

What Would it Take to Support Multiple Formats?

When reframing the problem from a new perspective, you get a chance to ask different questions and try out different approaches for a solution. One way of helping a team reframe any challenge is to cast it as a "What would it take…" question. Essen-

www.it-ebooks.info

tially you ask the team to describe a scenario under which the suggested alternative (in our case, supporting multiple output formats) would be a reasonable idea. For example "What has to exists (or be made possible) in order to support multiple output formats for the same API?"

And it turns out that the assumptions listed above are a great starting point for setting out a scenario under which supporing multiple formats for your API is reasonable.

For example, you might make the following statements:

1. Supporting multiple formats for the same API needs to be relatively easy to do.
2. We need to be able to safely add new format support over time without disrupting production code (or existing clients).
3. We need some kind of consistent criteria for selecting which formats to add both now and in the future.
4. Adding a format needs to be 'cheap enough' that, even it if turns out to be little-used it is not a big deal.

Even though most of the statements here are qualitative criteria ("relatively easy", "cheap enough", etc.) we can use the same patterns of judgement and evaluation on the format challenge that we do when resolving other implementation-related challenges such as *What is an API resource?*, *Which HTTP method should we use?* and others we face every day.

Luckily, there is a set of well-tested and documented programming patterns that we can use as a test-case for implementing multiple format support for out APIs. And they date back to some of the earliest work on software patterns in the 1980s.

The Representor Pattern

To explore what it would take to make supporting multiple output formats for APIs we need to work on a couple things. For a start, we should try to make it 1) Relatively easy initially and 2) Safe to add new formats after production release. I've found the first task is to clearly separate the work for format support from the actual functionality of the API. Making sure that you can continue to design and implement the basic API functionality (e.g. managing users, editing content, processing purchases, etc.) without tightly binding to an output format will go a long way toward making multiple format support safe and easy — even after initial release of your API into production.

The other challenge for this kind of work is to turn the process of converting internal domain data (e.g. data graphs and action details) to an output format into a consistent algorithm that works well for a wide range of formats. This will require some basic software pattern implementation as well as an ability to deal with a 'less than

www.it-ebooks.info

100% fidelity' between the domain model and the output model. We deal with this every day with HTML (HTML doesn't know anything about objects or strong typing) and need to adopt a similar approach with the common API formats, too.

> If your API design approach REQUIRES support for strong-typing *within* the message passed between client and server, it will not be easy and will probably be costly to support more than one output format. You should probably abandon the notion of multiple format support and stick with a proprietary object serialization pattern using `application/json` or `application/xml`. You can also skip the the rest of this chapter.

Finally, we'll need a mechanism for *selecting* the proper format for each incoming request. This, too, should be an algorithm we can implement consistently in our service. Preferably this will rely on existing information in HTTP requests and will not introduce some new custom meta-data that clients will need to support.

Ok, separate format processing, implement a consistent way to convert domain data into an output format, and identify request metadata to help us select the proper format. Let's start with separating the format processing from the domain.

Separating Format from Functionality

All too often I see service implementations that are bound too tightly to a single output format. This is a common problem for SOAP implementations. Usually because the developer tooling *leads* programmers into relying on a tight binding between the internal object model and the external output format. It is important to treat all formats (including SOAP XML output) as independent of the internal object model. This allows some changes in the internal model to happen without requiring changes to the external output format.

To manage this separation, we'll need to employ some *modularity* to keep the work of converting the domain model into a message external from the work of manipulating the domain model itself. This is using modularity to split up the assignment of work. Typically modularity is used to collect related functionality in a single place (e.g. all the functionality related to `users` or `customers` or `shoppingCarts`). The notion of using modularity as primarily a work assignment tactic comes from Davis Parnas' 1972 paper *On the Criteria to be Used in Decomposing Systems into Modules*. As Parnas states it…

> "'[M]odule' is considered to be a *responsibility assignment* rather than a subprogram. The modularizations include the design decisions which must be made before the work on independent modules can begin." [Emphasis in the original]
>
> —David Parnas

www.it-ebooks.info

Viewing the work of converting internal domain data into external output formats a *responsibility assignment* leads us to isolate the conversion process into its own module. Now we can manage that module seprately from the one(s) that manipulate domain data. A simple example of this clear separation might look like this:

```
var convert = new ConversionModule();
var output = convert.toHAL(domainData);
```

In the above imaginary pseudo-code, the conversion process is accessed via an instance of the `conversionModule()` that contains one or more public methods (e.g. `toHAL()`) that accept a `domainData` instance and produces the desired `output`. This is all quite vague right now, but at least we have a clear target for implementing safe, cheap, easy support for multiple output formats.

Once the functionality of the internal domain model is cleanly separated from the external format, we need some guidance on how to consistently convert the domain model into the desired output format. But before that, we'll need a pattern for matching selecting which format is appropriate.

The Selection Algorithm

An important implementation detail when supporting multiple output formats for an API is that of the output *selection* process. There needs to be some consistent algorithmic way to select the correct output format at runtime. The good news is that HTTP — still the most common application-level protocol for Web APIs — has this algorithm already defined: Content-Negotiation.

Section 3.4 of the HTTPbis specification (RFC7231) describes two patterns of content negotiation for "representing information":

Proactive
: The server selects the representation based on the client's preferences.

Reactive
: The server provides a list of possible representations to the client and the client selects the preferred format.

The most common pattern in use on the Web today is the Proactive one and that's what we'll implement in our Representor. Sepcifically, clients will send an `Accept` HTTP header that contains a list of one or more format preferences and the server will use that list to determine which format will be used in the response (including the selected format identifier in the server's `Content-Type` HTTP header).

A typical client request might be:

```
GET /users HTTP/1.1
Accept: application/vnd.hal+json, application/vnd.uber+json
...
```

www.it-ebooks.info

And, for a service that supports HAL but does not support UBER, the response would be:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.hal+json
...
```

It's All About Quality

The content negotiation examples shown in this book are greatly simplified. Client apps may include several media types in their accept list — even the `"/"` entry (which means "I accept everything!"). Also, the HTTP specification for the `Accept` header includes what is known as the `q` parameter that can qualify each entry in the accept list. Valid values for this paramter is a range of numbers from 0.001 (least-preferred entry) to 1 (most-preferred entry).

For example, this client request shows that, of the two acceptable formats, the HAL format is the most-preferred by this client app:

```
GET /users/ HTTP/1.1
application/vnd.uber+json;q=0.3, application/vnd.hal+json;q=1
```

So, that's what it looks like on the "outside" — the actual HTTP conversation. But what pattern is used *internally* to make this work on the server side? Thankfully, a solution for this kind of selection process was worked out in the 1990s.

A Solid STRATEGY

Many of the challenges of writing solid internal code can be summed up in a common *pattern*. And one of the most important books on code patterns is the 1994 "Design Patterns" book by Gamma, Helm, Johnson, and Vlissides. Those are rather tough names to remember and, over time, this group of authors has come to referred to as "The Gang of Four" or GoF. You'll sometimes even here people refer to the "Gang of Four book" when they are thinking of this impotant text.

Patterns in Architecture

The notion that architecture can be expressed as a common set of patterns was first written about by Christopher Alexander. His book *Timeless Way of Building* (1979) is an easy and thought-provoking read on how patterns play a role in physical architecture. It was his work on patterns that inspired the authors of the "Design Patterns" book and so many other software patterns books.

There are about twenty patterns in the GoF book, categorized into three types:

www.it-ebooks.info

- Creational Patterns
- Structural Patterns
- Behavioral Patterns

The pattern that will help us in our quest to implement support for multiple output formats for our API that is safe, cheap and easy is the STRATEGY behavioral pattern.

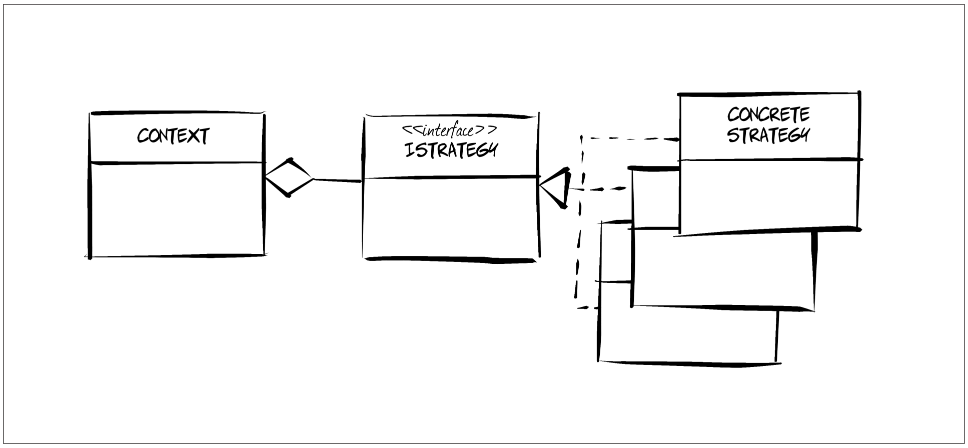


Figure 3-1. The STRATEGY Pattern

The intent of the STRATEGY pattern is to:

> "Define a family of algorithms, encapsulate each one, and make them interchangeable."
>
> —STRATEGY Pattern, *OODesign.com*

The STRATEGY pattern itself is made up of three distinct elements (called *participants*):

1. `Strategy`: Defines the interface common to all supported algorithms
2. `Context`: Uses the interface to call the appropriate `ConcreteStrategy`
3. `ConcreteStrategy`: Implements each concrete algorithm

For our use case, we'll implement the `Strategy` as a common way to describe internal domain objects, pass that to `Context` (along with the HTTP `Accept` header) and implement each format processor as a `ConcreteStrategy`.

That means we need to spend a few minutes on what the `ConcreteStrategy` might do and how it will process internal data.

www.it-ebooks.info

The TRANSFORM VIEW

A very succint version of the `ConcreteStrategy` idea as it can apply to HTTP responses appears in Martin Fowler's *Patterns of Enterprise Application Architecture*. Published in 2002, the book contains over 40 patterns that Fowler says "…are ones that I've seen in the field, usually on many different programming platforms." It's a valuable book that also has a great online website. One of the patterns in the is especially appropriate for our Representor discussion here. It the pattern Fowler calls the TRANSFORM VIEW. He describes the pattern this way:

"A view that processes domain data element by element and transforms it into HTML."

—Martin Fowler, *Patterns of Enterprise Application Architecture*

TRANSFORM VIEW or TEMPLATE VIEW?

Fowler's detailed description of the TRANSFORM VIEW pattern includes several comparisions between his TRANSFORM VIEW and TEMPLATE VIEW (a pattern closely tied to the Model-View-Controller MVC set of patterns). I won't cover the material here and it is well worth reading. Bascially, TEMPLATE VIEW works best when you are working from the format perspective and TRANSFORM VIEW works well when you're working from the domain data point of view.

While Fowler's pattern is specifically targeted for transforming domain data into the HTML format, it applies just as well to use cases requiring transformation into any standardized media type format like the ones used for Web APIs.

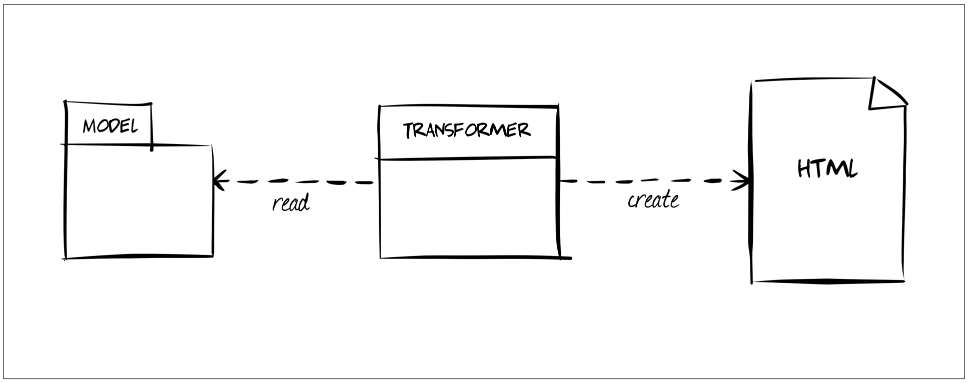


Figure 3-2. The TRANSFORM VIEW Pattern

So, when tackling the challenge of supporting multiple output formats for an API, the first step is to clearly separate the APIs functionality from the output format. Next, we

www.it-ebooks.info

need to apply the Gang-of-Four's STRATEGY pattnen to select formats and Fowler's TRANSFORM VIEW pattern to implement each conversion from donmain data to media type message.

So, armed with this background information, we can now look at a set of concrete implementation details to make it all happen.

A Server-Side Model

In this section, I'll walk through the high-level details of a working Representor implementation; the one that is used in all the services created for this book. Implementing a Representor means dealing with the following challenges:

1. Inspecting the HTTP request to identify the acceptable output formats for the current request
2. Using that data to determine which output format will be used
3. Converting the domain data into the target output format

The source code for this version of the 'TPS - Task Processing System' can be found in the associated github repo here: *https://github.com/LCHBook/simple-todo*. A running version of the app described in this chapter can be found here: *http://lchbook-ch02.herokuapp.com/* (TK: check URLs)

Handling the HTTP `Accept` Header

The first two items on that list are rather trivial to implement in any WWWW-aware codebase. For example, identifying acceptable output formats for a request means reading the `Accept` HTTP header. Here is a snippet of NodeJS code that does that:

```
// rudimentary accept-header handling
var csType = '';
var htmlType = 'text/html';
var csAccept = req.headers['accept'];
if(!csAccept|| csAccept==='*/*') {
  csType = htmlType;
}
else {
  csType = csAccept.split(',')[0];
}
```

Note the above code example makes two key assumptions:

1. If no `Accept` header is passed or the `Accept` header is set of "anything", the `Accept` header will be set to `text/html`, and

www.it-ebooks.info

2. If the `Accept` header lists more than one acceptable format, this service will just grab the first one listed.

This implementation is very limited. It does not support the use of `q` values to thelp the server better-understand client preferences and this service defaults to the `text/html` type for API responses. Both of these assumptions can be altered/improved through additional coding but I've skipped over that for this book today.

Implementing the STRATEGY Pattern

Now that we have the requested format value — the output *context* for this request — we can move on to the next step; implementing the STRATEGY pattern in NodeJS. For this book, I've created a simple module that uses a `switch … case` element that matches the request context string (the accepted format) with the appropriate `ConcreteStrategy` implementation.

The code looks like this:

```
// load representors ❶
var html = require('./representors/html.js');
var haljson = require('./representors/haljson.js');
var collectionJson = require('./representors/cj.js');
var siren = require('./representors/siren.js');

function processData(domainData, mimeType) {
  var doc;

  // clueless? assume HTML ❷
  if (!mimeType) {
    mimeType = "text/html";
  }

  // dispatch to requested representor ❸
  switch (mimeType.toLowerCase()) {
    case "application/vnd.hal+json":
      doc = haljson(object);
      break;
    case "application/vnd.collection+json":
      doc = collectionJson(object);
      break;
    case "application/vnd.siren+json":
      doc = siren(object);
      break;
    case "text/html":
    default: ❹
      doc = html(object);
      break;
  }
```

www.it-ebooks.info

```
    return doc;
  }
```

In the above code snippet, you can see that a set of *representors* are loaded at the top (see callout #1). The code in these modules will be covered below (TK: see *A Sample Representor*). Next (callout #2), if the `mimeType` value is not passed (or is invalid) it is automatically set to `text/html`. This is a bit of defensive coding. And then (at callout #3) the `switch … case` block that checks the incoming `mimeType` string with known (and supported) mime type strings in order to select the appropriate format processing module. Finally, in case an unknown/unsupported format is passed in, the `default` statement (callout #4) makes sure that the service runs the `html()` module to produce valid output.

We now have the basics of the Representor outlined. The next step is to actually implement each format-specific module (HTML, HAL, etc.). To solve this challenge, we need to take a side-road on our journey. One that establishes a general representor pattern.

General Representor Modules

In the STRATEGY pattern, each format module (`html()`, `haljson()`, etc.) is an instance of a `ConcreteStrategy` participant. While implementing these modules as domain-specific converters (e.g. user-to-html, user-to-hal, customer-to-html, customer-to-hal, etc.) would meet the minimum needs of our implementation, that approach will be tough to scale over time. Instead, what we need is a general-purpose format module; one that will *not* have any domain-specific knowledge. For example, the format modules used to handle `user` domain data will be the same format modules used to handle `customer` and `accounting` or any other domain-specific domain data.

To do that, we'll need to create a common interface for passing domain data into format modules. One that is independent of any single domain model.

The WeSTL Format

For this book, I've worked up common interface in the form of a standardized object model. One that service developers can quickly 'load' with domain data and pass to format modules. I also took the opportunity to reframe the challenge of defining interfaces for Web APIs. Instead of focusing on defining resources, I chose to focus on defining *state transitions*. For this reason, I've named this interface design the *Web Service Transition Language* or WeSTL (pronounced *wehs*'-*tul*).

www.it-ebooks.info

Curious about WeSTL?

I won't be able to go into too much depth on the *design* of the WeSTL format in this chapter. I want to focus instead on how we can use WeSTL to drive our general representation module implementation. Also, as I worked on this book, I continued to modify the WeSTL model to meet new challenges I encountered along the way. Even after all that work, WeSTL is still an unstable design prone to breaking changes. By the time you read this, WeSTL might have changed quite a bit! If you're curious about the thinking behind the WeSTL, check out the added Appendix (TK:link) and the related online content listed there.

When designing and implementing a Web APIs with WeSTL, the service developer collects up all the possible state transitions and describes them in the WeSTL model. By state transitions, I mean all the links and forms that could appear within any service response. For example, every response might have a link to the `home` page. Some responses will have forms allowing API clients to create new service data or edit existing data. There may even be a services responses that lists all the possible links and forms (state transtions) for the service.

Why State Transitions?

Focusing on state transitions may seem a bit unusual. First, the transition is the thing *between* states; it leads from one state to another. For example State A might be the 'home' page and State B might be the list of `users`. WeSTL documents don't describe State A or B. Instead, they describe the action that makes it possible to move from State A to State B. But this is also not quite correct. WeSTL documents do not indicate the starting state (A) or the ending state (B), just one possible way to move from *some* state to another. This focus on the actions that enable changes of state makes WeSTL (as far as I know) unique.

A simple example of how WeSTL can be used to describe transitions is shown below.

```
{
  "wstl" : {
    "transitions" : [
      {
        "name" : "home",❶
        "type" : "safe",
        "action" : "read",
        "prompt" : "Home Page",
      },
      {
        "name" : "user-list",❷
```

www.it-ebooks.info

```
          "type" : "safe",
          "target" : "user list"
          "prompt" : "List of Users"
        }
        {
          "name" : "change-password",
          "type" : "unsafe",
          "action" : "append"
          "inputs" : [❸
            {
              "name" : "userName",
              "prompt" : "User",
              "readOnly" : true
            },
            {
              "name" : "old-password",
              "prompt" : "Current Password",
              "required" : true,
            },
            {
              "name" : "old-password",
              "prompt" : "New Password (5-10 chars)",
              "required" : true,
              "pattern" : "^[a-zA-Z0-9]{5,10}$"
            }
          ]
        }
      ]
    }
  }
```

As you can see from the above WeSTL document, it contains three transition descriptions named `home`, `user-list` (callout #1), and `change-password` (#2). The first two transitions are marked `safe`. That means they doesn't write any data, only execute reads (e.g. HTTP GET). The thrid one, however (`change-password`) is marked `unsafe` since it writes data to the service (ala HTTP POST). You can also see several `input` elements described for the `change-password` transition (callout #3). These details will be used when creating an API resource for the User Manager service.

There are a number details left out in this simple example, but you can see how WeSTL works; it describes the transitions that can be used within the service. What's important to note is that this document does not define *Web resources* or constrain where (or even when) these transitions will appear. That work is handled by service developers elsewhere in the code.

So, this is what a WeSTL model looks like at "design-time"; before the service is up and running. Typically a service designer uses WeSTL in this mode. There is also another mode for WeSTL documents — "runtime." That mode is typically used when *implementing* the service.

www.it-ebooks.info

Runtime WeSTL

At runtime, an instance of the WeSTL model is created; one that contains only the valid transitions for a particular resource. This runtime instance also includes any data assocaited with that Web resource. In other words, WeSTL models at runtime reflect the current state of a resource — both the avaialble data and the appropriate transitions.

Creating a runtime WeSTL model in code might like this:

```
var transitions = require('./wstl-designtime.js');
var domainData = require('./domain.js');

function userResource(root) {
  var doc, coll, data;

  data = [];
  coll = [];

  // pull data for this resource❶
  data = domain.getData('user',root.getID());

  // add transitions for this resource❷
  tran = transitions("home");
  tran.href = root +"/home/";
  tran.rel = ["http:"+root+"/rels/home"];
  coll.splice(coll.length, 0, tran);

  tran = transitions("user-list");
  tran.href = root +"/user/";
  tran.rel = ["http:"+root+"/rels/collection"];
  coll.splice(coll.length, 0, tran);

  tran = transitions("change-password");
  tran.href = root +"/user/changepw/{id}";
  tran.rel = ["http:"+root+"/rels/changepw"];
  coll.splice(coll.length, 0, tran);

  // compose wstl model❸
  doc = {};
  doc.wstl = {};
  doc.wstl.title = "User Management";
  doc.wstl.transitions = coll;
  doc.wstl.data =  data;

  return doc;
}
```

As the code sample above shows, the `userResource()` function first pulls any associated data for the current resource — in this case, a single user record based on the ID value in the URL — as seen in callout #1, then pulls three transitions from the design-

www.it-ebooks.info

time WeSTL model (#2) and finally composes a runtime WeSTL model by combining the data, transitions, and a helpful title string (callout #3).

It should be pointed out that the only contraint the `wstl.data` element in this model is that it MUST be an array. IOt can be an array of JSON properties (e.g. name-value paris), an array of JSON objects, or even an array of *one* JSON object that is, itself, a highly nested graph. the WeSTL document MAY even include a property that points to a schema describing the `data` element. This might in the form of a JSON Schema document, RelaxNG, or some other schema description language.This schema information can be used by the general format module to help location and process the contents of the `data` element.

As of this writing, WeSTL is still an *unstable draft* and there is no definitive standard for describing complex `data` elements. You can view the most recent features of WeSTL in the associated github repo (TK:link).

This is how WeSTL allows service developers to define Web resources in a general way. First, service *designers* can create design-time WeSTL documents that describe all the possible transitions for the service. Second, sevice *developers* can use this design-time document as source material for constructing runtime WeSTL documents that include selected transitions plus associated runtime data.

Now we can finally write our general format modules.

A Sample Representor

Now that resources are represented using a generic interface using WeSTL, we can build a general format module that converts the standardized WeSTL model into an output format. Basically, the code accepts a runtime WeSTL document and then (as Fowler put it for his TRANSFORM VIEW) processes domain data element by element and transforms it into the target output format.

The example Representor shown here has been kept to the bare minium to help illustrate the process. It is not production-ready and should not be used without improvements that would make it more resilient at runtime. A fully-functional HAL Representor is included in the github repo associated with the chapter (TK: link).

To see how this might look, here is a high-level look at a simplified implementation of a general HAL representor.

```
function haljson(wstl, root, rels) { ❶
  var hal;
```

www.it-ebooks.info

```
  hal = {};
  hal._links = {};

  for(var o in wstl) {
    hal._links = getLinks(wstl[o], root, o, rels);
    if(wstl[o].data && wstl[o].data.length===1) {
      hal = getProperties(hal, wstl[o]);
    }
  }
  return JSON.stringify(hal, null, 2);❹
}

// emit _links object ❷
function getLinks(wstl, root, o, relRoot) {
  var coll, items, links, i, x;

  links = {};

  // list-level actions
  if(wstl.actions) {
    coll = wstl.transitions;
    for(i=0,x=coll.length;i<x;i++) {
      links = getLink(links, coll[i], relRoot);
    }

    // list-level objects
    if(wstl.data) {
      coll = wstl.data;
      items = [];
      for(i=0,x=coll.length;i<x;i++) {
        item = {};
        item.href = coll[i].meta.href;
        item.title = coll[i].title;
        items.push(item);
      }
      links[checkRel(o, relRoot)] = items;
    }
  }
  return links;
}

// emit root properties ❸
function getProperties(hal, wstl) {
  var props;

  if(wstl.data && wstl.data[0]) {
    props = wstl.data[0];
    for(var p in props) {
      if(p!=='meta') {
        hal[p] = props[p];
      }
```

www.it-ebooks.info

```
      }
    }
    return hal;
  }

  /* additional support functions appear here */
```

While this code example is just a high-level view, you should be able to figure out the important implementation details. First, the first argument of the top level function (`haljson()`) accepts a WeSTL model along with some runtime request-level data (callout #1). That function "walks" the WeSTL runtime instance and first, processes any links (transitions) in the model (callout #2) and then deals with any name-value pairs in the WesTL instance (callout #3). Once all the processing is done, the resulting JSON object (now a valid HAL document) is returned to the caller (#4).

An example of what ths code above would produce might be as follows…

```
{
  "_links" : {❶
    "self" : {
      "href": "http://localhost:8282/user/mamund"
    },
    "http://localhost:8282/rels/home": {
      "href": "http://localhost:8282/",
      "title": "Home",
      "templated": false
    },
    "http://localhost:8282/rels/collection": {
      "href": "http://localhost:8282/user/",
      "title": "All Users",
      "templated": false
    },
    "http://localhost:8282/rels/changepw": {
      "href": "http://localhost:8282/user/changepw/mamund",
      "title": "Change Password"
    }
  },
  "userName": "mamund", ❷
  "familyName": "Amundsen",
  "givenName": "Mike",
  "password": "p@ss",
  "webUrl": "http://amundsen.com/blog/",
  "dateCreated": "2015-01-06T01:38:55.147Z",
  "dateUpdated": "2015-01-25T02:28:12.402Z"
}
```

Now you can see how the WeSTL document has led from design-time mode to runtime instance and finally (via the HAL representor module) to the actual HAL document. The WeSTL transitions now appear in the HAL `_links` section (callout #1) and the related data for this `user` appears as name-value pairs called `properties` in HAL documents (starting at callout #2)

www.it-ebooks.info

Again, the above example is missing features in order to make this brief review more readable. And HAL is just one possible format implementation. Throughout the code for this book you'll find general formatters for a handful of formats. Hopefully, this short overview will give enough guidance to anyone who wishes to implement their own (possibly *better*) general representors for HAL and many other registered formats.

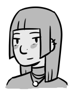
Summary

This chapter has been a bit of a diversion. I focused on the *server-side* Representor even though the primary aim of this book is to explor *client-side* hypermedia. But this Representor pattern is an important implementation approach and it will appear many times throughout the code examples in the book. And the process of working through programming lessons laerned in the past has been, I hope, helpful. We've built up a working example of a Representor by taking lessons from Parnas' "responsibility assignment" approach to modularity, the Content Negotiation features of HTTP, the STRATEGY pattern from the "Gang of Four", and Fowler's TRANSFORM VIEW pattern.

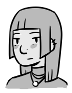
Not too shabby for a diverson.

Bob and Carol

"Hi, Carol. Ready to go over my results on the multi-format support?"

"Sure, Bob. I've been anxious to hear what you found out."


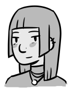
"Well, it was really interesting. After doing some brainstorming with the group -- including some of that 'reframing' technique you told me about -- we found a handful of previous work that applies to our challenge."

"Really? Let me guess. You found some old comp-sci papers from the 1980s with the solution already worked up, right?"


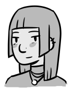
"Not quite. But close. First, we discussed the pros and cons of multi-format support for APIs. They all boiled down to some key points. If we want to

www.it-ebooks.info

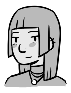
do this, we need to make adding format support safe, cheap, and easy."


"Right. You won't get this kind of thing right the first time and you'll need to be able to add new formats in the future without incurring lots of recode/redeploy costs."

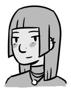
"Exactly. The good news is this kind of challenge has been faced lots of times before. We found some great background material to help us work up a solution."


"Now here comes the old comp-sci stuff!"

"Yep. First, David Parnas' descibed *responsibility assignment* as a modularity approach from a paper in 1972 so we pulled all the format stuff into a separate module. Next, we found two OO patterns -- the GoF STRATEGY pattern and Fowler's TRANSFORM VIEW pattern -- described a general solution to our challenge rather well. Finally, we can use HTTP's Content Negotiation feature as a way to select the proper output format at runtime."

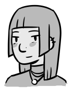

"Wow, so it really **was** already built for you!"

"Not quite. We had the theory, the next step was building the module -- the Representor. For that we needed to establish a common interface (from the STRATEGY pattern) and that meant inventing what we're calling the *Web Service Transition Language* or WeSTL. It's a model for describing transitions and even passing resource data from the internal service domain to the format-specific modules."


"Huh. We can talk about this transition model later. But, now you have a full GoF STRATEGY pattern implementation including the `Context`, `iStrategy`, and the `ConcreteStrategy` participants, right? That's cool."

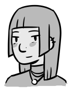
www.it-ebooks.info

"Yep, the whole team contributed and we're all happy with how it's going so far. Though this was a bit of a diversion, I'm really glad we took a couple days to look into this challenge and were able to come up with a loosely-coupled solution."



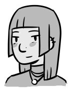
"Well, that's great Bob. Just one more question: What format did you finally select for the service?"

"Ha! That's the fun part. We don't have to pick just one. Now that we have this pattern implemented, we'll be looking for suggestions and feeback as we expand the list of supported formats past the default HTML."



"Excellent. Then my first request will be to get the API to support plain JSON objects, Bob."

"No problem. I think the team is already working on that one now."



"OK, then. I guess I need to get *my* team working on the JSON client right away."

References

1. Standard Generalized Markup Language (SGML) is documented in ISO 8879:1986. It is, howwever, based on IBM's *GML* format from the 1960s.
2. The "Simple Object Access Protocol (SOAP) 1.1" specification was published as a W3C Note in May 2000. In the W3C publication model, Notes have no standing as a recommendation of the W3C. It wasn't until the W3C published the SOAP 1.2 Recommendation in 2003 that SOAP, technically, was a 'standard.'
3. Crockford's 50 minute talk "The JSON Saga" was described as "The True Story of JSON." An unoffical transcript of this talk is available at [TK:link]
4. CSV was first specified by the IETF in RFC4180 "Common Format and MIME Type for Comma-Separated Values (CVS) Files" in 2005. This was later updated

www.it-ebooks.info

by RFC7111 (to add support for URI fragments) and additional CSV-related efforts have focused on supporting additional semantics.

5. The ["Atom Syndication Format" (*https://tools.ietf.org/html/rfc5023* and the "Atom Publishing Protocol (RFC5023) form a unqiue pair of specifications that outline both the document format and read/write semantics in different RFCs. There are also a handful of RFCs defining Atom format extensions.
6. More on the Open Data Protocol (OData) can be found at the OData web site.
7. Mike Kelly's "Hypertext Application Language (HAL) has proven to be one of the more popular of the *hypermedia-style* formats (as of this writing).
8. RFC6570 specifies URI Templates.
9. The "Collection+JSON" format was registered with the IANA in 2011 and is "a JSON-based read/write hypermedia-type designed to support management and querying of simple collections."
10. Both Siren and Zetta are projects spearheaded by Kevin Swiber.
11. As of this writing the Universal Basis for Exchanging Representations (UBER) is in stable draft stage and has not been registered with any standards body.
12. Beck's 1997 article "The Past and Future of Cognitive Therapy" describes his early experiences that led to what is now known as cognitive reframing.
13. Parnas' "On the Criteria to be Used in Decomposing Systems into Modules" is a very short (and excellent) article written for the ACM in 1972.
14. Details on HTTP Content Negotiation are covered in Section 3.4 of RFC7231. One of a series of HTTP-related RFC's (7230 through 7240).
15. The full title of the "Gang of Four" book is "Design Patterns: Elements of Reusable Object-Oriented Software" by Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
16. The STRATEGY pattern from the GoF book is nicely summarized at the Object-Oriented Design web site.
17. A good source for learning more about Christopher Alexander and his work can be found at the Pattern Language web site.
18. Martin Fowler's TRANSFORM VIEW is covered in his online catalog based on the 2002 book, Patterns of Enterprise Application Architecture.

Images Credits

1. Diogo Lucas, Figures 1 & 2

www.it-ebooks.info

www.it-ebooks.info

CHAPTER 4

Versioning and the Web

"Everything changes and nothing stands still."

—Heraclitus

Bob and Carol

"Carol, I've been starting to worry that we've missed something very important in our API design."

"Oh? What's that Bob?"

"Versioning."

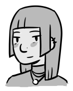
"You mean handling changes in the API over time, right?"

"Right. I think we need to account for that in our design."

"Hmmm. You know, now that I think about it, we've updated the API a couple times and never once talked about `versioning.' Wonder why that is."

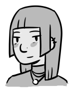
www.it-ebooks.info

"Well, Carol, I think we just forgot about it. We were in too much of a hurry to publish a working API and didn't think about the long-term implications."



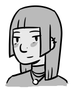
"Maybe. But, you know, I've not really *missed* versioning at all. I mean, things seem to be working fine, right?"

"Well, that's true for now. We're doing mostly small changes right now and don't have too many independent clients running against the API. But what happens in a few months or even a year from now?"



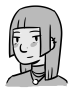
"I suspect the same thing happens, Bob. We keep making small changes and things keep working just fine."

"Well, I'm not so sure about that. I think we are going to run into problems at some point unless we address this now."



"OK, I see your point. So, let's take a look at it. We're using HTTP so far, right? HTTP has had a couple version updates over the last twenty or so years."

"That's true, Carol. And it all seemed to go pretty well. Hmmm. I think HTML is another example, right? Lots of versions there."

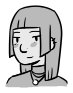


"Right. That makes me wonder what it is about the design of HTTP and HTML that made it possible to change over time without breaking existing implemenations."

"Huh. You've got me thinking here, Carol. Let's not make any changes to our API design yet. I want to do some more research on this topic and see what comes up."

www.it-ebooks.info

“Sounds good to me, Bob. Spend a few days on this and let’s get back together early next week.”



Whether you are in charge of designing, implementing, or just maintaining a Web API, at some point you start to think about how to handle change over time. The common word used in the API world is *versioning*. This idea of versioning is so deeply ingrained in the idea of API design that it is considered part of the Common Practice for APIs. Many API designers include versioning information (e.g. numbers in the URL, HTTP headers, or within the response body) without too much thought about the assumption behind that practice.

And there are lots of assumptions behind the common practices of “Versioning APIs.” Chief among them is the assumption that *any* change that is made to the API should be explictly noted in the API itself (usually in the URL). Another common assumption is that *all* changes are breaking changes — that failing to explicity signal changes means someone, somehwere will experience a fatal error. One more assumption worth mentioning here is that it is not possible to make meaningful changes to the functionality of Web APIs *unless* you make a breaking change.

Finally, there is enough uncertainty associated with the lifecycle of Web APIs that many API developers decide to simply hedge their bets by adding versioning information in the API design *just in case* it might be needed at some future point. This is a kind of Pascal’s Wager for API design.

Pascal’s Wager

Blaise Pascal, noted 17th century philospher, mathematician, and physicist is credited with creating Pascal’s Wager. A simplified version of this argument is that, when faced with a decision that cannot be made using logic alone, “a Game is being played… where heads or tails will turn up.” In his case, he was illustrating, since we do not know if there is a God, we should ‘bet’ that one exists since that is the bet with a better outcome.

Although his argument was more nuanced — and others have made similar observations about the nature of uncertainty — Pascal’s Wager has become a *meme* that essentially states “When in doubt, hedge your bets.”

Ok, with this as a background, let’s look into the assumptions behind the Versioning argument *and* some evidence of how some Web-related technology handles change over time.

www.it-ebooks.info

Versioning for the Internet

The idea that handling change over time on the Web means employing the "explixit versioning" technique flies in the face of some very important evidence about how things have worked on the Internet (not just the Web) over the last couple decades. We'll take some time here to look at just a few examples.

The examples we'll explore here are one of the foundational transport-level protocols (TCP/IP), the most common application-level protocols (HTTP), and HTML — the common markup language for the WWW. They have each undergone important modification over years and all without causing any major "breakage" for exsiting implementations.

Each of our examples are uniqiue but they all have a common design element that can help us in our understanding of how to handle change over time on the Web.

TCP/IP's Robustness Principle

The TCP/IP protocol is an essential part of how the Internet works today. In fact, TCP/IP is actually two related protocols; the Transmission Control Protocol (TCP) and the Internet Protocol (IP). Together they are sometimes referred to as the Internet Protocol Suite. Computer scientist, Alan Kay, as called TCP/IP "a kind of universal DNA of [the Internet]." Kay also has pointed out that the Internet has "never been stopped since it was first turned on in September, 1969." That means that this set of protocols has been working 24 hours a day, seven days a week for over forty years without a "restart." That's pretty amazing.

Kay's point is that the *source code* for TCP/IP has been changed and improved over all these years without the need to shut down the Internet. There are a number of reasons for this and one of the key elements is that it was *designed* for this very situation — to be able to change over time without shutting down. Kay has been quoted as saying that the two people credited with designing TCP (Bob Kahn) and IP (Vint Cerf) "knew what they were doing."

One bit of evidence of this 'knowing what to do' can be found in a short section of the TCP specification; section 2.10 with the title *The Robustness Principle*. The full text of this section is:

> "TCP implementations will follow a general principle of robustness: be conservative in what you do, be liberal in what you accept from others."
>
> —RFC793

The authors of this specification understood that it is important to design a system where the odds are tilted in favor of completing message delivery successfully. To do that, implementors are told to be careful to craft valid messages to send. Implementors are also encouarged to do their best to accept incoming messages — even if they

www.it-ebooks.info

are not quite ‘perfect’ in their format and delivery. When both things are happening in a system, the odds of messages being accepted and processed improves. TCP/IP works, in some part, because this principle is baked into the specification.

Postel’s Law

The Robustness Principle is often referred to as *Postel’s Law* because Jon Postel was the editor for the RFC that described the TCP protocol.

One way to implement Postel’s Law when building hypermedia-style client applications is to pass service responses through a routine that *converts* the incoming message into an internal representation (usually an object graph). This conversion process should be implemented in a way that allows successful processing even when there are flaws in the response such as missing default values that the converter can fill in or simple structural errors such as missing closing tags, etc. Also, when forming outbound requests — especially requests that will send an HTTP body (e.g. POST, PUT, and PATCH) — it is a good idea to run the composed body through a strict validation routine that will fix up formatting errors in any outbound messages.

Below is a bit of pseudo-code that illustrates how you can implement Postel’s Law in a client application.

```
// handling incoming messages
httpResponse = getResponse(url);
internalObjectModel = permissiveConverter(httpResponse.body);

...

// handling outgoing messages
httpRequest.body = strictValidator(internalObjectModel);
sendRequest(httpRequest);
```

So TCP teaches us to apply the Robustness Principle to our API implementations. When we do that, we have an improved likelihood that messages sent between parties will be accepted and processed.

HTTP’s MUST IGNORE

The HTTP protocol has been around a long time. It was runningin its earliest form in 1990 at CERN labs and has gone through several significant changes in the last 25 years. Some of the earliest editions of the HTTP specification made specific reference to the need for what was then called “client tolerance” in order to make sure that client applications would continue to function even when the responses from Web servers were not returning valid HTTP responses — these were called “deviant servers” in a special note linked to the 1992 draft of the HTTP specs.

www.it-ebooks.info

A key prinicple used in the early HTTP specifications is the MUST IGNORE directive. In its simplist form, any element of a request that the receiver does not understand, must be ignored without halting any processing of that request. Message processing should continue as if that portion of the message does not exist.

The final HTTP 1.0 documentation (RFC1945) has several places where this principle is documented. For example, in the section on HTTP Headers, it reads…

> "Unrecognized header fields should be ignored by the recipient and forwarded by proxies."
>
> —RFC1945

Note that in the above quote, the MUST IGNORE princple is extended to also include instructions for proxies to forward the unrecognized headers on to the next party. Not only should the receiver not reject messages with unknown headers but, in the case of proxy servers, those unknown headers are to be included in any forwarded messages. The HTTP 1.0 specification (RFC1945) contains eight separate examples of the MUST IGNORE principle. The HTTP 1.1 specifcation (RFC2616) has more than 30 examples.

MUST IGNORE or MAY IGNORE?

Throughout this section of the chapter I use the phrase MUST IGNORE when referring to the principle in the HTTP specification documents. This name is also used by David Orchard in his blog article *Versioning XML Vocabularies*. While the HTTP specs use the word 'ignore' many times, not all uses are prefaced by 'must.' In fact, some references to the ingore directive are qualified by the words MAY or SHOULD. The name MUST IGNORE, however is commonly used for the general principle of ignoring what you don't understand without halting processing.

Suporting the MUST IGNORE principle for Web clients means that incoming messages are not rejected when they contain elements that are not understood by the client application. This means the message-processing for Web clients needs to be tolerant of unrecognized elements in a response. The easiest way to achieve that is to code the clients to simply look for and process the elements in the message that they "know".

For example, a client may be coded to know that every incoming message contains three root elements: `links`, `data`, and `actions`. In a JSON-based media type, the response body of a this kind of request might look like this:

```
{
 "links" : [...],
 "data" : [...],
```

www.it-ebooks.info

```
     "actions" : [...]
    }
```

Some pseudo-code to process these messages might look like this:

```
WITH message DO
  PROCESS message.links
  PROCESS message.data
  PROCESS message.actions
END
```

However, that same client application might get a response body that contains an additional root-level element named `extensions`:

```
{
 "links" : [...],
 "data" : [...],
 "actions" : [...],
 "extensions" : [...]
}
```

In a client that honors the MUST IGNORE principle, this will not be a problem because the client will simply ignore the new element and continue to process the message as if the `extensions` element does not appear at all. This is an example of MUST IGNORE at work.

Structural vs. Semantic Change

It is important to point out that adding new elements to the message (like the example given here) is *structural* change. We need to design formats and implement parsers so that any new change to the structure is easily and safely ignored. However, if you want to be able to add data or action elements to responses that will *not* be ignored, you need to use another form of change — what I call *semantic* change. Adding or removing data elements and/or changing the links or forms that appear within a response is changing the semantics of the message, not the structure. Semantic changes SHOULD be automatically picked up by the client app and parsed/rendered as usual.

When you want changes within a response to be IGNORED, make a *structural* change. When you want changes to be automatically included, make a *semantic* change instead.

So HTTP's MUST IGNORE principle shows us that we need to be able safely to process a message even when it contains portions we do not understand. This is similar to Postel's Law from the TCP specifcation. Both rules are based on the assumption that some percentage of incoming messages will contain elements that the receiver has not been programmed to 'understand.' When that happens, the processing should

www.it-ebooks.info

not simply stop. Instead processing should continue on as if the unrecognized element had never appeared at all.

HTML's Backward Compatibility

HTML is another example of a design and implementation approach that accounts for change over time. Like HTTP, the HTML media type has been around since the early 1990s. And, like HTTP, HTML has undergone quite a few changes over that time. And yet, throughout all those changes — from Tim Berner's Lee's intial "HTML Tags" document in 1990, soon known as HTML 1.0 on up to the current HTML5 — those many changes have been guided by the principle of *Backward Compatibility*. Every attempt has been made to only make changes to the media type design that will *not* cause HTML browsers to halt or crash when attempting to process an HTML document.

The Earliest Known HTML Document

The earliest known HTML document is from November 1990 and is still available on the Web today. It was crafted two weeks before Tim Berners-Lee and his CERN colleague, Robert Cailliau attended ECHT '90 — the European HyperText Convention — in held in Paris. The entire HTML document looks like this:

```
<title>Hypertext Links</title>
<h1>Links and Anchors</h1>
A link is the connection between one piece of
<a href=WhatIs.html>hypertext</a> and another.
```

The fact that this page still renders in browsers today; 25 years later is a great example of how both message design (HTML) and client implementation principles (Web browsers) can combine to support successful Web interactions over decades.

From almost the very beginning, HTML was designed with both Postel's Law and HTTP's MUST IGNORE in mind. Berners-Lee makes this clear in one of the early working documents for HTML:

> "The HTML parser will ignore tags which it does not understand, and will ignore attributes which it does not understand…"
>
> —Berners-Lee 1992

What's interesting here is that this type of guidance shows the message designers (those defining HTML) are also providing specific guidance to client implementors (those coding HTML parsers). This principle of advising implementors on how to use and process incoming messages is an important feature of Internet standards in general. So important that an IETF document (RFC2119) establishing just *how* specifications should pass this advice to implementors. This document defines a set of special

www.it-ebooks.info

word for giving directive advice. They are "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL." Other standards bodies have adopted a similar approach to passing on guidance to implementors.

So, after reviewing lessons from TCP, HTTP, and HTML, we can come up with some general guidance for designing and maintaining APIs that need to support change over time.

Guidelines for Non-Breaking Changes

Dealing with change-over-time is best dealt with in a set of principles — a kind of change *aesthetic*. There is no one single action to take, design feature to include (or exclude), etc. Another key thing to keep in mind is that change will *always* occur. You can certainly use energy and effort to stave off change (e.g. "I know you want that feature in the API but we are not going to be able to make that change this year."). You can even work around change with a 'hack' (e.g. "Well, we don't support that, but you can get the same effect if you first, write a temp record and then filter based on the change-date of the temp file."). There are other ways to avoid facing change, but almost all long-lived and oft-used APIs will experience pressure to change and designing-in the ability to handle select types of change can reduce the stress, cost, and danger of changing APIs over time.

So, for those who are no longer attempting to stave of the inetivable changes to your APIs, here is some general guidance for Web API designers, service-providers, and comsumer-clients. I've used all these options in the past and found them helpful in many cases.

API Designers

For those in change of *designing* APIs and message formats, it is important to understand enough about the general problem area to get a sense of what types of changes are likely to be needed over time as well as design interfaces that lead implementors down the 'happy path' of creating API services and clients that are able to handle the common changes.

Taking into account the likely changes over time is tricky, but important. We saw this kind of thinking when we reviewed the way HTML is documented and designed(TK:internal ref). Paul Clements, one of the authors of the book *Software Architecture in Practice* claims that those who work to in software architecture have a responsibility to deal with change as a fundamental aspect of thier design:

> "The best software architecture *knows* what changes often and makes that easy."
>
> —Paul Clements

www.it-ebooks.info

With this in mind here are three valuable princicples for those tasked with designing Web APIs.

Promise Media Types, Not Objects

Over time, object models are bound to change — and these models are likely to change often for new services. Trying to get all your service consumers to learn, and track all your object model changes is not a good idea. And, even if you *wanted* all API consumers to keep up with your team's model changes, that means your feature velocity will be limited to that of the slowest API consumer in your ecosystem. This can be especially problematic when your API consumers are customers and not fellow colleagues within the same company.

Instead of exposing object models in your APIs, promise standard message formats (e.g. HTML, Atom, HAL, Cj, Siren, etc.). These formats don't require consumers understand of your service's interal object models. That means you're free to modify your internal model w/o breaking your promise to API consumers. This means providers will need to handle the task of translating internal domain data into external message formats but we covered that already in (TK:ch02).

Well-designed formats SHOULD allow API designers to safely introduce *semantic* changes (message content) and well-implemented API consumers will be able to parse/render these content changes without the need for code updates. These same formats MAY support *structural* changes to message in order to safely introduce changes that can be ignored by clients that do not understand them.

Document Link Identifiers, Not URLs

Your API SHOULD NOT bake static URLs into the design. URLs are likely to change over time. Especially in cases where your initial service is running in a test bed or small online community to start. Tricking API consumers into backing explicit URLs into their source code increases the likelihood that code will become obsolete and forces consumers into making redeployments if and when your URLs are updated.

Instead, your API design SHOULD promise to support a named operation (`shoppingCartCheckOut`, `computeTax`, `findCustomer`) instead of promising exact addresses for those operations (*`http://api.example.org/findCustomer`*). Documenting (and promising operations by name is a much more stable and maintainable design feature.

Remember, if you want new operations to be ingored by existing clients, make it part of the *structure* of the message (e.g. `<findCustomer … />`). However, when you want the operation to be automatically parsed and/or rendered, favor formats that allow you to include the operation identifiers as part of the message's *semantic* content (e.g.

www.it-ebooks.info

`<operation name="findCusomter" … />`. Good candidates for semantic identifers are properties such as `id`, `name`, `class` and `rel`.

Publish Vocabularies, Not Models

The notion of *canoncial models* has been around a long time — especially in large enterprise IT shops. The hope is that, with enough hard work, a single grand model of the company's business domain will be completely defined and property described so that everyone (from the business analyst on through to the front-line developer) will have a complete picture of the entire company's domain data. But this never works out.

The two things conspiring against canonical models are 1) scope, and 2) time. As the scope of the problem grows (e.g. the company expands, product offering increase, etc.) the model becomes unwieldy. And, as time goes on, even simple models experience modifications that complicate a single-model view of the world. The good news is there is another way to solve this problem: vocabularies.

Once you move to promising formats instead of object models, the work of providing shared understanding or your API domain's data and actions needs to be kept somewhere else. The great place for this is in a *vocabulary*. Eric Evans refers to this using the name UBIQUITOUS LANGUAGE — a common rigorous language shared between domain experts and system implementors. By focusing on a shared vocabulary designers can constant probe domain experts for clarification and developers can implement features and share datra with a high degree of confidence.

Another important reason to rely in vocabularies is that you can define consistent 'binding' rules between the vocabulary terms and the output formats used in your API. For example, you might document that data element names in the vocabulary will ALWAYS appear in the `class` property of HTML responses and the `name` property of Collection+_JSON responses, and so forth. This also helps API providers and consumers write general-use code that will work even when new vocabulary terms are added over time.

So, when designing APIs you should:

- Promise Media Types, Not Objects
- Document Link Identifers, Not URLs
- Publish Vocabularies, Not Models

Server Implementors

Like API designers, service implementors have a responsibility to create a software implementation that can account for change-over-time in an effective way. This

www.it-ebooks.info

means not only making sure local service changes can be made without any unnecessary complexity or instablity. It also means that changes made to the service over time are not likely to break existing client implementations — even implementations that the service knows nothing about.

Maintaining backward compatibility is the primary prinicple for service implementors when supporting change-over-time. Essentially, this constrains the types of changes a service can make over time. Only those changes which will not invalidate existing implementations. We saw this principle in play, for example, when reviewing HTTP's design principles (TK:internal ref).

With this in mind, here are three principles I've used to help support change-over-time while reducing the likelihood of breaking existing implementations.

Don't Take Things Away

One of the most important aspects of maintaining a backward-compatible service implementation is DON'T TAKE THINGS AWAY. In API projects I work on, I make this an explicit promise. Once API consumers know that you will not take things away, the value of your API will go up. That's because API consumers will be assured that, even when you add new data elements and actions to the API, the existing API consume code will still be valid.

One big reason to make this promise is that it allows API consumer teams to move at their own pace. They don't need to stop current sprints or feature work in order to deal with potentially breaking changes from some 'up-stream' service they are using in their application. That also means the services don't need to wait for the slowest API consumer team before they introduce new elements and actions in the API. This loose-coupling in the API update process can result in overall faster development processes since it reduces potential 'blocking' scenarios.

www.it-ebooks.info

We'll Find Another Provider, Thanks

If you create an API provider implementation that REQUIRES all consumer apps to upgrade at the same pace as the service does, you are essentially attempting to control the way API consumer teams operate. You are forcing them to work at *your* pace, on features *you* deem important. While this may be acceptable for internal teams, it can spell real trouble if your API consumers are third parties or customers. APIs that mess with customer release schedules, cause unwanted upgrade costs, and introduce features the consumer doesn't need right away will be seen as extra costs and burdens. I've heard of external API consumer teams making the decision to *stop* using APIs with tightly-coupled upgrade schedules and looking for APIs that allow the customer to update on their own scheule.

The more external API consumers you support, the more important it is to promis that you WILL NOT TAKE THINGS AWAY.

So, what does this backward-compatibility promise look like? Here's an example I learned from a Jason Rudolph at Github. This is example of what they call "evolutionary design" for APIs. He says:

> "When people are building on top of our API, we're really asking them to trust us with the time they're investing in building their applications. And to earn that trust, we can't make changes [to the API] that would cause their code to break."
>
> —Jason Rudolph, *Github*

Here's an example of their "evolutionary design" in action. They supported an API response that returned the current status of an account's request rate limit. It looked liked this:

```
*** REQUEST ***
GET rate_limit
Accept: application/vnd.github+json
...

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.github+json
...

{
  "rate" {
    "limit" : 5000,
    "remaining : 4992,
    "reset" : 1379363338
  }
}
```

www.it-ebooks.info

Over time, the github team learned that this response was more coarse-grained than was needed. It turns out they wanted to separate search-related rate limits from typical core API calls. So the new design would look like this:

```
*** REQUEST ***
GET rate_limit
Accept: application/vnd.github+json
...

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.github+json
...

{
  "resources" : {
    "core" : {
      "limit" : 5000,
      "remaining : 4992,
      "reset" : 1379363338
    },
    "search" : {
      "limit" : 20,
      "remaining : 19,
      "reset" : 1379361617
    }
  }
}
```

So, they had a dilemma on their hands. How could they make this important change to the interface without breaking existing implementations? Their solution was, I think, quite smart. Rather than *changing* the response body, they *extended* it. The new response for the `rate_limit` request now looks like this:

```
*** REQUEST ***
GET rate_limit
Accept: application/vnd.github+json
...

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.github+json
...

{
  "rate" : {
    "limit" : 5000,
    "remaining : 4992,
    "reset" : 1379363338
  },
  "resources" : {
    "core" : {
```

www.it-ebooks.info

```
        "limit" : 5000,
        "remaining : 4992,
        "reset" : 1379363338
      },
      "search" : {
        "limit" : 20,
        "remaining : 19,
        "reset" : 1379361617
      }
    }
  }
```

Notice that Github applied a *structural* change to the response (TK above). This is the change that can be safely ignored by clients that don't understand it. This is just one example of implementing backward-compatibility by not taking things away. The same genreal appraoch can be made for links and forms in a response, too.

Don't Change The Meaning of Things

Another important backward-compatiblity principle for service providers is DON'T CHANGE THE MEANING OF THINGS. That means, once you publish a link or form with an identifier that tells API consuemers what is returned (e.g. `<a href="…" rel="users" />` returns a list of users), you SHOULD NOT later use that same identifer to return something completely different (`a href="…" rel="users" />` later only returns a list of inactive users. Consistency in the *meaning* of a link identifer and/or data element is very important for maintaining backward-compatibility over time.

In cases where you want to add some new functionality to the API, it is much better to make a *semantic* change by adding the new functionality. And you should do this without removing the existing functionality. To use the above example, if you want to add the ability to return a list of inactive users, it is better to introduce an additional link (and identifier) while maintaining the existing one.

```
*** REQUEST ***
GET /user-actions HTTP/1.1
Accept: application/vnd.hal+json
...

**** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.hal+json
...

{
  "_links" :  {
    "users" : {"href" : "/user-list"},
    "inactive" : {"href" : "/user-list?status=inactive"}
  }
```

www.it-ebooks.info

```
}
...
```

In cases where the above response is used to create a human-driven UI, both the links will appear on the screen and the person running the app can decide which link to select. In the case of a service-only interface (e.g. some middleware that is tasked with collecting a list of users and processing it in some unique way), the added semantic information (e.g. the `inactive` link) will not be "interesting" and will be ignored. In both cases, this maintains backward-compatiblity and does not breaking exsiting implementations.

All New Things are Optional

Another important change-over-time principle for service implementors is to make sure ALL NEW THINGS ARE OPTIONAL. This means any new arguments (e.g. filters or update values) cannot be treated as requried elements. Also, any new functionality or workflow steps (e.g. you introduce a new workflow step between login and checkout) cannot be required in order to complete the process. Essentially, this boils down to making sure any new processing steps are optional.

One example of this is similar to the github case above (TK:ref). It is possible that, over time, you'll find that some new filters are needed when making requests for large lists of data. You might even want to introduce a default `page-size` to limit the load time of a resource and speed up responsiveness in your API. Here's how filter form looks before the introduction of the `page-size` argument:

```
*** REQUEST ***
GET /search-form HTTP/1.1
Accept: application/vnd.collection+json
...

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.collection+json
...

{
  "collection" : {
    "queries" : [
      {
        "rel" : "search"
        "href" : "/search-results",
        "prompt" : Search Form",
        "data" : [
          {
            "name" : "filter",
            "value" : "",
            "prompt" : "Filter",
            "required" : true
```

www.it-ebooks.info

```
          }
        ]
      }
    ]
  }
}
```

And here is the same response after introducing the `page-size` argument:

```
*** REQUEST ***
GET /search-form HTTP/1.1
Accept: application/vnd.collection+json
...

*** RESPONSE ***
200 OK HTTP/1.1
Content-Type: application/vnd.collection+json
...

{
  "collection" : {
    "queries" : [
      {
        "rel" : "search"
        "href" : "/search-results",
        "prompt" : Search Form",
        "data" : [
          {
            "name" : "filter",
            "value" : "",
            "prompt" : "Filter",
            "required" : true
          },
          {
            "name" : page-size",
            "value" : "all",
            "prompt" : "Page Size",
            "required" : false
          }
        ]
      }
    ]
  }
}
```

In the updated rendition, you can see the new argument (`page-size`) was explicitly marked optional (`"required" : false`). You can also see the a default value was provided (`"value" : "all"`). This may seem a bit conterintuitive. The update was introduced in order to *limit* the number of records sent in responses. So why set the default value to `"all"`? It is set to `"all"` because that was the initial promise in the first rendition of the API. We can't change the *meaning* of this request now to only

www.it-ebooks.info

include *some* of the records. This follows the DON'T CHANGE THE MEANING OF THINGS principle from above (TK:ref).

So, as service implementors, you can go a long way toward maintaing backward compatibility by supporting these three principles:

- Don't Take Things Away
- Don't Change the Meaning of Things
- Make New Things Optional

Client Implementors

Those who on the consuming end of APIs also have some responsibility to support change-over-time. We need to make sure we're prepared for the backward-compatible features employe by API designers and service implementors. But we don't need to wait for designers and providers to make changes in their own work before creating stable API consumer apps. We can adopt some of our own principles for creating robust, resilient API clients. Finally, also need to help API designers and service providers understand the challenges of creating adaptable API consumers by encouraging them to adopt the kinds of principles described here when they create APIs.

Code Defensively

The first thing API consumers can do is adopt a coding strategy that protects the app from cases where expected data elements and/or actions are missing in a response. This is can be accomplished when you CODE DEFENSIVELY. You an think of this as honoring *Postel's Law* (TK:ref) by being "liberal in what you accept from others." There are a couple very simple ways to do this.

For example, when I write client code to process a response, I almost always include code that first checkes for the *existence* of an element before attempting to parse it. Here's some client code that you'll likely find the in examples associated with this book.

```
// handle title
function title() {
  var elm;

  if(hasTitle(g.cj.collection)===true) { ❶
    elm = d.find("title");
    elm.innerText = g.cj.collection.title;
    elm = d.tags("title");
    elm[0].innerText = g.cj.collection.title;
  }
}
```

www.it-ebooks.info

You can see that I first check to see if the `collection` object has a `title` property (callout #1). If yes, I can continue processing it.

Here’s another example where I supply local default values for cases where the service response is missing expected elements (callout #1) *and* check for the existence of a property (callout #2):

```
function input(args) {
  var p, lbl, inp;

  p = node("p");
  p.className = "inline field";
  lbl = node("label");
  inp = node("input");
  lbl.className = "data";
  lbl.innerHTML = args.prompt||""; ❶
  inp.name = args.name||""; ❶
  inp.className = "value "+ args.className;
  inp.value = args.value.toString()||""; ❶
  inp.required = (args.required||false); ❶
  inp.readOnly = (args.readOnly||false); ❶
  if(args.pattern) { ❷
    inp.pattern = args.pattern;
  }
  push(lbl,p);
  push(inp,p);

  return p;
}
```

There are other examples of coding defensively that I won’t include here. The main idea is to make sure that client applications can continue functioning even when any given response is missing expected elements. When you do this, even most *unexpected* changes will not cause your API consumer to crash.

Code to the Media Type

Another important principle for building resilient API consumer apps is to CODE TO THE MEDIA TYPE. Essentially, this is using the same approach that was discussed in Chapter XX (tk:link). Only this time, instead of focusing on creating a pattern for converting internal domain data into an output format (via the Representor), the opposite is the goal for API consumers: convert a standardized output format into an internal domain model. By doing this, you can go a long way toward protecting your client application from both *semantic* and *structural* changes in the service responses.

For all the client examples I implement in this book, the media type messages (HTML, HAL, Cj, and Siren) are converted into the same internal domain model: the HTML Document Object Model or DOM. The DOM is a consistent and easy model

www.it-ebooks.info

to work with and writing client-side javascript for it is the way most browser-based API clients work.

Here is a short code snippet that shows how I convert Siren `entities` into HTML DOM objects for rendering in the browser:

```
// entities
function entities() {
  var elm, coll;
  var ul, li, dl, dt, dd, a, p;

  elm = d.find("entities");
  d.clear(elm);

  if(g.siren.entities) {
    coll = g.siren.entities;
    for(var item of coll) {
      segment = d.node("div"); ❶
      segment.className = "ui segment";

      a = d.anchor({ ❷
        href:item.href,
        rel:item.rel.join(" "),
        className:item.class.join(" "),
        text:item.title||item.href});
      a.onclick = httpGet;
      d.push(a, segment);

      table = d.node("table"); ❸
      table.className = "ui very basic collapsing celled table";
      for(var prop in item) {
        if(prop!=="href" &&
          prop!=="class" &&
          prop!=="type" &&
          prop!=="rel") {
          tr = d.data_row({ ❹
            className:"item "+item.class.join(" "),
            text:prop+"&nbsp;",
            value:item[prop]+"&nbsp;"
          });
          d.push(tr,table);
        }
      }
      d.push(table, segment);
      d.push(segment, elm); ❺
    }
  }
}
```

It might be a bit tough to see how the HTML dom is utilized in this example since I use a helper class (the `d` object) to access most of the DOM functions. But you can see

www.it-ebooks.info

that, for each Siren `entity` I create an HTML `div` tag (callout #1). I then create an HTML anchor tag (callout #2) for each item. I set up an HTML `<table>` to hold the Siren `entity`'s properties (callout #3) and add a new table row (`<tr>`) for each one (callout #4). Finally, after completing all the rows in the table, I add the results to the HTML page for visible display (callout #5).

This works because all the implementation examples in this book for common HTML browsers. For cases where the target clients are mobile devices or native desktop applications, I need to work out another strategy. One way do handle this is to create "reverse representors" for each platform. In other words, create a custom `Format-to-Domain` handler for iOS, one for Andriod, and for Windows Mobile, etc. Then the same for Linux, Mac, and Windows desktops, and so forth. This can get tedious, though. That's why using the browser DOM is still appealing and why some mobile apps rely on tools like Apache Cordova, Mono, Appcelerator and other cross-platform development environments.

Client-Side Representors

As of this writing, there are a number of efforts to build Representor libraries that focus on the client — the 'reverse' of the example I outlined in Chapter XX(tk:link). The team at Apiary are working on the Hyperdrive project. The Hypermedia Project is a Microsoft.NET specific effort. And Joshua Kalis has started a project (to which I am a contributor) called Rosetta. Finally, the Yaks project is an independent OSS effort to create a framework that includes the representor pattern to support plug-ins for new formats.

There may be more projects by the time you read this book, too.

Leverage the API Vocabulary

Once you start building clients that CODE TO THE MEDIA TYPE, you'll find that you still need to know domain-specific details that appear in responses. Things like:

- Does this response contain the list of users I asked for?
- How do I find all the inactive customers?
- Which of these invoice records are over-due?
- Is there a way for me to find all the products that are no longer in stock in the warehouse?

All these questions are *domain-specific* and are not tied to any single response format like HAL, Cj, Siren, etc. One of the reasons the HTML browser as been so powerful is that the browser source code doesn't need to know anything about acounting or user management. That's because the *user* driving the browser knows that stuff. The

www.it-ebooks.info

browser is just the *agent* of the human user. That's why we often refer to browsers as 'user agents.' For many API client cases, there is a human user available to interpret and act upon the domain-specific information in API responses. However, there are cases where the API client is not acting as a direct 'user agent.' Instead it is just a middleware compontent or utility app tasked with some job by itself (e.g. find all the overdue invoices, etc.). In these cases, the client app needs to have enough domain information to complete it's job. And that's where API Vocabularies come in.

There are a handful of projects focused on documenting and sharing domain-specific vocabularies over the WWW. One of the best know examples of this is the Schema.org project (pronounced *schema dot org*). Schema.org contains lists of common terms for all sorts of domains. Large Web companies like Google, Microsoft, and Facebook all use Schema.ogr vocabularies to drive parts of their system.

Vocabularies A Plenty

Along with Schema.org, there are other vocabulary efforts such as the IANA Link Relations registry, the microformats group, and the *Dublin Core Metadata Initiatve* or DCMI. I and a few other have also been working on an Internet draft for *Application-Level Profile Semantics* or ALPS for short.

I won't have time to go into vocabularies in this book and encourage you to check out these and other similar efforts in order to learn more about how they can be used in your client-side apps.

So what does this all look like? How can you use vocabularies to enable API clients to act own thier own safely? Basically, you need to "teach" the API consumer to perform tasks based on some baked-in domain knowlege. For example, I might want to create an API consumer that uses on service to finds overdue invoices and pass that information off to another service for further processing. This means the API consumer needs to "know" about invoices and what it means to be "overdue." If the API I am using has published a vocabulary, I can look there for the data and action element idenfiers I need to perform my work.

Here's what that published vocabulary might look like as expressed in a simplified ALPS XML document:

```
<alps>
  <doc>Invoice Management Vocabluary</doc>
  <link rel="invoice-mgmt" href="api.example.org/profile/invoice-mgmt" />

  <!-- data elements -->
  <descriptor id="invoice-href" />
  <descriptor id="invoice-number" />
  <descriptor id="invoice-status">
    <doc>Valid values are: "active", "closed", "overdue"</doc>
```

www.it-ebooks.info

```
  </descriptor>

  <!-- actions -->
  <descriptor id="invoice-list" type="safe" />
  <descriptor id="invoice-detail" type="safe" />
  <descriptor id="invoice-search" type="safe">
    <descriptor href="#invoice-status" />
  </descriptor>
  <descriptor id="write-invoice" type="unsafe">
  <descriptor href="#invoice-href" />
  <descriptor href="#invoice-number" />
  <descriptor href="#invoice-status">
  </dscriptor>
</alps>
```

Now, when I build my client application, I know that I can "teach" that app to understand how to deal with an invoice record (`invoice-number` and `invoice-status`) and know how to search for overude invoices (used `search-invoice` with the `invoice-status` value set to `"overude"`). All I need is the starting address for the service and the ability to recognize and execute the search for overdue invoices. The pseudo-code for that example might look like this:

```
:: DECLARE ::
search-link = "invoice-search" ❶
search-status = "overdue"
write-invoice = "write-invoice"
invoice-mgmt = "api.example.org/profile/invoice-mgmt"

search-href = "http://api.example.org/invoice-mgmt"
search-accept = "application/vnd.siren+json"

write-href = "http://third-party.example.org/write-invoices"
write-accept = "application/vnd.hal+json"

:: EXECUTE ::
response = REQUEST(search-href AS search-accept)
IF(response.vocabulary IS invoivce-mgmt) THEN ❷
  FOR-EACH(link IN response)
    IF(link IS search-link) THEN
      invoices = REQUEST(search-link AS search-accept WITH search-status) ❸
      FOR-EACH(link IN invoices)
        REQUEST(write-href AS write-accept
          FOR write-invoice WITH EACH invoice) ❹
      END-FOR
    END-IF
  END-FOR
END-IF

:: EXIT ::
```

www.it-ebooks.info

Although this is only imaginary psuedo-code, you can see the app has been loaded with domain-specific information (callout #1). Then, after the initial request is made, the response is checked to see if it promises to use the `invoice-mgmt` vocabluary (callout #2). If that check passes, the app searches all the links in the response to find the `search-link` and, if found excecutes a search for all invoices with the status of `over due` (callout #3). Finally, if any invoices are returned in that search, they are sent to a new service using the `write-invoice` action (callout #4).

Something to note here is that the defensive coding is on display (the IF statements) ad the code has only initial URLs memorized — the remaining URLs come from within the responses themselves.

Leveraging Vocabularies for your API means you can focus on the important aspects (the data elements and actions) and *not* worry about plumbing details such as URL matching, memorizing the location of a data element within a document, etc.

React to Link Relations for Workflow

The last client implementation principle I'll cover here is to REACT TO LINK RELATIONS FOR WORKFLOW. This means, when working to solve a multi-step problem, focus on selected link relation values instead of writing client apps that 'memorize' a fixed set of steps. This is important because memorizing a fixed set pf steps is a kind of 'tight-binding' of the client to a fixed sequence of events that may not actually happen at runtime due to transient context issues (e.g. part of the service is down for maintence, the logged in use no longer has rights to one of the steps, etc). Or, over time, new steps might be introduced or the *order* of events might change within the service. These are all reasons to NOT bake multi-step details into your client app.

Instea, since the the service you are using has also followed the API principles of DOCUMENT LINK IDENTIFIERS, PUBLISH VOCABULARIES, and DON'T TAKE THINGS AWAY, you can leverage this information and implement a client that is "trained" to look for the proper identifiers and use vocabulary information to know which data elements need to be passed for each operation. Now, even if the links are moved within a response (or even moved to a *different* response) your client will still be able to accomplish your goal well into the future.

One way to approach this REACT TO LINKS principle is to isolate all the important actions the client will need to take and simply implement them as state-alone operations. Once that is done, you can write a single routine that 1) makes a request, 2) inspects that request for one of the 'known' actions, and when found, executes the recognized action.

Below is an example of a twitter-like quote-bot I created for my 2011 book *Building Hypermedia APIs*.

www.it-ebooks.info

```
/* 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);
  }
}
```

In the above example, this routine constantlly monitors the apps current internal `status` and, it is changes from one state to another, this app knows just what to be looking for within the current response and/or what action to take in order to advance to the next step in the effort to reach the final goal. For this bot, the goal is to post insiprational quotes to the social media feed. This bot also knows that it *might* need to authenticate in order to access the feed or possibly even create a new user account. Notice the use of the javascript `switch…case` structure here. There is no notion of 'execution order' written into the code. Just a set of possible states and related operations to attempt to execute.

www.it-ebooks.info

Writing clients in this way allows you to create middleware components that can accomplish a set goal without forcing that client to memorize a particular order of events. That means even when the order of things changes over time — as long as the changes are made in a backward-compatible way — this client will still be able to complete it's assigned tasks.

So, some valuable principles for implemnting clients that support change-over-time include:

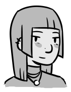
- Code Defensively
- Code to the Media Type
- Leverage the API Vocabulary
- React to Link Relations for Workflow

Summary

This chapter focuses on dealing with the challenge of 'change-over-time' for Web APIs. We looked at examples of planning for and handling change over the decades in three key Web-related fields: TCP/IP and Postel's Law, HTTP and the MUST IGNORE principle, and the backward-compatibility pledge that underpins the design of HTML. We then looked that some general principles we can use when designing APIs, implementing API services, and building client apps that consume those APIs.

The key message of this chapter is that change is inevitable and the way to deal with it is to plan ahead and to adopt the point of view that all changes do not REQUIRE you break the interface. Finally, we learned that successful organizations adopt a *change aesthetic* — a collection of related principles that help guide API design, the inform service implementors, and encourage API consumers to all work toward maintaining backward-compatibility.

Bob and Carol



"Hi, Bob. I decided to stop over to see how you dealt with all the _versioning_ stuff we talked about a last week."

"Hey, Carol. Good to see you. You know, we found a number of examples similar to the ones you mentioned in our last get-together. Postel's Law for TCP/IP, HTTP's MUST IGNORE principle, even a quote from Tim Beners-Lee about how HTML parsers should ignore things they don't understand."

www.it-ebooks.info

"So, that's 25 or more years of dealing with change that never invalidated exsiting implementations, right? That's encouraging. But these are all transport and transfer-level specifications. Our challenge is at the application domain level, right? "

"Yep. And that definitely makes things a bit more interesting -- but not at all impossible. While the actual _implemetation_ deatils for TCP/IP and HTTP might be different, the _principles_ behind them apply to all aspects of API design and implementation."

"Great, so we can just ignore all the *versioning* stuff, then and keep rolling along."

"No, that's not quite right. We identified a number of important principles that we will be publishing to all our teams. They need guidance on how to design and build APIs that can handle change-over-time in a backward-compatible way."

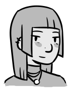
"Backward-compatible way. That's the key, isn't, Bob? With that as our key principle, we can help teams continue to add features and enrich data responses without breaking existing apps."

"Exactly. I've emailed you the list of recommended principles already and we'll incorprate that into the guidance docs we share with everyone."

"Sounds great. By the way, what's the story on those new features you promised me in our last meeting?"

"Oh, didn't you hear? We released those into production yesterday. I guess you never noticed since we didn't have to break any existing clients in the process."

www.it-ebooks.info



"Well played, Bob. Well played."

References

1. Blaise Pascal's Wager has more to do with the nature of uncertainty and probability theory than anything else. A decent place to start reading about his Wager is the Wikipedia entry.
2. Alan Kay's 2011 talk on *Programming and Scaling* contains a commentary on how TCP/IP has been updated and improved over the years without ever having to "stop" the Internet.
3. TCP/IP is documented in two key IETF documents: RFC793 (TCP) and RFC791 (IP).
4. The *Client tolerance of bad servers* note can be viwed in the W3C's HTTP protocol archive pages.
5. The IETF specification document for RFC1945 contains eight separate examples of the MUST IGNORE principle. The HTTP 1.1 specifcation (RFC2616) has more than 30 examples.
6. Dave Orchard's 2003 blog post: "Versioning XML Vocabularies" does a good job of illustrating a number of valubale "Must Ignore" patterns.
7. Tim Berner-Lee's Markup archive from 1992 is a great source for those looking into the earliest days of HTML
8. The "2119 Words" can be found in IETF's RFC2119.
9. The book _Software Architecture in Practice+ was written by Len Bass, Paul Clements, and Rick Kazman.
10. I learned about Github's approach to managing backward compatibility from a Yandex 2013 talk by Jason Rudolph on "API Design at Github". As of this writing, the video and slides are still available online.
11. The Schema.org effort includes the [*http://schema.org*]website, a W3C community site, a github repository and an online discussion group.
12. The book *Building Hypermedia APIs* is a kind of 'companion' book to this one. That book focuses on API design with some server-side implementation details.

www.it-ebooks.info

CHAPTER 5

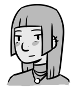
Collection+JSON Clients

"The human animal differs from the lesser primates in his passion for lists."

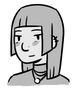
—H. Allen Smith

Bob and Carol

"You know, Carol, I've been reviewing the Collection+JSON hypermedia format and wondering if it would help us on our quest for a more adaptable API client."


"Interesting that you should mention Cj, Bob. My team was just considering it as the next one to try. They tell me it may be promising."

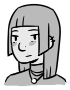
"Right. I notice that Cj looks a lot like Siren and HAL but also has some things in the docs about how clients can use the CRUD pattern similar to the way our plain JSON client did."


"Exactly! And, along with support for CRUD, Cj offers support for query templates similar to the action elements in Siren."

"However, I am a bit skeptical about the way Cj sends data in it's Items collection. That seems a bit odd."

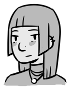
www.it-ebooks.info

“Well, that’s what most of our discussion was about yesterday. Cj requires additional metadata for each domain object and that means it will be hard to simply ‘deserialize’ our domain objects into the response as we do in Siren and HAL.”



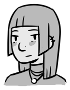
“Huh, so that’s more work for the server-side representor and more payload on the client, right? I wonder if we need all that information.”

“Well, as far as we can tell, Cj payload size is not any larger than our typical HTML payloads and we send HTML all the time.”



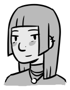
“Hmmm. Interesting. I wonder if that metadata provides something important we can’t see yet?”

“Well, some on my client-side team think that additional metadata will improve adaptability over time.”



“Really? Well, let’s try it out. My team can build up a Cj representor pretty quickly on the server side. And your group should be able to create a Cj client without too much trouble, right Carol?”

“That’s right, Bob. Let’s see what our teams come up with and meet back here in another week.”



“Ok, Carol. Let’s do it!”

The last hypermedia format we’ll review in this book is the Collection+JSON format. It has similarities with both HAL (Chapter XX TK) and Siren (Chapter XX TK) but also has a rather unique approach. The Cj format is designed, from the very beginning, as a *list-style* format — it is meant to return lists of records. As we’ll see when we review the design in the next section, there is more to the format than just the list, but lists is are what Cj is all about.

www.it-ebooks.info

Cj also takes a cue from the classic Create-Read-Update-Delete (CRUD) pattern used by most Web API clients today. We covered this style in Chapter XX (TK) and some of the lessons from the client we built in that chapter will apply for the Cj client, too.

In this chapter we'll take a quick look at format design, the Cj Representor code and the Cj general client. Then, as we have for other client implementations, we'll introduce changes to see how well the API client holds up as the backend API makes backward-compatible changes.

Finally, we'll be sure to check in on our progress along the path to meeting the OAA Challenge. While HAL excels at handling ADDRESSES and Siren has great support for ACTIONS, Cj was designed to meet the OBJECTS challenge — the ability to share metadata about the domain objects at runtime. And, as we'll see in the review, Cj meets the OBJECT challenge with a novel solution — by making them inconsequential to the client-server experience.

The Collection+JSON Format

I designed and published the Collection+JSON format in 2011 — the same year Mike Kelly released his HAL (tk chapter) specification. Collection+JSON (aka Cj) was designed to make it easy to manage lists of data like blog posts, customers, products, users, etc. The description that appears on the Cj specification page says:

> "[Cj] is similar to the The Atom Syndication Format (RFC4287) and the The Atom Publishing Protocol (RFC5023) . However, Collection+JSON defines both the format and the protocol semantics in a single media type. [Cj] also includes support for Query Templates and expanded write support through the use of a Write Template."
>
> —Collection+JSON specs

Essentially, Cj is Atom in JSON with FORMs. The good news is that Cj follows' Atom's support for the Create-Read-Update-Delete (CRUD) pattern. That means most developers can understand Cj's read/wrire semantics rather easily. the added bonus for Cj is that it has elements for describing HTML-like FORMs for filtering data (with Cj's *queries* element) and for updating content on the server (via the *template*) element. However, as we'll see in the review that follows, the way the *template* element is used can be a bit of a challenge.

www.it-ebooks.info

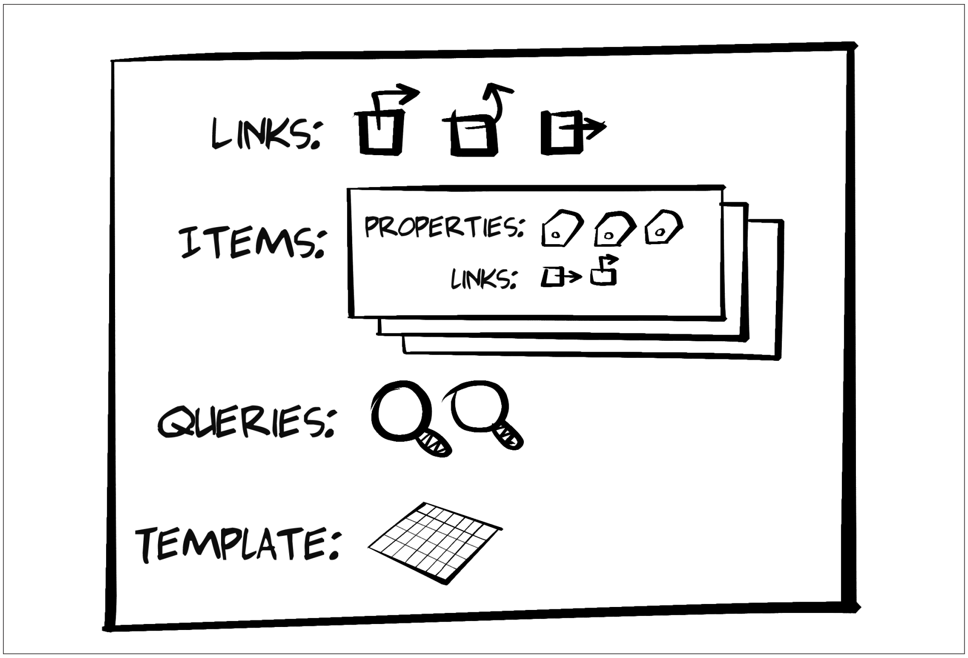


Figure 5-1. The Collection+JSON Document Model

You can get a better understanding of the Collection+JSON media type by checking out the online Cj documentation. There is also a Cj discussion list and github organization where additional information is shared. See the References section (TK) of this chapter for details.

The basic elements of every Cj message are:

- Links : A set of one or more `link` elements. These are very similar to HAL and Siren `link` elements
- Items : One or more data items — basically the APIs domain objects. The `properties` of HAL and Siren are very similar to Cj `items`.
- Queries : These are basically HTML "GET" FORMs. Cj `queries` are like HAL's templated links and Siren's `action` elements (with the `method` set to "GET")
- Template : In Cj all *write* operations (HTTP POST & PUT) are done using the `template` element. It contains one or more `data` objects — each one like HTML `input` elements. Again, this is like Siren `action` elements. HAL doesn't have anything that match the Cj `template`

www.it-ebooks.info

Cj also has an `error` element for returning error information and a `content` element for returning free-form text and markup. We'll not cover these here today. You can read up on them in the Cj documentation mentioned in the refernce section. (TK).

Here's an example of a simple Collection+JSON message that shows the major sections of a Cj document including `links` (#1), `items` (#2), `queries` (#3), and the `template` (#4) element.

```
{
  "collection": {
    "version": "1.0",
    "href": "http://orm-hyper-todo.herokuapp.com", ❺
    "title": "ORM Hyper-Tasks",
    "links": [ ❶
      {
        "href": "http://orm-hyper-todo.herokuapp.com/",
        "rel": "collection",
        "prompt": "All task"
      }
    ],
    "items": [ ❷
      {
        "rel": "item",
        "href": "http://orm-hyper-todo.herokuapp.com/1sv697h2yij",
        "data": [
          {"name": "id", "value": "1sv697h2yij", "prompt": "id"},
          {"name": "title", "value": "Marina", "prompt": "title"},
          {"name": "completed", "value": "false", "prompt": "completed"}
        ]
      },
      {
        "rel": "item",
        "href": "http://orm-hyper-todo.herokuapp.com/25ogsjhqtk7",
        "data": [
          {"name": "id", "value": "25ogsjhqtk7", "prompt": "id"},
          {"name": "title", "value": "new stuff", "prompt": "title"},
          {"name": "completed", "value": "true", "prompt": "completed"}
        ]
      }
    ],
    "queries": [ ❸
      {
        "rel": "search",
        "href": "http://orm-hyper-todo.herokuapp.com/",
        "prompt": "Search tasks",
        "data": [
          {"name": "title", "value": "", "prompt": "Title"}
        ]
      }
    ],
    "template": { ❹
```

www.it-ebooks.info

```
        "prompt": "Add task",
        "rel": "create-form",
        "data": [
          {"name": "title", "value": "", "prompt": "Title"},
          {"name": "completed", "value": "false", "prompt": "Complete"}
        ]
      }
    }
  }
```

Another important attribute of a Cj document is the root-level `href` (see callout #5). The value of `href` is used when adding a new record to the `items` collection. We'll talk more about this property when we cover the `template` element in below (TK ref).

Links

The `links` element in a Cj document is always a valid JSON array that contains one or more `link` objects. Important `link` element properties include `href`, `rel`, and `prompt` properties. These work similar to the way HTML `<a>…</a>` tags — static URLs for HTTP GET actions.

Representing Links in Collection+JSON

```
  "links": [
    {
      "href": "http://rwcbook12.herokuapp.com/home/",
      "rel": "home collection",
      "prompt": "Home"
    },
    {
      "href": "http://rwcbook12.herokuapp.com/task/",
      "rel": "self task collection",
      "prompt": "Tasks"
    },
    {
      "href": "http://rwcbook12.herokuapp.com/user/",
      "rel": "user collection",
      "prompt": "Users"
    }
  ]
```

In Cj, the `links` section typically holds links that are relevant for the current document or, in a human-centric UI, the current screen or Web page. Along with important navigation links for the app (see in the above example), the `links` section may include things like page-level navigation (`first`, `previous`, `next`, `last`) or other similar links.

Another handy property on Cj `link` objects is the `render` property. This tells consuming apps how to treat the link. For example, if the `render` value is set to `"none"`,

www.it-ebooks.info

client apps are not expected to display the link. This is handy when passing `link` elements for things like CSS stylsheets, profile URLs, or other types of information.

Representing Links in Collection+JSON

```
"links": [
  {
    "href": "http://api.example.org/profiles/task-management",
    "rel": "profile",
    "render" : "none"
  }
]
```

Items

Probably the most unique element in Cj documents is the `item` section. The `items` section is similar to HAL's root-level `properties` and Siren's `properties` object, Cj `items` contain the domain objects in the response like "users", "customers", "products", and so forth. However, unlike the way HAL and Siren express domain objects, Cj has a highly structured approach. HAL and Siren express their domain objects as either simple name-value pairs or, in the case of Siren, as `subentities`. And both HAL and Siren support sending nested JSON objects as properties. But Cj doesn't work like that and this can be a source of both frustration and freedom.

Here is an example of a "user" object expressed as a Cj `item`:

```
{
  "rel": "item http://api.example.org/rels/user",
  "href": "http://api.example.org/user/alice", ❶
  "data": [ ❷
    {"name": "id", "value": "alice", "prompt": "ID", "render":"none"},
    {"name": "nick", "value": "alice", "prompt": "Nickname"},
    {"name": "email", "value": "alice-ted@example.org", "prompt": "Email"},
    {"name": "name", "value": "Alice Teddington, Jr.", "prompt": "Full Name"}
  ],
  "links": [ ❸
    {
      "prompt": "Change Password",
      "rel": "edit-form http://api.example.org/rels/changePW",
      "href": "http://api.example.org/user/pass/alice"
    },
    {
      "prompt": "Assigned Tasks",
      "rel": "collection http://api.example.org/rels/filterByUser",
      "href": "http://api.example.org/task/?assignedUser=alice"
    }
  ]
}
```

www.it-ebooks.info

As you see in the above example, a Cj `item` contains the `rel` and `href` (#1), a list of `data` elements (#2) and may also contain one or more `link` elements for read-only actions associated with the `item`. The way Cj expresses the item properties (`id`, `nick`, `email` and `name`) is unique among the formats covered in this book. Cj documents return not just the property identifier and value (e.g. `"id":"alice") but also a suggested +prompt` property. Cj also supports other attributes including `render` to help clients decide whether to display the property on screen. This highly structured format makes it possible to send both the domain data *and* metadata about each property and object. As we'll see when we start working on the Cj client app, this added data comes in handy wen creating a human-centri interface.

The `links` collection within each Cj `item` contains one or more static safe links (like those in the root-level `links` collection. This space can be used to pass item-level links within a Cj response. For example, in the snippet above, you can see a link that points to a form for updating the user password and a link that points to a filtered list of tasks related to this user object. The item-level `links` section is optional and any link that appears in the collection MUST be treated as a safe link (e.g. dereferenced using HTTP GET).

Queries

The `queries` element in Collection+JSON is meant to hold safe requests (e.g. HTTP GET) that have one or more parameters. These are similar to HTML FORMS with the `method` attribute set to "GET". The `queries` section in a Cj document is an array with one or more query objects. They look similar to Cj link objects but can have an associated `data` array, too.

Here's an example:

```
{
  "rel": "search",
  "name" : "usersByEmai",
  "href": "http://api.example.org/user/",
  "prompt": "Search By Email",
  "data": [
    {
      "name": "email",
      "value": "",
      "prompt": "Email",
      "required": "true"
    }
  ]
}
```

As you can see from the above example, a Cj query object has `rel`, `name`, `href`, and `prompt` attributes. Then there is one or more `data` elements. The `data` elements are similar to HTML `input` elements. Along with the `name`, `value`, and `prompt` attributes,

www.it-ebooks.info

data elements can have required and (not shown above) readOnly and pattern attributes. These last attributes help service send clients additional metadata about the arguments for a query.

Note that Cj query objects do not have an attribute to indicate which HTTP method to use when executing the query. that is because Cj queries always use the HTTP GET method.

There is another Cj element that is similar to HTTP FORM: the template element.

Template

Cj's template element looks similar to the Cj queries element — but even *smaller*. It just has a set of one or more data objects. These represent the input arguments for a write action (e.g. HTTP POST or PUT). Here's what a Cj template looks like:

```
"template": {
  "prompt": "Add Task",
  "data": [
    {"name": "title", "value": "", "prompt": "Title", "required": "true"},
    {"name": "tags", "value": "", "prompt": "Tags"},
    {"name": "completeFlag", "value": "false", "prompt": "Complete",
      "patttern": "true|false"}
  ]
}
```

The template element can have an optional prompt, but the most important part of the template is the data array that decribes the possible input arguments for the write operation. Like the data elements that appear in Cj queries and items, the template's data items include name and value properties along with a prompt property. And, like the queries version of data elements, they can have additional metadata attributes including readOnly, required and pattern. The pattern element works the same way as the HTML pattern attribute.

There are two important aspects of write operations that are missing from the Cj template: 1) the target URL, and 2) the HTTP method. That's because, in Cj, the template applies to two different parts of the CRUD model: "create" and "update". Just how the request is executed depends on what the client app wants to do.

Using Cj Templates to Create New Resouces

When used to create a new member of the collection, the client app fills out the template and then uses the HTTP POST for the method the the value of the Cj document's href as the target URL.

www.it-ebooks.info

For example, using the CJ document represented at the start of this chapter (TK ref), a client application can collect inputs from a user and send a POST request to add a new TASK record. The HTTP request would look like this:

```
*** REQUEST ***
POST / HTTP/1.1 ❶
Host: http://orm-hyper-todo.herokuapp.com
Content-Type: application/vnd.collection+json
...

"template": {
  "data": [
    {"name": "title", "value": "adding a new record"},
    {"name": "tags", "value": "testing adding"},
    {"name": "completeFlag", "value": "false"}
  ]
}
```

The Cj specification says that clients can send the `template` block (as seen above OR just send an array of `data` objects and servers SHOULD accept both. Also, servers SHOULD accept payloads with `data` objects that include `prompts` and other properties and just ignore them.

As you can see in the example above, the URL from the Cj *document* `href` along with the HTTP POST method is used to add a new resource to the Cj collection.

Using Cj Templates to Update and Existing Resource

When client apps want to update an existing resource they use the HTTP PUT method and the `href` property of the `item` to update. Typically, client apps will automatically fill in the `template.data` array with the values of the existing item, allow users to modify that data and then execute the PUT request to send the update information to the server.

```
*** REQUEST ***
PUT /1sv697h2yij HTTP/1.1 ❶
Host: http://orm-hyper-todo.herokuapp.com
Content-Type: application/vnd.collection+json
...
"template": {
  "data": [
    {"name": "id", "value": "1sv697h2yij"},
    {"name": "title", "value": "Marina Del Ray"},
    {"name": "completed", "value": "true"}
  ]
}
```

www.it-ebooks.info

Note that (at callout #1) the URL from the item's `href` property is used along with the HTTP PUT method. This is how Cj clients use the `template` to update an existing `item`.

So, one template, two ways to use it. That's how Cj describes write operations.

Error

The Collection+JSON design also includes an `error` element. This is used to pass domain-specific error information from server to client. For example, if a resource cannot be found or an attempt to update an existing record failed, the server can use the `error` element to return more than HTTP 404 or 400. It can return a text description of the problem and even include advice on how to fix it.

For example, if a someone attempted to assign a TPS TASK to a non-existant user, the server might respond like this:

```
{
  "collection": {
    "version": "1.0",
    "href": "//rwcbook12.herokuapp.com/error/",
    "title": "TPS - Task Processing System",
    "error": {
      "code": 400,
      "title": "Error",
      "message": "Assigned user not found (filbert). Please try again.",
      "url": "http://rwcbook12.herokuapp.com/task/assign/1l9fz7bhaho"
    }
  }
}
```

A mentioned earlier, there are some additional elements and properties of Cj documents that I won't cover here. You can check out the full specification at the online site listed in the Reference section at the end of this chapter (TK).

A Quick Summary

By now, we can see that the three featured hypermedia types (HAL, Siren, and Cj) have several things in common. Like HAL and Siren, Cj has an element (`links`) for communicating Links or ADDRESSES. And, like Siren, Cj's `queries` and `template` elements communicate ACTION metadata in responses. And all three have a way to communicate domain-specific objects (HAL's root-level properties, Siren's `properties` object and Cj's `items` collection). Cj's `items` collection is unique since it includes metadata about each property in the domain object (e.g. `prompt` & `render`). This elevates Cj's ability to handle the OBJECT aspect of the OAA Challenge. We'll talk about this again when we build the Cj client app.

www.it-ebooks.info

For now, we have enough background to review the Cj Representor and then walk through our Cj Client SPA code.

The Collection+JSON Representor

As with other formats, the process of coding a Cj Representor is a matter of converting our internal resource representation (in the form of a WeSTL object) into a valid Collection+JSON document. And, like the other representors, it takes only about 300 lines of NodeJS to build up a fully-functional module to produce valid Cj responses.

The source code for the Cj Representor can be found in the associated github repo here: *https://github.com/RWCBook/cj-client*. A running version of the app described in this chapter can be found here: *http://rwcbook12.herokuapp.com/* (TK: check URLs)

Below is a quick walk-through of the Cj Representor code with highlights.

The Top-Level Processing Loop

The top-level processing loop for my Cj Representor is very simple. It starts by initializing an empty `collection` object (to represent a Cj document in JSON) and then populates this object with each of the major Cj elements:

- Links
- Items
- Queries
- Template
- Error (if needed)

Here's what the function looks like:

```
function cj(object, root) {
  var rtn;

  rtn = {};
  rtn.collection = {}; ❶
  rtn.collection.version = "1.0";

  for(var o in object) {
    rtn.collection.href = root+"/"+o+"/"; ❷
    rtn.collection.title = getTitle(object[o]); ❸
    rtn.collection.links = getLinks(object[o].actions);
    rtn.collection.items = getItems(object[o],root);
    rtn.collection.queries = getQueries(object[o].actions);
```

www.it-ebooks.info

```
        rtn.collection.template = getTemplate(object[o].actions);

        // handle any error
        if(object.error) { ❹
          rtn.collection.error = getError(object.error);
        }
      }
      // send results to caller
      return JSON.stringify(rtn, null, 2); ❺
    }
```

The code above has just a few interesting items. After intializing a `collection` document (#1) and establishing the document-level `href` (#2), the code walks through the passed-in WeSTL object tree (#3) and constructs the Cj `title`, `links`, `items`, `queries`, and `template` elements. Then, if the current object is an error, the Cj `error` element is populated (#4). Finally, the completed Cj document is returned (#5) to the caller.

Now, let's take a look at each of the major routines used to build up the Cj document.

Links

The `links` element in Cj holds all "top-level" links for the document. The Cj Representor code scans the incoming WesTL object for any `action` element that qualifies and, if needed, resolves any URI templates before adding the link to the collection.

Here's the code:

```
    // get top-level links
    function getLinks(obj, root, tvars) {
      var link, rtn, i, x, tpl, url;

      rtn = [];
      if(Array.isArray(obj)!==false) {
        for(i=0,x=obj.length;i<x;i++) { ❶
          link = obj[i];
          if(link.type==="safe" &&
            link.target.indexOf("app")!==-1 &&
            link.target.indexOf("cj")!==-1) ❷
          {
            if(!link.inputs) {
              tpl = urit.parse(link.href);
              url = tpl.expand(tvars); ❸
              rtn.push({ ❹
                href: url,
                rel: link.rel.join(" ")||"",
                prompt: link.prompt||""
              });
            }
          }
```

www.it-ebooks.info

```
      }
    }
    return rtn; ❺
  }
```

Here are the high points in the `getLinks` function:

1. If we have action objects, loop through them
2. First, check to see if the current link meets the criteria for *top-level* links in a Cj document
3. If it does, use the passed-in `tvars` collection (template variables) to resolve any URI Template
4. Then add the results to the link collection
5. Finally, return the populated collection to the caller.

Items

The next interesting function is the one that handles `items`. This is the most involved routinr in the Cj Representor. That's because Cj does quite a bit to supply both data and metadata about each domain object is passes to the client app.

Here's the code.

```
// get list of items
function getItems(obj, root) {
  var coll, temp, item, data, links, rtn, i, x, j, y;

  rtn = [];
  coll = obj.data;
  if(coll && Array.isArray(coll)!==false) {
    for(i=0,x=coll.length;i<x;i++) {
      temp = coll[i];

      // create item & link
      item = {}; ❶
      link = getItemLink(obj.actions);
      if(link) {
        item.rel = (Array.isArray(link.rel)?link.rel.join(" "):link.rel);
        item.href = link.href;
        if(link.readOnly===true) {
          item.readOnly="true";
        }
      }

      // add item properties
      tvars = {}
      data = [];
```

www.it-ebooks.info

```
        for(var d in temp) { ❷
          data.push(
            {
              name : d,
              value : temp[d],
              prompt : (g.profile[d].prompt||d),
              render:(g.profile[d].display.toString()||"true")
            }
          );
          tvars[d] = temp[d];
        }
        item.data = data;

        // resolve URL template ❸
        tpl = urit.parse(link.href);
        url = tpl.expand(tvars);
        item.href = url;

        // add any item-level links ❹
        links = getItemLinks(obj.actions, tvars);
        if(Array.isArray(links) && links.length!==0) {
          item.links = links;
        }

        rtn.push(item); ❺
      }
    }
    return rtn; ❻
  }
```

The `getItems` routine is the largest in the Cj Representor. It actually handles three key things, the URL for the item, the item's data properties, and any links associated with the item. Here's the breakdown:

1. For each data item in the list, first set the `href` property
2. Then loop through the properties of the domain object and construct Cj `data` elements.
3. After collecting the data values, use that collection to resolve any URL template in the item's `href`
4. Next, go collect up (and resolve) any Cj `link` objects for this single `item`
5. Once all that is done, add the results to the internal item collection and then
6. Finally, return the completed collection to the calling routine.

The resulting `item` collection looks like this:

```
  "items": [
    {
      "rel": "item",
```

www.it-ebooks.info

```
    "href": "http://rwcbook12.herokuapp.com/task/1l9fz7bhaho",
    "data": [
      {"name":"id","value":"1l9fz7bhaho","prompt":"ID","render":"true"},
      {"name":"title","value":"extensions","prompt":"Title","render":"true"},
      {"name":"tags","value":"forms testing","prompt":"Tags","render":"true"},
      {"name":"completeFlag","value":"true","prompt":"Complete Flag",
        "render":"true"},
      {"name":"assignedUser","value":"carol","prompt":"Asigned User",
        "render":"true"},
      {"name":"dateCreated","value":"2016-02-01T01:08:15.205Z",
        "prompt":"Created","render":"false"}
    ],
    "links": [
      {
        "prompt": "Assign User",
        "rel": "assignUser edit-form",
        "href": "http://rwcbook12.herokuapp.com/task/assign/1l9fz7bhaho"
      },
      {
        "prompt": "Mark Active",
        "rel": "markActive edit-form",
        "href": "http://rwcbook12.herokuapp.com/task/active/1l9fz7bhaho"
      }
    ]
  }
  ... more items here ...
]
```

Queries

The `getQueries` routine is the one that generates the "safe" parameterized queries—basically HTML "GET" FORMS. That means, along with a URL, there is a list of one or more argument descriptions. These would be the HTML `input` elements of a FORM. The code is generating Cj `queries` is very straightfoward and looks like this:

```
// get query templates
function getQueries(obj) {
  var data, d, query, q, rtn, i, x, j, y;

  rtn = [];
  if(Array.isArray(obj)!==false) {
    for(i=0,x=obj.length;i<x;i++) { ❶
      query = obj[i];
      if(query.type==="safe" &&  ❷
        query.target.indexOf("list")!==-1 &&
        query.target.indexOf("cj") !==-1)
      {
        q = {}; ❸
        q.rel = query.rel.join(" ");
        q.href = query.href||"#";
        q.prompt = query.prompt||"";
```

www.it-ebooks.info

```
        data = [];
        for(j=0,y=query.inputs.length;j<y;j++) { ❹
          d = query.inputs[j];
          data.push(
            {
              name:d.name||"input"+j,
              value:d.value||"",
              prompt:d.prompt||d.name,
              required:d.required||false,
              readOnly:d.readOnly||false,
              patttern:d.pattern||""
            }
          );
        }
        q.data = data;
        rtn.push(q); ❺
      }
    }
  }
  return rtn; ❻
}
```

The walk-through is rather simple:

1. Loop through all the transitions in the WeSTL document
2. Find the transitions that are valid for the cj `queries` collection
3. Start an empty query object and set the `href` and +rel= properties.
4. Loop through the WeSTL `input` elements to create Cj `data` elements for the query
5. Add the completed query to the collection
6. Finally, return that collection to the calling routine.

Again, there is no HTTP method supplied for each query since the spec says all Cj `queries` should be executed using HTTP "GET".

That covers the 'read' FORMS in Cj. Next is the work to handle the 'write' forms — Cj `template`.

Template

In Cj, 'write' FORMS are represented in the `template` element. The `getTemplate` routine in our Cj Representor handles generating the `template` element and the code looks like this:

```
// get the add template
function getTemplate(obj) {
  var data, temp, field, rtn, tpl, url, d, i, x, j, y;
```

www.it-ebooks.info

```
    rtn = {};
    data = [];
    if(Array.isArray(obj)!==false) {
      for(i=0,x=obj.length;i<x;i++) {
        if(obj[i].target.indexOf("cj-template")!==-1) { ❶
          temp = obj[i];

          // emit data elements
          data = [];
          for(j=0,y=temp.inputs.length;j<y;j++) { ❷
            d = temp.inputs[j];
            field = { ❸
              name:d.name||"input"+j,
              value:(d.value||"",
              prompt:d.prompt||d.name,
              required:d.required||false,
              readOnly:d.readOnly||false,
              patttern:d.pattern||""
            };
            data.push(field); ❹
          }
        }
      }
    }
    rtn.data = data;
    return rtn; ❺
  }
```

There is not much to the `getTemplate` routine, so the highlights are a bit boring:

1. Loop through the WeSTL transitions and find the one valid for Cj `template`
2. Then loop through the transition's `input` collection
3. Use that information to build a Cj `data` element
4. And add that to the collection of `data` elements for this `template`
5. Finally, after adding the completed `data` collection to the `template` object, return the results to the caller.

As a reminder, there is no `href` property or HTTP method for Cj `templates`. The URL and method to use are determined by the client at runtime based on whether the client is attempting a Create or Update action.

That leaves just one small object to review: the Cj `error` element.

www.it-ebooks.info

Error

Unlike HAL and Siren, Cj has a dedicated `error` element for responses. This makes is easy for clients to recognize and render any domai-specific error information in server responses. There are only four defined fields for the Cj `error` object: `title`, `message`, `code`, and `url`. The `getError` function is small and looks like this:

```
// get any error info
function getError(obj) {
  var rtn = {};

  rtn.title = "Error";
  rtn.message = (obj.message||"");
  rtn.code = (obj.code||"");
  rtn.url = (obj.url||"");

  return rtn;
}
```

There is really nothing to talk about here since the routine is so simple. It is worth pointing out that Cj responses can include both error information *and* content in the `links`, `items`, `queries`, and `template` elements. That makes is possible to return a fully-populated Cj document along with some error data to help the user resolve any problems.

The `error` object uses the `url` property to pass the URL related to the error. In all other Cj elements, the URL is passed in the `href` property. This is an inconsistency in the design that I may need to "fix" someday. But, in keeping with supporting only backward-compatible changes, that means the future of the `error` element will likely include both the `url` and the `href` property.

With the Cj Representor walk-through completed, it's time to review the Cj Client SPA.

The Collection+JSON SPA Client

OK, now we can review the Collection+JSON Single-Page App (SPA). This Cj client supports all the major features of Cj including `links`, `items`, `queries`, and `template`. It also supports other Cj element including `title`, `content`, and `error` elements.

The source code for the Cj Representor can be found in the associated github repo here: *https://github.com/RWCBook/cj-client*. A running version of the app described in this chapter can be found here: *http://rwcbook12.herokuapp.com/files/cj.client.html* (TK: check URLs)

www.it-ebooks.info

As we did with the JSON, HAL, and Siren SPAs, we'll start with a review of the HTML container and them move on to review the top-level parsing routine along with the major functions that parse the key Cj document sections to build up the rest of the general Cj client.

The HTML Container

All the SPA apps in this book start with an HTML container and this one is no different. Below is the static HTML that is used to host the Cj documents sent by the server.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cj</title>
    <link href="./semantic.min.css" rel="stylesheet" />
  </head>
  <body>
    <div id="links"></div> ❶
    <div style="margin: 5em 1em">
      <h1 id="title" class="ui page header"></h1> ❷
      <div id="content" style="margin-bottom: 1em"></div> ❸
      <div class="ui mobile reversed two column stackable grid">
        <div class="column">
          <div id="items" class="ui segments"></div> ❹
        </div>
        <div class="column">
          <div id="edit" class="ui green segment"></div>
          <div id="template" class="ui green segment"></div> ❺
          <div id="error"></div> ❻

          <div id="queries-wrapper">
            <h1 class="ui dividing header">
              Queries
            </h1>
            <div id="queries"></div> ❼
          </div>
        </div>
      </div>

      <div>
        <pre id="dump"></pre>
      </div>

    </div>
  </body>
  <script src="dom-help.js">//na </script>
  <script src="cj-client.js">//na </script> ❽
  <script>
    window.onload = function() {
```

www.it-ebooks.info

```
        var pg = cj();
        pg.init("/", "TPS - Task Processing System"); ❾
      }
    </script>
  </html>
```

A lot of the HTML shown above is there to support the layout needs of the CSS library. But you can still find all the major Cj document elements represented by `<div>` tags in the page. They are:

1. The `links` collection
2. The `title` element
3. The `content` element
4. The `items` element
5. The `template` element
6. The `error` element
7. The `queries` element

The Cj parsing script is loaded at callout #8 and, after everything loads, the initial request starts at callout #9. that line calls into the top-level parse loop for the Cj library.

The Top-Level Parse Loop

In the Cj client, the top-level parse loop gets called each time a user makes a selection in the UI, this follows the Request-Parse-Wait (RPW) pattern I covered in Chapter XX (TK). It turns out the parse loop for Cj is a bit simpler than the ones for the JSON, HAL, and Siren clients.

```
// init library and start
function init(url) {
  if(!url || url==='') {
    alert('*** ERROR:\n\nMUST pass starting URL to the Cj library');
  }
  else {
    g.url = url;
    req(g.url,"get"); ❶
  }
}

// primary loop
function parseCj() { ❷
  dump();
  title();
  content();
  links();
```

www.it-ebooks.info

```
    items();
    queries();
    template();
    error();
    cjClearEdit();
  }
```

The code set above looks pretty similar by now. After making the intiial request (callout #1), the `parseCj` routine is called and it walks through all the major elements of a Collection+JSON document. The only other interesting elements in this code snippet are the internal routines. First, the call to the `dump()` method at the start of the loop — this is just for debugging help on screen — and second, the `cjClearEdit()` call at the end of the routine to handle cleaning up the HTML `div` used to display the UI's current editing form.

I'll skip talking about the `title` and `content` routines here — you can check them out yourself in the soure code. Below is a walk-through of the other major routines to handle Cj responses.

Links

The routine that handles parsing and rendering Cj `links` is pretty simple. However, it has a bit of a twist. The code checks the domain-specific metadata about the link. For example, some links are not rendered on the screen (e.g. HTML stylesheets, IANA profile identifiers, etc.). Some other links should actually be rendered as embedded images instead of navigationel links. The Cj design allows servers to indicate this level of link metadata in the message itself — something the HAL and Siren clients do not support in thier design.

Here's the code for the `links()` function.

```
// handle link collection
function links() {
  var elm, coll, menu, item, a, img, head, lnk;

  elm = d.find("links");
  d.clear(elm);
  if(g.cj.collection.links) { ❶
    coll = g.cj.collection.links;
    menu = d.node("div");
    menu.className = "ui blue fixed top menu";
    menu.onclick = httpGet;

    for(var link of coll) { ❷
      // stuff render=none Cj link elements in HTML.HEAD ❸
      if(isHiddenLink(link)===true) {
        head = d.tags("head")[0];
        lnk = d.link({rel:link.rel,href:link.href,title:link.prompt});
        d.push(lnk,head);
```

www.it-ebooks.info

```
          continue;
        }
        // render embedded images, if asked ❹
        if(isImage(link)===true) {
          item = d.node("div");
          item.className = "item";
          img = d.image({href:link.href,className:link.rel});
          d.push(img, item);
          d.push(item, menu);
        }
        else {
          a = d.anchor({rel:link.rel,href:link.href,text:link.prompt, ❺
            className: "item"});
          d.push(a, menu);
        }
      }
      d.push(menu, elm); ❻
    }
  }
```

Even though there are quite a few lines of code here, it's all straight-forward. The highlights are:

1. After making sure the are Cj `links` to process, set up some layout to hold them.
2. Now start looping through the `links` collection
3. If the `link` elmement should now be rendered, place it in the HTML `<head>` section of the page
4. If the `link` element should be rendered as an image, process it properly
5. Otherwise, treat is as a simple `<a>` tag and add it to the layout.
6. Finally, push the results to the viewable screen.

And here is an example of rendering the Cj `links` at runtime:

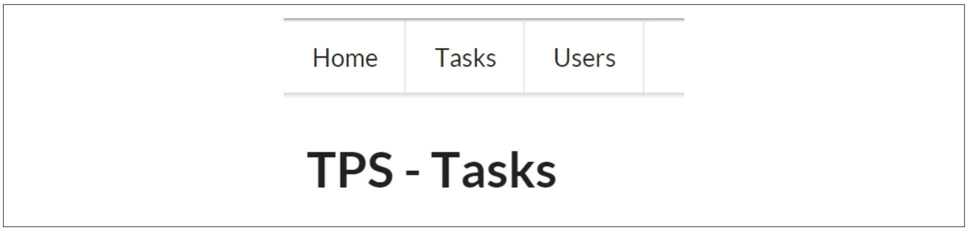


Figure 5-2. Rendering Cj `links` at runtime

It turns out the cases where the `link` element is not displayed (callout #3) or the `link` is an image (callout #4) takes more code than cases where the `link` element is just a

www.it-ebooks.info

navigational element (callout #5). We'll see some more of the kind of code when we parse the `items` collection.

Items

The `items()` function is the most involved routine in the Cj library. At 125 lines, it is also the longest. Taht'a because (as we saw when reviewing the `items` handling in the Cj Representor, the `items` element is the most involved of all the Cj document design. I won't include all the lines of this routine but will show the key processing in the routine. You can find the full set of code in the source code repo associated with this chapter.

I'll break up the code review for the `items()` routine into three parts:

- Rendering Cj `item` editing links
- Rendering Cj `item` `links`
- Rendering Cj `item` `data` properties

First, the code that handles each `item`'s Read-Update-Delete links — the last three elements of the CRUD pattern. Each Cj `item` has an `href` property and, optionally, a `readOnly` property. Using this informatoin as a guide, Cj clients are responsible for rendering support for the Read, Update, and Delete links. You can see this in the code below. At callout #1, the Read link is created. The Update link is created at #2 and the Delete link is created at #3. Note the checking of both the `readOnly` status of the client as well as whether the `template` can be found in the Cj document. These values are used to decide which links (Update and Delete) are rendered for the `item`.

```
// item link
a1 = d.anchor( ❶
  {
    href:item.href,
    rel:item.rel,
    className:"item link ui basic blue button",
    text:item.rel
  }
);
a1.onclick = httpGet;
d.push(a1,buttons);

// edit link
if(isReadOnly(item)===false && hasTemplate(g.cj.collection)===true) {
  a2 = d.anchor( ❷
    {
      href:item.href,
      rel:"edit",
      className:"item action ui positive button",
```

www.it-ebooks.info

```
          text:"Edit"
        }
      );
      a2.onclick = cjEdit;
      d.push(a2, buttons);
    }

    // delete link
    if(isReadOnly(item)===false) {
      a3 = d.anchor( ❸
        {
          href:item.href,
          className:"item action ui negative button",
          rel:"delete",
          text:"Delete"
        }
      );
      a3.onclick = httpDelete;
      d.push(a3,buttons);
    }
```

The next important snippet in the `items()` routine is the one that handles any item-level `links`. So, in the code you can see (at callout #1) if there are `links` for this `item`, each link is checked to see if it should be rendered as an image (callout #2) and, if not, it can be rendered as a navigational link (callout #3). Finally, after the the links are processed, the results are added to the item display (callout #4).

```
    if(item.links) { ❶
      for(var link of item.links) {
        // render as images, if asked
        if(isImage(link)===true) { ❷
          p = d.node("p");
          p.className = "ui basic button";
          img = d.image(
            {
              className:"image "+link.rel,
              rel:link.rel,
              href:link.href
            }
          );
          d.push(img, p);
          d.push(p,secondary_buttons);
        }
        else {
          a = d.anchor( ❸
            {
              className:"ui basic blue button",
              href:link.href,
              rel:link.rel,
              text:link.prompt
            }
```

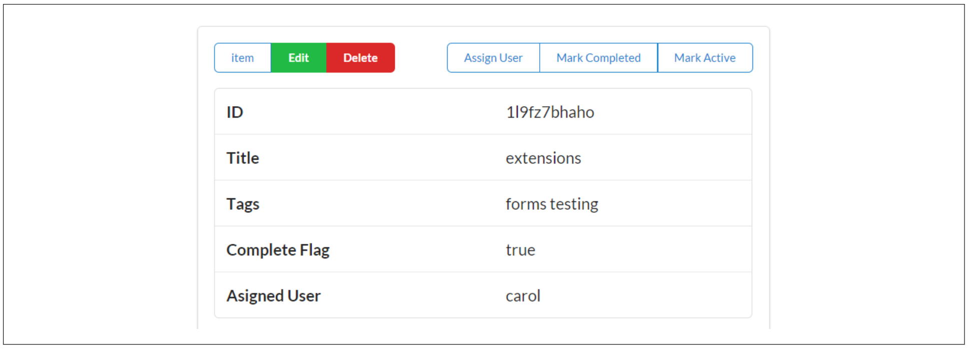
www.it-ebooks.info

```
          );
          a.onclick = httpGet;
          d.push(a,secondary_buttons);
        }
      }
      d.push(secondary_buttons,segment); ❹
    }
```

The last snippet to review in the `items()` routine is the one that handles all the actual `data` properties of the `item`. In this client, they are rendered one-by-one as part of a UI table display. The code (see below) is not very complicated.

```
    for(var data of item.data) {
      if(data.display==="true") {
        tr = d.data_row(
          {
            className:"item "+data.name,
            text:data.prompt+"&nbsp;",
            value:data.value+"&nbsp;"
          }
        );
        d.push(tr,table);
      }
    }
```

That's all for the `items()` routine. Next up is the routine that handles the `queries` element of the Cj document. And here is example of the generated UI for Cj items:



ID	1l9fz7bhaho
Title	extensions
Tags	forms testing
Complete Flag	true
Asigned User	carol

Figure 5-3. Generated Cj Items

Queries

The `queries()` routine processes all the elements in the Cj `queries` collection and turns them into HTML "GET" FORMS. The code is not very complex but it is a bit verbose. It takes quite a few lines to generate an HTML FORM! The code our Cj Client uses for generating the UI for Cj `queries` is below.

```
    // handle query collection
    function queries() {
```

www.it-ebooks.info

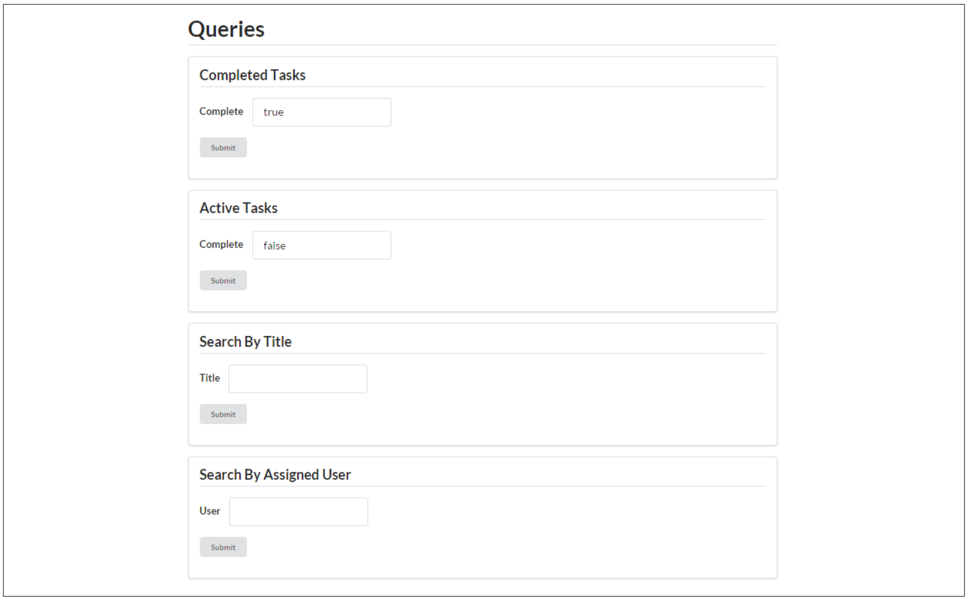
```
  var elm, coll;
  var segment;
  var form, fs, header, p, lbl, inp;

  elm = d.find("queries");
  d.clear(elm);
  if(g.cj.collection.queries) { ❶
    coll = g.cj.collection.queries;
    for(var query of coll) { ❷
      segment = d.node("div");
      segment.className = "ui segment";
      form = d.node("form"); ❸
      form.action = query.href;
      form.className = query.rel;
      form.method = "get";
      form.onsubmit = httpQuery;
      fs = d.node("div");
      fs.className = "ui form";
      header = d.node("div");
      header.innerHTML = query.prompt + "&nbsp;";
      header.className = "ui dividing header";
      d.push(header,fs);
      for(var data of query.data) { ❹
        p = d.input({prompt:data.prompt,name:data.name,value:data.value});
        d.push(p,fs);
      }
      p = d.node("p"); ❺
      inp = d.node("input");
      inp.type = "submit";
      inp.className = "ui mini submit button";
      d.push(inp,p);
      d.push(p,fs);
      d.push(fs,form);
      d.push(form,segment);
      d.push(segment,elm); ❻
    }
  }
}
```

The `queries` routine has just a few interesting points to cover:

1. First, see if there are any `queries` in this response to process
2. If yes, loop through each of them to build up a query form
3. Create the HTML `<form>` element and populate it with the proper details
4. Walk through each `data` element to create the HTML `<inputs>` that are needed
5. Then add the submit button to the form, and
6. Finally add the resulting markup to the UI for rendering on the page.

www.it-ebooks.info

That's how the Cj client handles generating all the 'safe' query forms (e.g. HTTP "GET"). There are a few parts that deal with the HTML layout that are left out here, but you can see the important aspects of the `queries()` routine. Here is an example of the generated query forms in our Cj client app.


Queries

Completed Tasks
Complete true
Submit

Active Tasks
Complete false
Submit

Search By Title
Title
Submit

Search By Assigned User
User
Submit

Figure 5-4. The Generated Cj Query FORMS

Template

Just as Cj `queries` describe safe actions (e.g. HTTP `GET`), the Cj `template` describes the unsafe actions (e.g. HTTP POST+ and `PUT`). The code looks very similar to the code for generating Cj `queries`.

```
// handle template object
function template() {
  var elm, coll;
  var form, fs, header, p, lbl, inp;

  elm = d.find("template");
  d.clear(elm);
  if(hasTemplate(g.cj.collection)===true) { ❶
    coll = g.cj.collection.template.data;
    form = d.node("form"); ❷
    form.action = g.cj.collection.href;
    form.method = "post";
    form.className = "add";
    form.onsubmit = httpPost;
    fs = d.node("div");
```

www.it-ebooks.info

```
        fs.className = "ui form";
        header = d.node("div");
        header.className = "ui dividing header";
        header.innerHTML = g.cj.collection.template.prompt||"Add";
        d.push(header,fs);
        for(var data of coll) { ❸
          p = d.input(
            {
              prompt:data.prompt+"&nbsp;",
              name:data.name,
              value:data.value,
              required:data.required,
              readOnly:data.readOnly,
              pattern:data.pattern
            }
          );
          d.push(p,fs);
        }
        p = d.node("p"); ❹
        inp = d.node("input");
        inp.className = "ui positive mini submit button";
        inp.type = "submit";
        d.push(inp,p);
        d.push(p,fs);
        d.push(fs,form);
        d.push(form, elm); ❺
      }
    }
```

Here are the highlights for the `template` routine:

1. Confirm there is a `template` element in the loaded Cj document
2. If there is, start building and populating an HTML `<form>`
3. Using the `template`'s `data` properties, create one or more HTML `<input>` elements
4. After all the inputs are created, add an HTML submit button
5. Finally, add the completed HTML FORM to the UI.

You will also notice in the code above that the HTML `<form>` element is set to use the `POST` method. This takes care of the CREATE use-case for Cj `template`. For the UPDATE use case, there is a *shadow* routine in the Cj client called `cjEdit()`. This is invoked when the user presses the "Edit" button generated for each `item`. I won't review the code for `cjEdit()` here (you can check out the source yourself) but will just mention that it looks almost identicial except for a few changes related to the HTTP PUT use case.

Here is an example of the Cj `template` rendered for the CREATE use-case.

www.it-ebooks.info

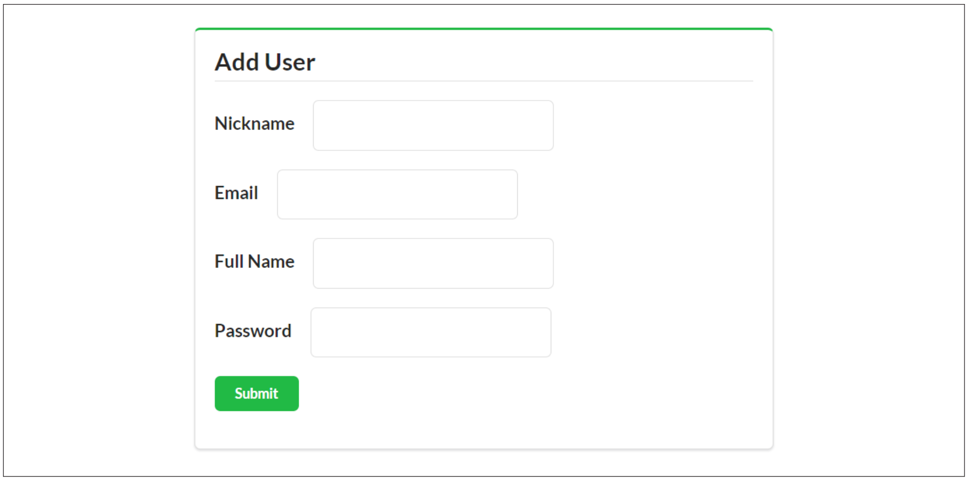


Figure 5-5. Generating the Cj CREATE UI

The only code left to review for the Cj Client is the code that handles any `error` elements in the response.

Error

Cj is the only hypermedia design featured in this book that has built-in support for sending domain-specific error information. The Cj `error` element is very simple. It has only four properties: `title`, `message`, `code`, and `url`. So the client routine for rendering errors is simple, too.

The code below shows that the Cj client app just echoes the properties of the `error` element in Cj responses directly to the screen.

```
// handle error object
function error() {
  var elm, obj;

  elm = d.find("error");
  d.clear(elm);
  if(g.cj.collection.error) {
    obj = g.cj.collection.error;

    p = d.para({className:"title",text:obj.title});
    d.push(p,elm);

    p = d.para({className:"message",text:obj.message});
    d.push(p,elm);

    p = d.para({className:"code",text:obj.code});
    d.push(p,elm);
```

www.it-ebooks.info

```
        p = d.para({className:"url",text:obj.url});
        d.push(p,elm);
      }
    }
```

Quick Summary

The Cj client has a number of differences from the HAL and Siren clients we reviewed earlier in the book. The most significant is the way domain objects are handled in Cj. Instead of just echoing a set of name-value pairs or even a nested JSON object graph, Collection+JSON documents only support returning flat lists of `items`. Each `item` represents more than just a domain object's properties. It also includes metadata about the domain object (`prompt` and `render` information) and a collection of one or more `link` elements associated with the domain object.

The way safe and unsafe actions are expressed in Cj is also unique. Instead of leaving it up to source code (as in HAL) or relying on a general model for all actions (as in Siren), the Cj design supports two different action elements: the `queries` and `template` elements. Cj `queries` are for safe actions (e.g. HTML `GET`) and the `template` is for unafe actions (e.g. HTTP `POST` and `PUT`).

The other main element in Cj documents is the `links` colletion and this is very similar to the way both HAL and Siren express links, too.

Now that we have a fully-functional Cj general client, let's introduce some modifications to the backend TPS API and see how it deals with backward compatible changes.

Dealing with Change

In previous chapters covering the JSON (TK), HAL, and Siren SPA Clients, I introduced various backward-compatible changes to the TPS API in order to exlore the run-time adaptability of the client. The changes all dealt with one or more changes the three key aspects Web API clients need to deal with: OBJECTS, ADDRESSES, and ACTIONS. How the client apps reacted to the changes gave us an indication of their adaptability using our OAA Challenge.

For the Cj client, I'll introduce and entirely new domain object (NOTES) along with a full set of ACTIONS and ADDRESSSES. This level of change represents examples of all the kinds of changes we've introduced to the other SPA clients before. This will test the Cj Client's ability to recognize and deal with domain objects and operations that were introduced long after the initial production release of the API and client implementations.

www.it-ebooks.info

Adding the NOTE Object to the TPS API

Let's assume that the TPS team decides to add support for attaching comments or NOTES to TASK records in the TPS API. That means defining a NOTE object's fields and adding support for the basic CRUD operations on NOTE objects along with some other NOTE-specific actions like filters, etc.

The source code for the updated Cj Representor with NOTE support can be found in the associated github repo here: *https://github.com/RWCBook/cj-client-note*. A running version of the app described in this chapter can be found here: *http://rwcbook13.herokuapp.com/files/cj.client.html* (TK: check URLs)

In this section, I'll review the API design elements (internal NOTE object and public API), the resulting WeSTL document, and take a look at a bit of the server code. Then, after completing the backend changes, we'll fire up the Cj client and see what happens.

The NOTE API Design

Our NOTE object will have a small set of fields, support the basdic CRUD operations, a couple filters, and a custom `NoteAssignTask` operation. The table below shows the NOTE object properties:

Table 5-1. Note Object Properties

Property	Type	Status	Default
id	string	required	none
title	string	required	none
text	string	optional	none
assignedTask	taskID	required	none

Along with the Create-Read-Update-Delete (CRUD) asctions, we'll need a couple filters (`NoteListByTitle` and `NoteListByText`) that allows users to enter partial strings and find all the NOTE records that contain that string. We'll also add a special operation to assign a NOTE to a TASK (`NoteAssignTask`) that takes to `id` values (a NOTE `id` and a TASK `id`). The table below lists all the operations, arguments, and HTTP protocol deatils.

www.it-ebooks.info

Table 5-2. TPS Note Object API

Operation	URL	Method	Returns	Inputs
NoteList	`/note/`	GET	NoteList	none
NoteAdd	`/note/`	POST	NoteList	`id,` `title,` `text,` `asssignedTask`
NoteItem	`/note/{id}`	GET	NoteItem	none
NoteUpdate	`/note/{id}`	PUT	NoteList	`id,` `title,` `text,` `asssignedTask`
NoteRemove	`/note/{id}`	DELETE	NoteList	none
NoteAssignTask	`/note/assign/{id}`	POST	NoteList	`id,` `assignedTask`
NoteListByTitle	`/note/`	GET	NoteList	`title`
NoteListByText	`/note/`	GET	NoteList	`text`

That's all we need on the design side. Let's look at how we'll turn this design into a working API in our TPS service.

The NOTE API Service Implementation

I won't go into the details of the internal code (data and object manipulation) for implementing the NOTES object support in the TPS API. However, it is worth pointing out a few things on the interface side since it affects how we'll set up the Cj responses sent to the existing client.

The first thing to add is the component code that defines the object described in Table 9.1 above. This code also vcalidates inputs and enforces relationship rules (e.g. making sure users don't assign NOTE records to non-existent TASK records). In the TPS API service, the NOTE object definition looks like this:

```
props = ["id","title","text","assignedTask","dateCreated","dateUpdated"]; ❶
elm = 'note';

// shared profile info for this object ❷
profile = {
  "id" : {"prompt" : "ID", "display" : true},
```

www.it-ebooks.info

```
    "title" : {"prompt" : "Title", "display" : true},
    "text" : {"prompt" : "Text", "display" : true},
    "assignedTask" : {"prompt" : "Assigned Task", "display" : true},
    "dateCreated" :  {"prompt" : "Created", "display" : false},
    "dateUpdated" :  {"prompt" : "Updated", "display" : false}
  };
```

Notice that the props array (callout #1) defines valid fields for a NOTE and the profile object (callout #2) contains the rules for displaying objects to users (e.g. the prompt and display flags).

Below is the addTask routine for the note-component.js server-side code. It shows how the component builds up a new NOTE record to store (callout #1) and validates the inputs (#2) including checking for the existence of the supplied assignedTask ID (callout #3). Then, as long as there are no errors found, the code sends the new NOTE record off for storage (#4).

```
function addNote(elm, note, props) {
  var rtn, item, error;

  error = "";

  item = {} ❶
  item.title = (note.title||"");
  item.text = (note.text||"");
  item.assignedTask = (note.assignedTask||"");

  if(item.title === "") { ❷
    error += "Missing Title ";
  }
  if(item.assignedTask==="") {
    error += "Missing Assigned Task ";
  }
  if(component.task('exists', item.assignedTask)===false) { ❸
    error += "Task ID not found. ";
  }

  if(error.length!==0) {
    rtn = utils.exception(error);
  }
  else {
    storage(elm, 'add', utils.setProps(item,props)); ❹
  }

  return rtn;
}
```

That's enough of the internals. Now let's look at the interface code — the WeSTL entries that define the transitions for manipulating NOTE objects, and the resource code that handles the HTTP protocol requests exposed via the API.

www.it-ebooks.info

Here are some of the transition descriptions for NOTE objects. I've included the noteFormAdd transition that will populate the Cj template (callout #1) and the two transitions for the "Assign Task" action: the one that offers a link to the form (callout #2) and the template for making the assignement (callout #2).

```
// add task ❶
trans.push({
  name : "noteFormAdd",
  type : "unsafe",
  action : "append",
  kind : "note",
  target : "list add hal siren cj-template",
  prompt : "Add Note",
  inputs : [
    {name : "title", prompt : "Title", required : true},
    {name : "assignedTask", prompt : "Assigned Task", required : true},
    {name : "text", prompt : "Text"}
  ]
});

...

trans.push({ ❷
  name : "noteAssignLink",
  type : "safe",
  action : "read",
  kind : "note",
  target : "item cj read",
  prompt : "Assign Task",
});
trans.push({ ❸
  name : "noteAssignForm",
  type : "unsafe",
  action : "append",
  kind : "note",
  target : "item assign edit post form hal siren cj-template",
  prompt : "Assign Task",
  inputs : [
    {name: "id", prompt:"ID", readOnly:true, required:true},
    {name: "assignedTask", prompt:"Task ID", value:"", required : true}
  ]
});
```

Because the Cj media type design relies heavily on the CRUD pattern, unsafe operations that don't easily fall into the CRUD model (in this case, the noteAssignForm opertation) need to be handled differently. In Cj, these non-standard CRUD actions are offered as templates and executed with an HTTP POST — the way you'd create a new object in a standard CRUD pattern.

www.it-ebooks.info

To support this, I need *two* transitions. One that returns the "assign template" (noteAssignLink) and the other that accepts the POST call to commit the assign arguments to storage (noteAssingForm). Since WeSTL doesn't supply URLs, the source code (in the /connectors/note.js file on the server) does that at runtime. Here's what that snippet of code looks like:

```
// add the item-level link
wstl.append({name:"noteAssignLink",href:"/note/assign/{id}",
  rel:["edit-form","/rels/noteAssignTask"],root:root},coll);

// add the assign-page  template
wstl.append({name:"noteFormAdd",href:"/note/",
  rel:["create-form","/rels/noteAdd"],root:root},coll);
```

Finally, I'll need to account for this when handling HTTP requests, too. Here is the code that responds to the HTTP GET for the "assign page" (e.g. /note/assign/1qw2w3e)

```
case 'GET':
  if(flag===false && parts[1]==="assign" && parts[2]) {
    flag=true;
    sendAssignPage(req, res, respond, parts[2]);
  }
```

And here's the snippet of code that responds to the HTTP POST request that commits the assignment:

```
case 'POST':
  if(parts[1] && parts[1].indexOf('?')===-1) {
    switch(parts[1].toLowerCase()) {
      case "assign":
        assignTask(req, res, respond, parts[2]);
      break;
    }
  }
```

There is more in the server-side code (e.g adding the page-level link to the new NOTES API, etc.) that you can check out yourself. The point here is that Cj forces API designers to explicitly account for the non-CRUD unsafe actions (via POST) right up front. This is a bit more work for API designers (well, you'd have to do it eventually anyway) but it makes support in the Cj client much easier. In fact, that support is *already there*.

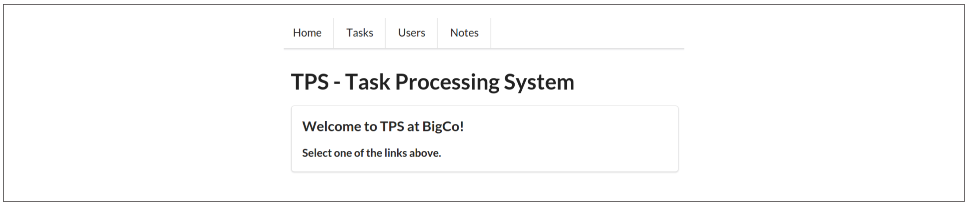
So let's see what happens when we fire up our existing, unchanged Cj client against this updated TPS API.

Testing the NOTE API With the Exsiting Cj Client

In a real-life scenario, the TPS API would be updated into production without prior warning to all the exasiting Cj clients. Then, at some point one of the client might

www.it-ebooks.info

make an initial request to the TPS API, just as in previous days and automatically see the new “Notes” option at the top of the page (see image below).



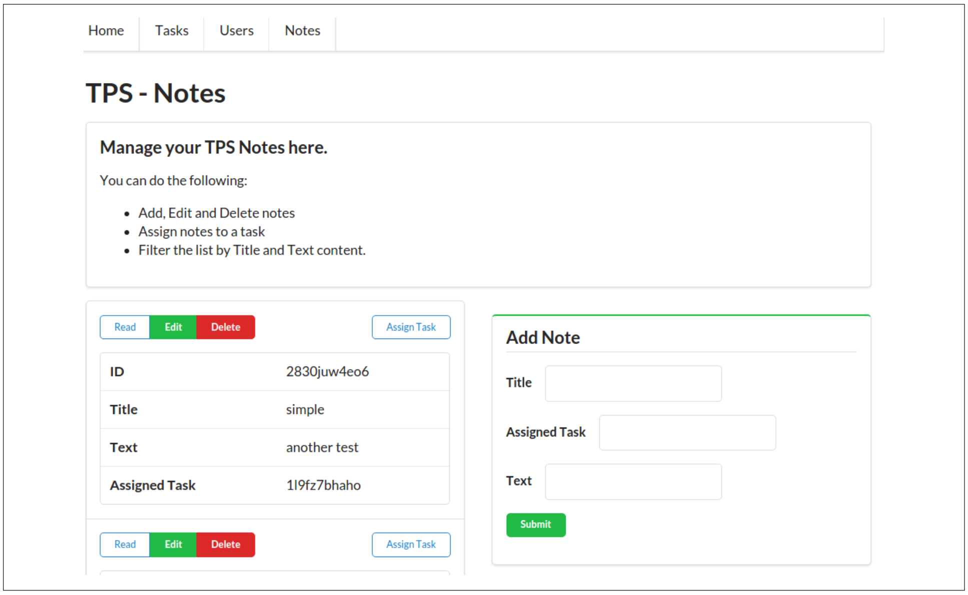
TPS - Task Processing System

Welcome to TPS at BigCo!

Select one of the links above.

Figure 5-6. New NOTES option in the Cj Client

When the user clicks on the “Notes” link, the fully-populated interface comes up with all the display and input constraints enforced. Along with the expected “Read”, “Edit”, and “Delete” buttons for each item plus the “Add Note” form, users will also see (in the image below) the special “Assign Task” link that appears for each NOTE in the list.



TPS - Notes

Manage your TPS Notes here.

You can do the following:

- Add, Edit and Delete notes
- Assign notes to a task
- Filter the list by Title and Text content.

ID	2830juw4eo6
Title	simple
Text	another test
Assigned Task	1l9fz7bhaho

Add Note

Title

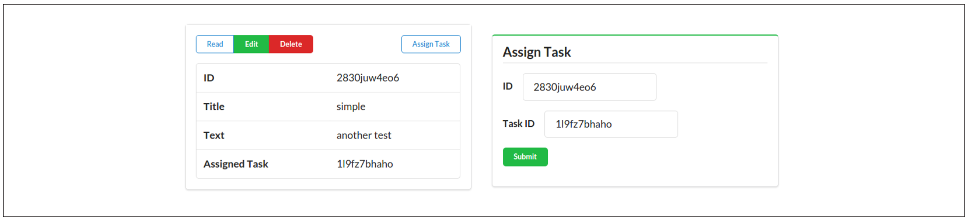
Assigned Task

Text

Figure 5-7. The NOTES Page in the Cj Client

Finally, clicking on the “Assign Task” button will bring up the screen that prompts users to enter the `id` value of the TASK to which this note should be attached (see image below).

www.it-ebooks.info



ID	2830juw4eo6
Title	simple
Text	another test
Assigned Task	1l9fz7bhaho

Assign Task

ID 2830juw4eo6

Task ID 1l9fz7bhaho

Figure 5-8. Assigning a Note to a Task

So, the existing Cj client was able to support all the new API functionality (the new OBJECT, ADDRESSES, and ACTIONS) without any new coding. At this point the TPS Web API team has quite a bit of freedom to modify the existing production API. As long as the changes are backward-compatible, not only will the Cj client not *break* when new changes appear, the new changes will be fully supported (not just safely ignored) for now and into the future.

But things are not perfect w/ the Cj client. As you might have noticed that last couple screens, entering the text of a note is a problem — the text box is too small. It should be rendered as an HTML `<textarea>` control to support longer text entry and even scroll bars. Even more problematic is the "Assign Task" data entry. There users are expected to supply two rather opaque record identifiers (NoteID and TaskID) in order to complete the "Assign Task" action. That's not very user-friendly and it is likely to be rejected by any team responsible for building quality user experiences.

To fix this short-coming, we'll need to extend the Cj design to do a better job of describing (and supporting) more user-friendly input experiences. There are lots of options for improvement but I'll just focuse on one of them for now: supporting a dropdown or suggested list of possible values for input.

Extending Collection+JSON

Cj has powerful support for passing metadata on links (ADDRESSES), forms (ACTIONS), and domain-objects (OBJECTS). However, it has weak support for passing metadata about user inputs. The ability to indicate input properties such as `required`, `readOnly`, and `pattern` (all directly from HTML5) is a start, but more is needed. Swiber's Siren (TK ref?), for example, has much stronger support for input metadata.

The good news is that Cj has a clear option for creating extensions to fill in gaps in the design. And that's what I'll do here. This section outlines an extension for supporting a `type` attribute for Cj `data` elements (ala HTML5's `type` attribute) and a `suggest` attribute to provide input metadata similar to that of the HTML `<select>` input control.

www.it-ebooks.info

The source code for the updated Cj Representor with NOTE support can be found in the associated github repo here: *https://github.com/RWCBook/cj-client-types*. A running version of the app described in this chapter can be found here: *http://rwcbook14.herokuapp.com/files/cj.client.html* (TK: check URLs)

These extensions will improve Cj client apps' ability to provide a solid user experience.

Supporting Improved Input Types

Adding support for HTML5-style input types (e.g. `email`, `number`, `url`, etc.) is a pretty simple extension for Cj. It's just a matter of adding the `type` property to the Cj output and honoring it on the client side. There are also a series of associated properties like `min`, `max`, `size`, `maxlength`, etc. that should also be supported.

Below is an example of what extended `type` support in Cj would look like in the Cj representation. Note the use of the `pattern` (callout #1) and `"type":"email"` (#2).

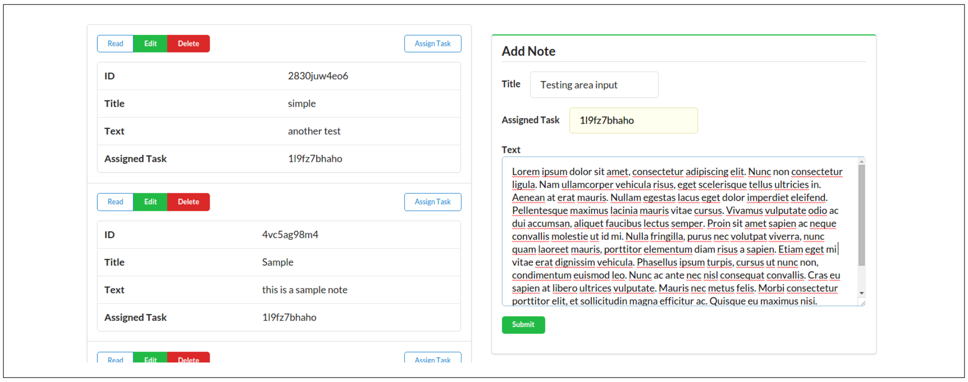
```
"template": {
  "prompt": "Add User",
  "rel": "create-form userAdd create-form",00
  "data": [
    {"name": "nick","value": "","prompt": "Nickname","type": "text",
      "required": "true","pattern": "[a-zA-Z0-9]+"}, ❶
    {"name": "email","value": "","prompt": "Email","type": "email"}, ❷
    {"name": "name","value": "","prompt": "Full Name","type": "text",
        "required": "true"},
    {"name": "password","value": "","prompt": "Password","type":"text",
        "required": "true","pattern": "[a-zA-Z0-9!@#$%^&*-]+"} ❶
  ]
}
```

Another handy HTML UI element is the `<textarea>` element. Adding support for `textarea` in Cj is also pretty simple. The Cj template would look like this (see callout #1):

```
"template": {
  "prompt": "Add Note",
  "rel": "create-form //localhost:8181/rels/noteAdd",
  "data": [
    {"name": "title","value": "","prompt": "Title",
      "type": "text","required": "true"},
    {"name": "assignedTask","value": "","prompt": "Assigned Task",
      "type": "text","required": "true"},
    {"name": "text","value": "","prompt": "Text",
      "type": "area","cols": 40,"rows": 5} ❶
  ]
}
```

www.it-ebooks.info

And here's how the CJ client screen looks w/ support for the `area` input type added:



ID	2830juw4eo6
Title	simple
Text	another test
Assigned Task	1l9fz7bhaho

ID	4vc5ag98m4
Title	Sample
Text	this is a sample note
Assigned Task	1l9fz7bhaho

Add Note

Title: Testing area input

Assigned Task: 1l9fz7bhaho

Text:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc non consectetur ligula. Nam ullamcorper vehicula risus, eget scelerisque tellus ultricies in. Aenean at erat mauris. Nullam egestas lacus eget dolor imperdiet eleifend. Pellentesque maximus lacinia mauris vitae cursus. Vivamus vulputate odio ac dui accumsan, aliquet faucibus lectus semper. Proin sit amet sapien ac neque convallis molestie ut id mi. Nulla fringilla, purus nec volutpat viverra, nunc quam laoreet mauris, porttitor elementum diam risus a sapien. Etiam eget mi vitae erat dignissim vehicula. Phasellus ipsum turpis, cursus ut nunc non, condimentum euismod leo. Nunc ac ante nec nisl consequat convallis. Cras eu sapien at libero ultrices vulputate. Mauris nec metus felis. Morbi consectetur porttitor elit, et sollicitudin magna efficitur ac. Quisque eu maximus nisi.

Figure 5-9. Adding Cj Support for Textarea Input

So, adding support for most HTML5-style inputs is rather easy. But there are a couple HTML5-style inputs that take a bit more effort and one of them is badly needed for the TPS user experience — the `<select>` or 'dropdown' list.

The `suggest` Object

Supporting HTML-style dropdown lists takes a bit of planning. I'll need to make modifications to the Cj document design, Representor, and the Client library. I won't go through a lot of detail here — just the highlights.

Updating the Cj Design

First, we'll need a way to communicate the dropdown input within the Cj `data` elements. The design I chose allows for two types of implementation: 'direct' content, and 'related' content. I'll explain the differences as we go along.

Here is a sample `suggest` element that uses the 'direct' content approach:

```
data :
  [
    {
    "name": "completeFlag",
    "value": "false",
    "prompt": "Complete",
    "type": "select", ❶
    "suggest": [ ❷
      {"value": "false", "text": "No"},
      {"value": "true", "text": "Yes"}
    ]
  }
]
```

www.it-ebooks.info

At callout #1, the new type attribute is set to "select" and, at callout #2, the suggest element is an array with an object that holds both the value and the text for HTML <select> elements.

The other type of suggest implementation I want to support is what I call the 'related' model. It uses related data in the response as content for the dropdown list. That means I need to add a new element to the Cj document's root: related. This Cj root element is a named object with one or more named JSON arrays that hold content for a dropdown list. Here's what that looks like (see callout #1):

```
{
  "collection": {
    "version": "1.0",
    "href": "//localhost:8181/task/assign/1l9fz7bhaho",
    "links": [...],
    "items": [...],
    "queries": [...],
    "template": {...},
    "related": { ❶
      "userlist": [
        {"nick": "alice"},
        {"nick": "bob"},
        {"nick": "carol"},
        {"nick": "fred"},
        {"nick": "mamund"},
        {"nick": "mook"},
        {"nick": "ted"}
      ]
    }
  }
}
```

And here's the matching implementation for the suggest attribute (#1) for Cj data elements:

```
data: [
  {
    "name": "assignedUser",
    "value": "mamund",
    "prompt": "User Nickname",
    "type": "select",
    "required": "true",
    "suggest": {"related": "userlist","value": "nick","text": "nick"} ❶
  }
]
```

Now, Cj client applications can find the related data in the response (by the value of related) and use the property names listed in the suggest element to populate the list.

www.it-ebooks.info

Updating the Cj Representor

We need to include the `related` property in the output of the Cj Representor. That's pretty easy. We just create a small function to pull any related content into the response (callout #1) and add that to the top-level routine that builds up the Cj response document (callout #2):

```
// handle any related content in the response
function getRelated(obj) {
  var rtn;

  if(obj.related) {
    rtn = obj.related;
  }
  else {
    rtn = {};
  }
  return rtn;
}

...

// building the cj response document
rtn.collection.title = getTitle(object[o]);
rtn.collection.content = getContent(object[o]);
rtn.collection.links = getLinks(object[o].actions);
rtn.collection.items = getItems(object[o],root);
rtn.collection.queries = getQueries(object[o].actions);
rtn.collection.template = getTemplate(object[o].actions);
rtn.collection.related = getRelated(object[o]); ❷
```

Updating the Cj Client library

The Cj client library has a handful of things to deal with now including:

- Recognizing the new `suggest` attribute in responses
- Locating any possible `related` content in the responses
- Parsing the `suggest` element into a valid `<select>` element in the UI
- Processing the value of the `<select>` element and including it in the POST and PUT actions

Most of this work will happen in my `dom-help.js` routine — that's where the request to creat an input element in the UI takes place. Here is a snippet of code I added to the `input(args,related)` routine:

```
....
if(args.type==="select" || args.suggest) { ❶
  inp = node("select");
  inp.value = args.value.toString()||"";
```

www.it-ebooks.info

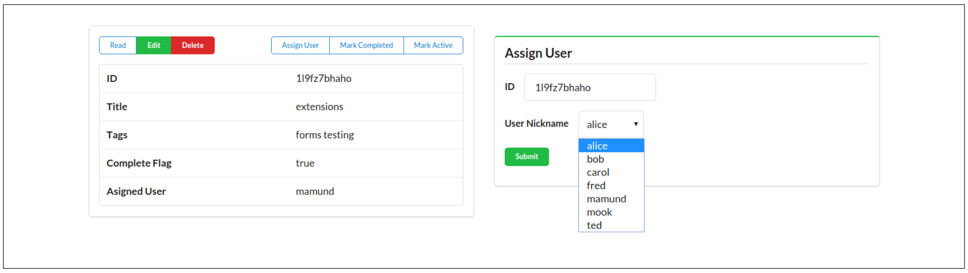
```
      inp.className = "ui dropdown ";
      if(Array.isArray(args.suggest)) { ❷
        for(var ch of args.suggest) {
          opt = option(ch);
          push(opt,inp);
        }
      }
      if(related) { ❸
        lst = related[args.suggest.related];
        if(Array.isArray(lst)) { ❹
          val = args.suggest.value;
          txt = args.suggest.text;
          for(var ch of lst) { ❺
            opt = option({text:ch[txt],value:ch[val]});
            push(opt,inp);
          }
        }
      }
    }
    ....
```

In the code above:

1. See if this is a `suggest` control
2. If there is an array of values, use that to build the `<option>` elements for the HTML `<select>` control
3. Check to see if a pointer to the `related` content in the response was passed
4. If it was, and it returns a valid array of data,
5. Use that content to build up the `<option>` elements.

There are some additional client-side library changes to manage the details and collect and send selected values back to the service. You can check out the source code for details.

Once all this is in place, the UI for screens like "Assign Task" look much more inviting:



ID	1l9fz7bhaho
Title	extensions
Tags	forms testing
Complete Flag	true
Asigned User	mamund

Assign User

ID 1l9fz7bhaho

User Nickname alice

alice
bob
carol
fred
mamund
mook
ted

Figure 5-10. Adding Dropdown Support to Cj

www.it-ebooks.info

Now, with the Suggest extension and added support for improved input metadata, Cj offers not only fully-functional support for adding new OBJECTS to the backend API, but it also has better user interface support. It is worth pointing out that most of the input support I added to Cj as extensions already exsits as part of the design for Siren (TK).

Reader Challenge

My `suggest` implementation has two modes: ‘direct’ and ‘related’. There is at least one more mode that I didn’t implement that I’d really like to see: ‘web’ mode. In Web mode, the `suggest.related` value is a valid URL pointing to an API response that returns the list of choices. It could be used to create a simple drop down or it could be used to implement a key-stroke experience that performs a search on each key-press and returns suggested results. I’ll leave the details to my intrepid readers to work out on their own — and submit as an update to the online github repo.

Summary

In previous chapters covering the JSON (TK), HAL, and Siren SPA Clients, I introduced various backward-compatible changes to the TPS API in order to exlore the run-time adaptability of the client. The changes all dealt with one or more changes the three key aspects Web API clients need to deal with: OBJECTS, ADDRESSES, and ACTIONS. How the client apps reacted to the changes gave us an indication of their adaptability using our OAA Challenge.

Here’s a quick summary of our experience so far:

JSON Client
: Changes to URLs, adding fields, or actions all were ignored by the client app. That gave us no “wins” on the OAA Challenge.

HAL Client
: Changes to URLs did not adversely affect the client. However, changes to domain objects and actions were ignored. That’s 1 “win” for HAL: ADDRESSES.

Siren Client
: Changes to URLs and actions were all recognized and supported in the Siren client. However, changes to domain objects were not picked up by the client. That’s 2 “wins” for Siren: ADDRESSES and ACTIONS.

Cj Client
: As we saw in this chapter, adding an entirely new domain object (NOTES) was automatically picked-up by the Cj client. All the URLs, fields, and operations

www.it-ebooks.info

appeared in the UI w/o the need to make any changes to the client code. That gives Cj all 3 “wins” in the OBJECTS-ADDRESSES-ACTIONS Challenge.

Of couse, along the way, each client was enchanced through the application of extensions (HAL-FORMS, SIREN-SOP, and Cj-Suggest). And there are other possible ways in which to improve both the adaptability and the user experience of all of the client implementations reviwed in the book. But the Collection+JSON design offers clients the widest possible range of adaptability for the kinds of API changes that are most commonly experienced by API client apps.

The Relationship between Data and Metadata

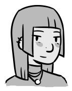
And that brings up an important pattern that has run throughout the examples in the book: the relationship between metadata and adaptability. As you scan the message models used for our client examples you’ll find that each one, in order, has provided increased levels of metadata in the API responses. From the plain JSON responses that contain *no* metadata on up the the Cj responses which (sometimes) contain more metadata than ‘data.’ And, with each increase in metadata, the client libraries gain more ability to adapt to changes in the backend API. It follows that, if you want to improve the adaptability of your client applications, you need to focus on the metadata shared in responses.

It should be noted that not all production implementations need to support adaptabiltiy on all three of the axes featured here. The application of HAL, Siren, Cj (or other media types) makes sense when they help solve real problems. It it not at all likely that one format is appropriate for all cases and it is up to API designers and implementers to select the formats that provide the best feature match for the expected use-cases.

Having said that, there is one more challenge worth exploring: the challenge of implementing a single SPA application that supports more than one hypermedia format — and can automatically switch between formats at runtime based on the service response.

And that’s the challenge we’ll take up in the next — and final — chapter of the book.

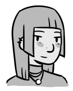
Bob and Carol



“Well, Bob, Collection+JSON turned out to be a very interesting hypermedia type.”

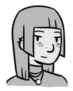
www.it-ebooks.info

“Yes, it did, Carol. My team tells me they had no trouble putting together the representor for it, too. Even with the added metadata in the responses.”



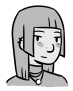
“Right, and that added metadata really helped our client team build a solid general client app. The Cj app seems much more adaptable to backend API changes than any of the others we’ve tried so far.”

“That’s encouraging, Carol. Although I will say I heard some on the API design side saying they were finding some API operations were a bit more challenging to implement due to Cj’s CRUD support.”



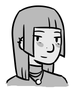
“Yes, we heard that, too, Bob. Turns out operations like “Assign user” in our TPS API don’t map easily to the simple CRUD pattern.”

“Yep, but our API team was able to provide item-level links that point to responses with POST templates to make it all work.”



“And by doing that, our client app was able to support the new NOTES functionality without any client-side updates. That was great.”

“Yep. The API team was really pleased when they realised that Cj gives them the ability to introduce that level of new functionality and have it automatically appear in the client app.”



“Of course, we needed to add some extensions to improve the human-drive UI. Cj was missing several things that we had in Siren, for example. The good news is that adding UI extensions seems really easy in Cj.”

“Well, Carol. I think this was the most successful general hypermedia client app we’ve built so far.”

www.it-ebooks.info

"I agree, Bob. Kind of makes me wonder if there are any more challenges we need to address."

"Actually, I think there is. My server side team has been thinking about implementing a Microservice-style backend and that might cause us some trouble for these hypermedia clients."

"You think so?"

"Well, I need to check back with my team. Let's meet again tomorrow to discuss it, OK?"

"OK, Bob. See you tomorrow."

References

1. The most up-to-date specification docs for Cj can be found at *http://amundsen.com/media-types/collection*
2. You'll find various Cj examples and extensions in the online github repository at *https://github.com/collection-json*
3. The Google discussion group for Cj is archived at *https://groups.google.com/forum/?fromgroups=#!forum/collectionjson*
4. The HTML `<input>` element has quite a few options. Check out the docs online at the W3C website.
5. The specification for the HTML `<textarea>` element can be found at the W3C web site.

Images Credits

- Diogo Lucas: Figure 1

www.it-ebooks.info