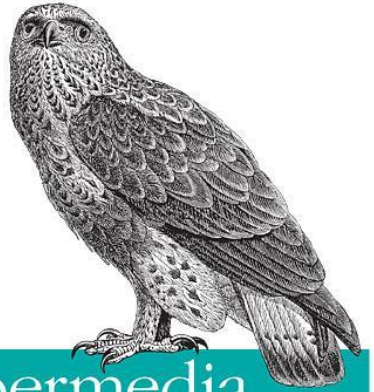
Designing for Reuse: Creating APIs for the Future

Mike Amundsen,
Layer 7 / CA
@mamund

Introduction


Ctrl
PgUp
Mike Amundsen
@mamund

Creating Evolvable Hypermedia Applications



Building

Hypermedia APIs with HTML5 & Node

O'REILLY®

Mike Amundsen

O'REILLY®

Designing APIs for the Web

Mike Amundsen

VIDEO

Services for a Changing World

RESTful Web APIs

O'REILLY®

Leonard Richardson, Mike Amundsen & Sam Ruby

API Design Training | API /

www.apiacademy.co

API ACADEMY

Search API Academy

API Strategy | API Design | API Management | Resources | About

Register | Sign In

Your Guide to API Design & Implementation Best Practices

API Academy delivers free online lessons and in-person consulting services covering essential API techniques and tools for business managers, interface designers and enterprise architects

What is an API?

Get an overview of what an API is and what it does, to help you realize the

API Design Basics

Understand the API architecture process and learn basic design and

Web API Architectural Styles

Get a detailed overview of the main architectural styles for Web and

Choosing a Solution

Choose between the various solutions that offer the basic components for

The Challenge

CHANGE IS INEVITABLE.
CHANGE IS CONSTANT.

Versioning

The Costs of Versioning an API

by **Mark Little** on Dec 01, 2013 | 16 **Discuss**

Share

Contract versioning and API/Service versioning has always been a consideration for SOA based systems. Whether because of the impact it has on composability, or client-service governance, it is still something of an art rather than a science. There are many examples of groups giving the benefit of their experiences (e.g., around REST is extremely popular). However, recently Jean-Jacques Dubray has written an article which attempts to inject some scientific objectivity into this problem domain.

> I have been asked recently to create an estimate of the costs of versioning APIs (or Web Services). I wanted to share this estimate because I feel a lot of people still don't understand the cost implications of API/Service versioning.

According to JJ, during the work they found that the cost of building APIs was dependent upon the approach used subsequently to version them.

Content removed by author’s request

Content removed by author’s request


Dropbox

Deprecate version after a year

Content removed by author’s request

Content removed by author’s request


salesforce®

They release a new version every 120 days

Currently supporting 20+ parallel versions

Content removed by author’s request

Content removed by author’s request


PayPal™

No breaking changes

Content removed by author’s request

To Version
means
"to turn"

There's another word for backward-compatible versioning...

Backward-compatible versioning is essentially creating extensions

So, how do you enable backward-compatible API extensions?

I'll talk about two ways today...

Messages (on the wire)...

Implementation (in the code)

Message Design

Extending Messages

```
{
  "url" : "http://api.example.com/1",
  "givenName" : "Mike",
  "familyname" : "Amundsen",
  "email" : "mca@amundsen.com",
  "phone" : "123-456-7890"
}
```

Extending Messages

```
{
  "url" : "http://api.example.com/1",
  "givenName" : "Mike",
  "familyname" : "Amundsen",
  "email" : "mca@amundsen.com",
  "phone" : "123-456-7890",
  "ext" : [
    {"name" : "workPhone", "value" : "234-567-8901"},
    {"name" : "workEmail", "value" : "mike.amundsen@example.com"}
  ]
}
```

Extending messages let's you easily add backward-compatible changes

Structure vs Data

```
{
  "id" : "q1w32e3r4",
  "url" : "http://api.example.com/1",
  "person" : {
   "givenName" : "Mike",
   "familyname" : "Amundsen",
   "email" : "mca@amundsen.com",
   "phone" : "123-456-7890"
  },
  "address" : {
    "street1" : "123 Main",
    "street2" : "Apt #1",
    "city" : "Byteville",
    "stateRegion" : "MD",
    "postalCode" : "12345"
  }
}
```

Structure vs Data

```
{
  "id" : "q1w32e3r4",
  "url" : "http://api.example.com/1",
  "givenName" : "Mike",
  "familyname" : "Amundsen",
  "email" : "mca@amundsen.com",
  "phone" : "123-456-7890",
  "street1" : "123 Main",
  "street2" : "Apt #1",
  "city" : "Byteville",
  "stateRegion" : "MD",
  "postalCode" : "12345"
}
```

Focus on passing data,
not structure


New
and
Improved!

There are several new formats designed specifically for passing data on the Web

application/hal+json

```
{
    "_links": {
        "self": { "href": "/orders" },
        "curies": [{ "name": "ea", "href": "http://example.com/docs/rels/{rel}",
        "next": { "href": "/orders?page=2" },
        "ea:find": {
            "href": "/orders{?id}",
            "templated": true
        },
        "ea:admin": [{
            "href": "/admins/2",
            "title": "Fred"
        }, {
            "href": "/admins/5",
            "title": "Kate"
        }]
    },
    "currentlyProcessing": 14,
    "shippedToday": 20,
    "_embedded": {
        "ea:order": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "ea:basket": { "href": "/baskets/98712" },
                "ea:customer": { "href": "/customers/7809" }
            },
```

2. Collection Representation

A typical Collection+JSON will contain a set of `links`, list of `items`, a `queries` collection, an

```
{ "collection" :
  {
    "version" : "1.0",
    "href" : "http://example.org/friends/",

    "links" : [
      {"rel" : "feed", "href" : "http://example.org/friends/rss"}
    ],

    "items" : [
      {
        "href" : "http://example.org/friends/jdoe",
        "data" : [
          {"name" : "full-name", "value" : "J. Doe", "prompt" : "Full Name"},
          {"name" : "email", "value" : "jdoe@example.org", "prompt" : "Email"}
        ],
        "links" : [
          {"rel" : "blog", "href" : "http://examples.org/blogs/jdoe", "prompt" : "Blog"}
          {"rel" : "avatar", "href" : "http://examples.org/images/jdoe", "prompt" : "Ava
        ]
```

The media type for JSON Siren is `application/vnd.siren+json`.

```
{
  "class": [ "order" ],
  "properties": {
      "orderNumber": 42,
      "itemCount": 3,
      "status": "pending"
  },
  "entities": [
    {
      "class": [ "items", "collection" ],
      "rel": [ "http://x.io/rels/order-items" ],
      "href": "http://api.x.io/orders/42/items"
    }
```

Example <data> Elements (XML)

```
<uber version="1.0">
  <data rel="self" url="http://example.org/" />
  <data name="list" rel="collection" url="http://example.org/
  <data name="search" rel="search collection" url="http://exar
  <data name="todo" rel="item http://example.org/rels/todo" u
    <data name="title">Clean House</data>
    <data name="dueDate">2014-05-01</data>
  </data>
  <data name="todo" rel="item http://example.org/rels/todo" u
    <data name="title">Paint the fence</data>
    <data name="dueDate">2014-06-01</data>
  </data>
</uber>
```

Here's an example response from JSON API:

```
{
  "links": {
    "posts.author": {
      "href": "http://example.com/people/{posts.author}",
      "type": "people"
    },
    "posts.comments": {
      "href": "http://example.com/comments/{posts.comments}",
      "type": "comments"
    }
  },
  "posts": [{
    "id": "1",
    "title": "Rails is Omakase",
    "links": {
      "author": "9",
      "comments": [ "5", "12", "17", "20" ]
```

What's the common theme in these new designs?

Message over Object

"[I]t is far easier to standardize representation and relation types than it is to standardize objects and object-specific interfaces."

- Roy T. Fielding

The most common data-passing format on the Web is…

The most common data-passing format on the Web is…

```
givenName=Mike&familyName=Amundsen&phone=123-456-7890
```

Because it is easy to extend.

```
givenName=Mike&familyName=Amundsen&phone=123-456-7890
```

Message design is not the only place to plan for extensions

Implementation Design

Component != Connector

Component

- Database
- File System
- Message Queue
- Transaction Manager
- Source Code

Component == IP

Component == $$$

Component == Private

Connector

- Web Server
- Browser Agent
- Proxy Server
- Shared Cache

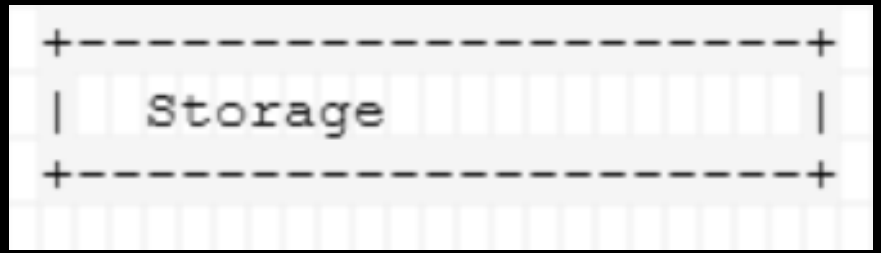
Connector == Shared Tech

Connector == Commodity

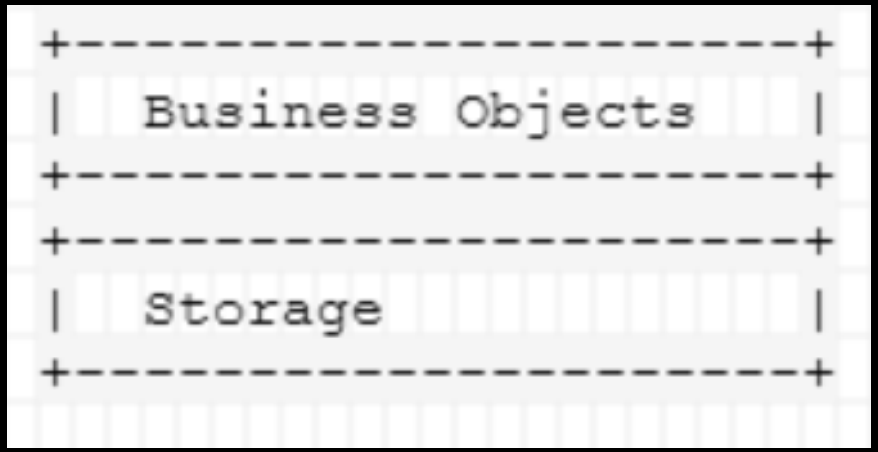
Connector == Public

Keep Connectors and Components separated

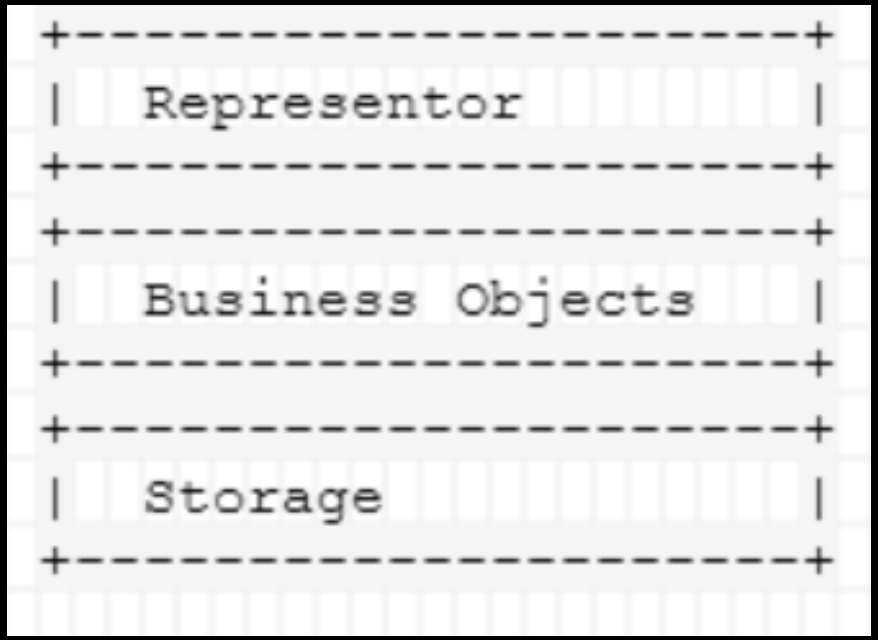
Implementation Stack



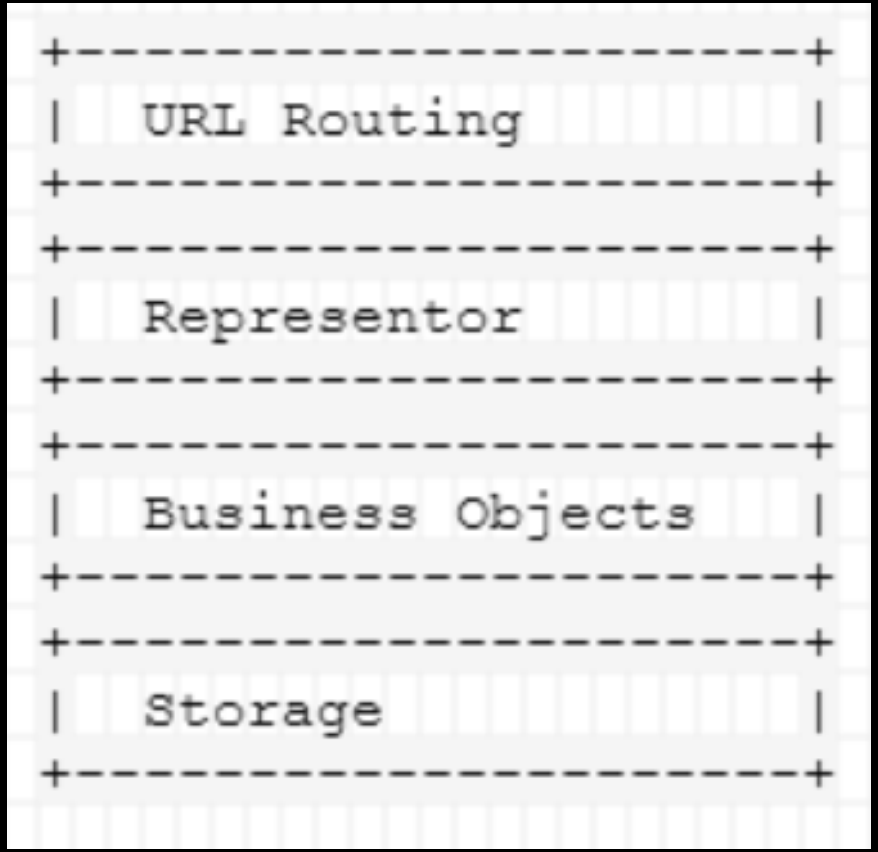
Implementation Stack



Implementation Stack



Implementation Stack



Class Schedule Server

```
22  // internal modules
23  var storage = require('./storage.js');
24  var component = require('./component.js');
25  var representation = require('./representation.js');
26
27  // connector modules
28  var course = require('./connectors/course.js');
29  var home = require('./connectors/home.js');
30  var schedule = require('./connectors/schedule.js');
31  var student = require('./connectors/student.js');
32  var teacher = require('./connectors/teacher.js');
33  var utils = require('./connectors/utils.js');
34
35  // routing rules
36  var reHome = new RegExp('^\/$','i');
37  var reCourse = new RegExp('^\/course\/.*','i');
```

```
71      // course
72      if(flg===false && reCourse.test(req.url)) {
73          flg = true;
74          doc = course(req, res, parts, root)
75      }
76
77      // schedule
78      if(flg===false && reSchedule.test(req.url)) {
79          flg = true;
80          doc = schedule(req, res, parts, root)
81      }
82
83      // student
84      if(flg===false && reStudent.test(req.url)) {
85          flg = true;
86          doc = student(req, res, parts, root);
87      }
```

```
62  function sendList(req, res) {
63      var rtn, doc;
64
65      rtn = component.course('list', root);
66      rtn.action = {};
67      rtn.action.template = listActions('course',root
68
69      doc = {action:{link:[]}};
70      doc.action.link = utils.pageActions(root);
71
72      doc.list = [];
73      doc.list.push(rtn);
74      rtn = doc;
75
76      return rtn;
77  }
```

```
89  exports.course = function(action, args1, args2) {
90      var object, rtn;
91
92      object = 'course';
93      rtn = null;
94
95      switch(action) {
96          case 'list':
97              rtn = loadList(storage(object, 'list'), object);
98              rtn = addEditing(rtn, object, args1);
99              break;
100         case 'read':
101             rtn = loadList(storage(object, 'item', args1), obj
102             rtn = addEditing(rtn, object, args1);
103             break;
```

```
86  function getItem(object, id) {
87      var item;
88
89      item = JSON.parse(fs.readFileSync(folder+object+'/'+id)
90      return item;
91  }
92
93  function addItem(object, item) {
94      item.id = makeId();
95      item.dateCreated = new Date();
96      fs.writeFileSync(folder+object+'/'+item.id, JSON.string
97      return getItem(object, item.id);
98  }
99
100 function updateItem(object, id, item) {
101     var current;
```

```
21      doc += '<root>';
22
23      // handle action element
24      if(object && object.action) {
25          doc += actionElement(object.action);
26      }
27
28      // handle link collection
29      if(object && object.links) {
30          doc += '<links>';
31          coll = object.links;
32          for(i=0, x=coll.length; i<x; i++) {
33              doc += linkElement(coll[i]);
34          }
35          doc += '</links>';
36      }
37
38      // handle lists
39      if(object && object.list) {
40          for(i=0,x=object.list.length; i<x;i++) {
41              doc += listElement(object.list[i]);
```

Each of these implmentation elements can be updated independently w/o breakage

Client Strategies

Most client apps are bound to URIs and the CRUD pattern

URI-Style Clients (CRUD)

- HTML SPA Container

```
<body>
  <h1>Tasks CRUD</h1>

  <!-- data goes here -->
  <ul id="list-data"></ul>

  <input id="add" type="button" value="Add" class=
  <input id="search" type="button" value="Search"
  <input id="list" type="button" value="List" clas

</body>
```

URI-Style Clients (CRUD)

- URIs, Objects, and Actions

```
var g = {};
g.msg = {};

g.addUrl = '/tasks/';
g.listUrl = '/tasks/';
g.searchUrl = '/tasks/search?text={@text}';
g.completeUrl = '/tasks/complete/';
```

URI-Style Clients (CRUD)

- URIs, Objects, and Actions

```
// fill in the list
elm = document.getElementById('list-data');
if(elm) {
  elm.innerHTML = '';
  for(i=0,x=g.msg.tasks.length;i<x;i++) {
    li = document.createElement('li');
    li.id = g.msg.tasks[i].id;
    li.title = 'click to delete';
    li.appendChild(document.createTextNode(g.msg.tasks[i].te
    li.onclick = completeItem;

    elm.appendChild(li);
  }
}
```

URI-Style Clients (CRUD)

- URIs, Objects, and Actions

```
// handle "search"
function searchList() {
  var text;

  text = prompt('Enter search:');
  if(text) {
    makeRequest(g.searchUrl.replace('{@text}',encodeURIComponent(tex
  }
}

// handle "add"
function addToList() {
  var text;

  text = prompt('Enter text:');
```

URI-Style Clients (CRUD)

- Composed HTML

```
<ul id="list-data">
  <li id="0" title="click to delete">this is some item</li>
  <li id="1" title="click to delete">this is another item</li>
  <li id="2" title="click to delete">this is one more item</li>
  <li id="3" title="click to delete">this is possibly an item</li>
</ul>
<input id="add" type="button" value="Add" class="button">
<input id="search" type="button" value="Search" class="button">
<input id="list" type="button" value="List" class="button">
```

URI-Style Clients (CRUD)

- JS Summary

```
g.addUrl = '/tasks/';
g.listUrl = '/tasks/';
g.searchUrl = '/tasks/search?text={@text}';
g.completeUrl = '/tasks/complete/';

// prime system
function init() {⟷}

// handle "list"
function refreshList() {⟷}

// handle "search"
function searchList() {⟷}

// handle "add"
function addToList() {⟷}

// handle "complete"
function completeItem() {⟷}

/* parse the returned document */
function showList() {⟷}

function initButtons() {⟷}
function clickButton() {⟷}

// handle network request/response
function makeRequest(href, context, body) {⟷}
function processResponse(ajax, context) {⟷}
```

A better approach is to bind to the message.

Hypermedia Client (REST)

- HTML FSM Container

```
<body>
  <h1>Tasks Hypermedia</h1>

  <!-- response data goes here -->
  <ul id="data"></ul>

  <!-- actions go here -->
  <div id="actions"></div>

</body>
<script src="tasks.js" type="text/ja
```

Hypermedia Client (REST)

- Media Types and Controls

```
/* parse the response */
function showResponse() {
  var elm, li, i, x;

  // fill in the list
  elm = document.getElementById('data');
  if(elm) {
    elm.innerHTML = '';
    for(i=0,x=g.msg.collection.length;i<x;i++) {
      li = document.createElement('li');
      li.appendChild(document.createTextNode(g.msg.collection[i]

      // see if we have an affordance here
      try {
        if(g.msg.collection[i].link.rel==='complete') {
          if(g.msg.collection[i].link.data) {
            li.setAttribute('data', g.msg.collection[i].link.dat
```

Hypermedia Client (REST)

- Media Types and Controls

```
// handle possible hypermedia controls
function showControls() {
  var elm, inp, i, x;

  // find and render controls
  elm = document.getElementById('actions');
  if(elm) {
    elm.innerHTML = '';
    for(i=0,x=g.msg.links.length;i<x;i++) {
      inp = document.createElement('input');
      inp.type = "button";
      inp.className = "button";
      inp.id = g.msg.links[i].rel;
      inp.setAttribute('method',g.msg.links[i].method);
```

Hypermedia Client (REST)

- Composed HTML

```
<h1>Tasks Hypermedia</h1>
<!-- response data goes here -->
▼<ul id="data">
  <li data="id" dvalue="0" id="complete" href="/tasks/complete/" method="post">this is some item</li>
  <li data="id" dvalue="1" id="complete" href="/tasks/complete/" method="post">this is another
  item</li>
  <li data="id" dvalue="2" id="complete" href="/tasks/complete/" method="post">this is one more
  item</li>
  <li data="id" dvalue="3" id="complete" href="/tasks/complete/" method="post">this is possibly an
  item</li>
</ul>
<!-- actions go here -->
▼<div id="actions">
  <input type="button" class="button" id="add" method="post" href="/tasks/" value="add" data="text">
  <input type="button" class="button" id="list" method="get" href="/tasks/" value="list">
  <input type="button" class="button" id="search" method="get" href="/tasks/search" value="search"
  data="text">
```

Hypermedia Client (REST)

- Summary JS

```
var thisPage = function() {

  var g = {};
  g.msg = {};
  g.listUrl = '/tasks/';

  // prime the system
  function init() {⟷}

  /* parse the response */
  function showResponse() {⟷}

  // handle possible hypermedia controls
  function showControls() {⟷}
  function clickButton() {⟷}

  // handle network request/response
  function makeRequest(href, context, body) {⟷}
  function processResponse(ajax, context) {⟷}

  var that = {};
  that.init = init;
  return that;
};
```

Message over Object

The Lessons of HTTP

You never get it right the first time...

92.06.11 RC, TBL

Summary of HTTP 0.9

The current version of HTTP can be summed up as follows:

- A browser only sends the command GET followed by a server/document identification optionally f
- A server replies to a GET by supplying a piece of ASCII text marked up in HTML, whereby just p
  default style for HTML is (a mistake??) free format.]
- the server holds no state w.r.t. the browser.
- if a document contains anywhere the ISINDEX tag then the browser takes this to mean that a valid
  appending the keywords given in the search panel.

Note that if responses from the server are encoded into HTML tags, then HTTP is very asymmetric: the c

Advantages

Network Working Group
Request for Comments: 1945
Category: Informational

T. Berners-Lee
MIT/LCS
R. Fielding
UC Irvine
H. Frystyk
MIT/LCS
May 1996

Hypertext Transfer Protocol -- HTTP/1.0

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

HTML 3.2 Reference Specification

W3C Recommendation *14-Jan-1997*

---

y W3C members and other interested parties and has been endorsed by the Director as a W3C Recomn
ial or cited as a normative reference from another document. W3C's role in making the Recommendatic
spread deployment. This enhances the functionality and interoperability of the Web.

4 The Content-Type Header Field

The purpose of the Content-Type field is to describe the data contained in the body fully
present the data to the user, or otherwise deal with the data in an appropriate manner.

(See Historical Note)

The Content-Type header field is used to specify the nature of the data in the body of an
information that may be required for certain types. After the type and subtype names, th
attribute/value notation. The set of meaningful parameters differs for the different types.
"charset" parameter by which the character set used in the body may be declared. Comm

In general, the top-level Content-Type is used to declare the general type of data, while t

10.7 Expires

The `Expires` entity-header field gives the date/time after wh
providers to suggest the volatility of the resource, or a date a
must not cache this entity beyond the date given. The presen
change or cease to exist at, before, or after that time. Howev
will change by a certain date should include an Expires head
by `HTTP-date` in Section 3.3.

```
Expires        = "Expires" ":" HTTP-date
```

14.19 ETag

The ETag response-header field provides the
entity tag for the requested variant. The
tags are described in sections 14.24, 14.2
MAY be used for comparison with other enti
(see section 13.3.3).

```
ETag = "ETag" ":" entity-tag
```

Examples:

```
ETag: "xyzzy"
```

10.2 Authorization

A user agent that wishes to authenticate itself with a server--us
so by including an `Authorization` request-header field with th
`credentials` containing the authentication information of the

```
Authorization  = "Authorization" ":" creden
```

HTTP access authentication is described in Section 11. If a req
`credentials` should be valid for all other requests within this

Responses to requests containing an `Authorization` field are

10.3 Content-Encoding

The `Content-Encoding` entity-header field is used as a modif
additional content coding has been applied to the resource, an
obtain the `media-type` referenced by the `Content-Type` head
document to be compressed without losing the identity of its u

```
Content-Encoding = "Content-Encoding" ":"
```

Content codings are defined in Section 3.5. An example of its

```
Content-Encoding: x-gzip
```

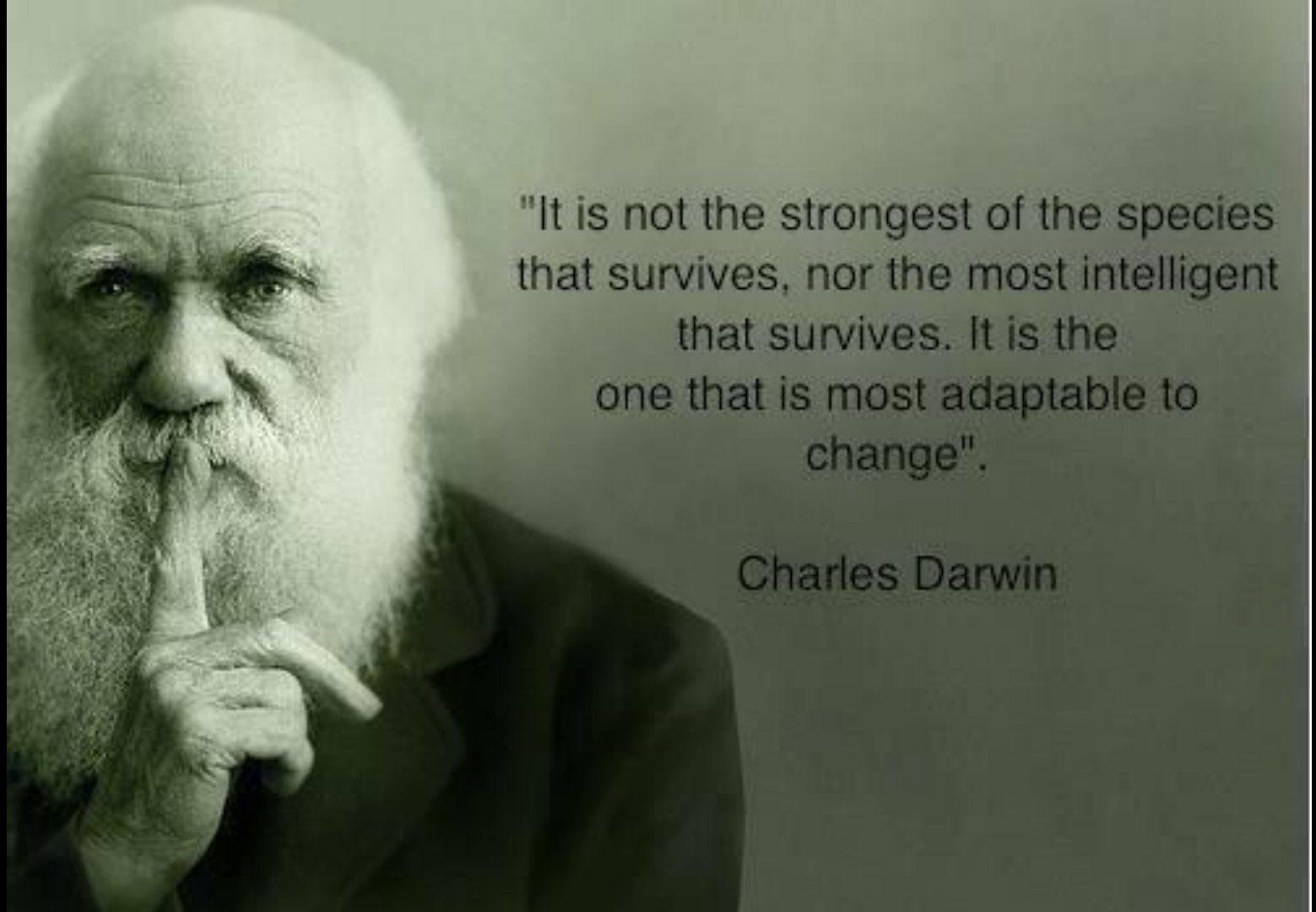
The `Content-Encoding` is a characteristic of the resource ide

So, lots of changes to the protocol over the last 15 years and...

It's all backward compatible!

"If you want a protocol to last a few decades, don't assume too much."
- Roy Fielding

So the real lesson in all this?


"It is not the strongest of the species
that survives, nor the most intelligent
that survives. It is the
one that is most adaptable to
change".
Charles Darwin

So...

Message Design...

- Use the extension pattern
- Keep structure in messages low
- Consider new message-based media types
- `form-urlencoded` is still a winner
- Commit to no "breaking changes"

Server Implementation...

- Keep component and connectors apart
- Use representors
- Make sure storage, biz, representors, and routers can change w/o breakage

Client Implementation...

- Bind to messages, not objects/actions
- Code defensively, don't assume
- Make sure requestor can convert messages into internal objects as needed

The lessons from HTTP

- You won't get it right the first time
- Build support for extensions into your work
- If you need to change it once, you might need to change it often.

CHANGE IS INEVITABLE.
CHANGE IS CONSTANT.

Designing for Reuse: Creating APIs for the Future

http://g.mamund.com/oscon2014-reuse

Mike Amundsen,
Layer 7 / CA
@mamund