Building API-Enabled Applications with ASP.NET MVC and Web API

Colin Bowern
Principal Consultant - ObjectSharp
@colinbowern
www.colinbowern.com

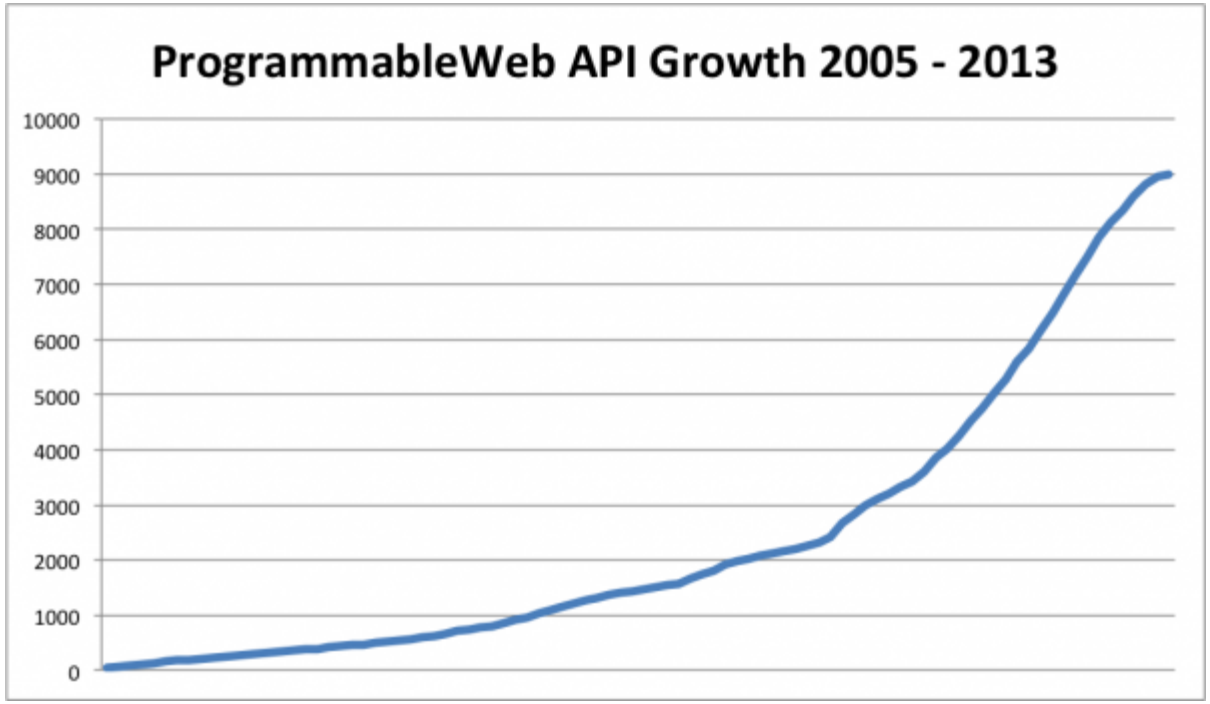
in this session...

- Understand why APIs can give you an edge over the competition
- Techniques to build your web application as the first and best consumer of your API
- Discuss security and caching engineering challenges that you will face when building out your API

apis are all the rage these days

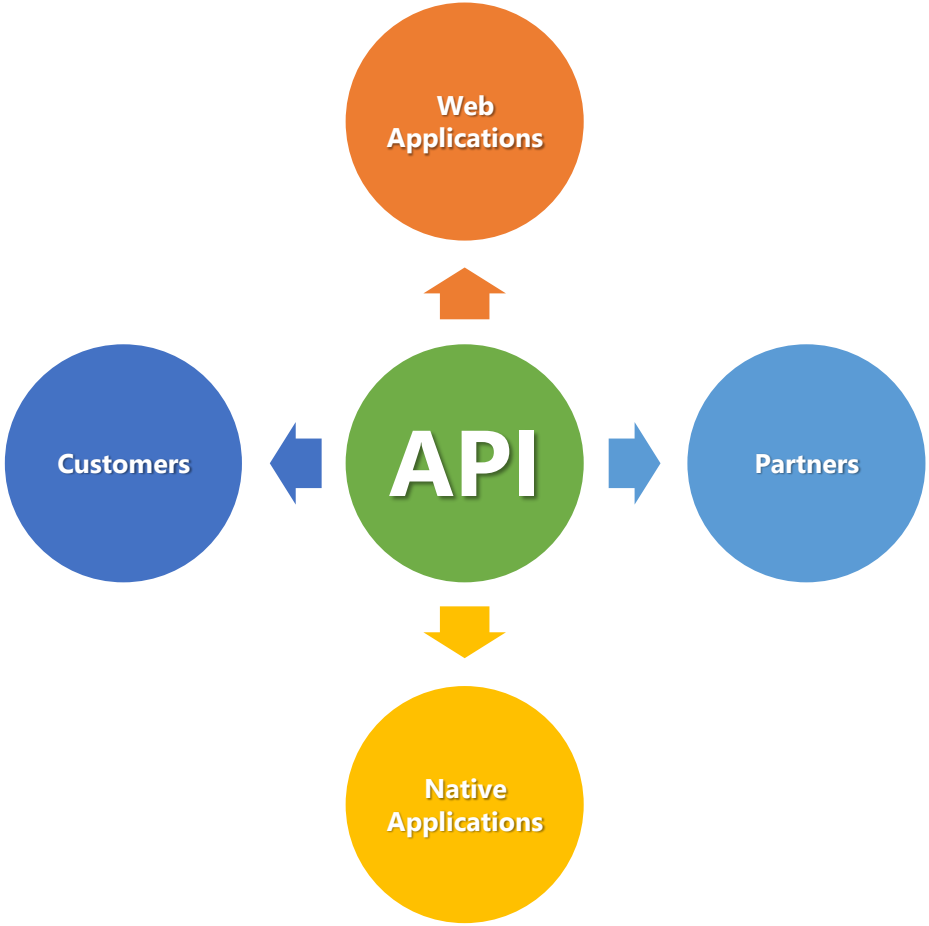
DevTeach

apis are everywhere



DevTeach

apis are the gateway to your business



DevTeach

creating a good api is hard

1. Provide a valuable service
2. Have a plan and a business model
3. Make it simple, flexible and easily adopted
4. It should be managed and measured
5. Provide great developer support

Want to know more? John Musser's OSCON 2012 talk covers these points in detail: colinb.me/GreatOpenAPI

signs you are doing it wrong

- Assume if you build it they will come
- Exposes raw underlying data model
- Complex or proprietary security implementation
- HTTP-based APIs that ignore HTTP semantics
- Poor error handling
- Poor documentation
- Unexpected and undocumented releases
- Inadequate support
- Expect an MVC framework “gives” you a great API

(Source: http://blog.programmableweb.com/2012/08/03/top-10-api-worst-practices)

before you get busy with your first api...

It's not enough to write tests for an API you develop; you have to write unit tests for code that uses your API. When you do, you learn first-hand the hurdles that your users will have to overcome when they try to test their code independently.

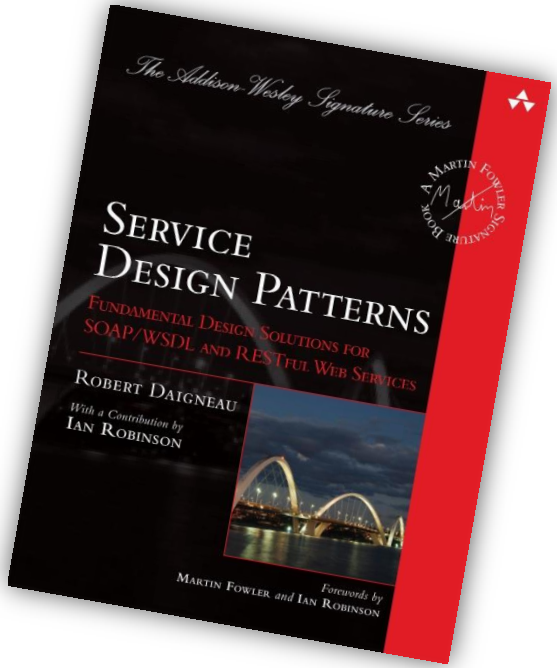
- Michael Feathers, 97 Things Every Programmer Should Know
colinb.me/GoldenRuleOfApiDesign

choosing an api style

- **RPC** – Request handled by specific procedure using a set of parameters
- **Message** – Service determines what procedure will fulfill request based on contents and context

We'll focus on this today

- **Resource** – Operations are performed on service resources (data entitles, business processes)



DevTeach

DEMO

HELLO API

planning your resource URLs

URL	Description
/api	API Entry Point
/api/(collection)	Collection of resource
/api/(collection)/(id)	Resource with specific ID
/api/(collection)/(id)/(sub-collection)	Sub-Collection inside Resource
/api/(collection)/(id)/(sub-collection)/(sub-id)	Sub-Collection Resource with specific ID inside Resource

Generally speaking:

- Rarely go beyond two collections deep
- Always provide absolute URL references

handling requests

Operation	Description	Safe?	Idempotent?
GET	Retrieve a resource	✓	✓
POST	Create a new resource	✗	✗
PUT	Update a resource with complete copy provided in request body	✗	✓
DELETE	Delete a resource	✗	✓
PATCH	Update a resource with diff of specific version (via If-Match) ! Format not standardized (JSON Patch draft vs. XML RFC 5261)	✗	✓
HEAD	Same GET without the content body	✓	✓
OPTIONS	Describes requirements for operations allowed on resource	✓	✓

do you patch or post to initiate process?

- How do you handle actions on a resource (e.g. initiating state change requests)?
  - Recall that REST ≠ CRUD
- Option 1: PATCH /Tasks/1 { Status: 'Complete' }
  - What happens if caller includes other changes?
- Option 2: POST /Tasks/1/Complete
  - Do the hypermedia thing and expose it in the GET response:

```
<task id="1">
    ...
    <link rel="MarkComplete" method="POST" href="/Tasks/1/Complete" />
</task>
```

returning useful status codes

Code	Meaning	Examples
1xx	Just wanted you to know…	100 Continue
2xx	Everything is fine	200 OK 202 Accepted
3xx	I think you want to look over there	301 Moved Permanently 302 Found 303 See Other 307 Temporary Redirect
4xx	You screwed up	400 Bad Request 401 Unauthorized 403 Forbidden 418 I'm a teapot
5xx	We screwed up	501 Not Implemented 504 Service Unavailable

side bar: redirection in a nutshell

Code	Subsequent Requests	Follow?
301 – Permanent	New URL	GET only
302 – Temporary	Old URL	GET only
303 – Next step in the process is over there	Old URL	All verbs
307 – Temporary	Old URL	GET only

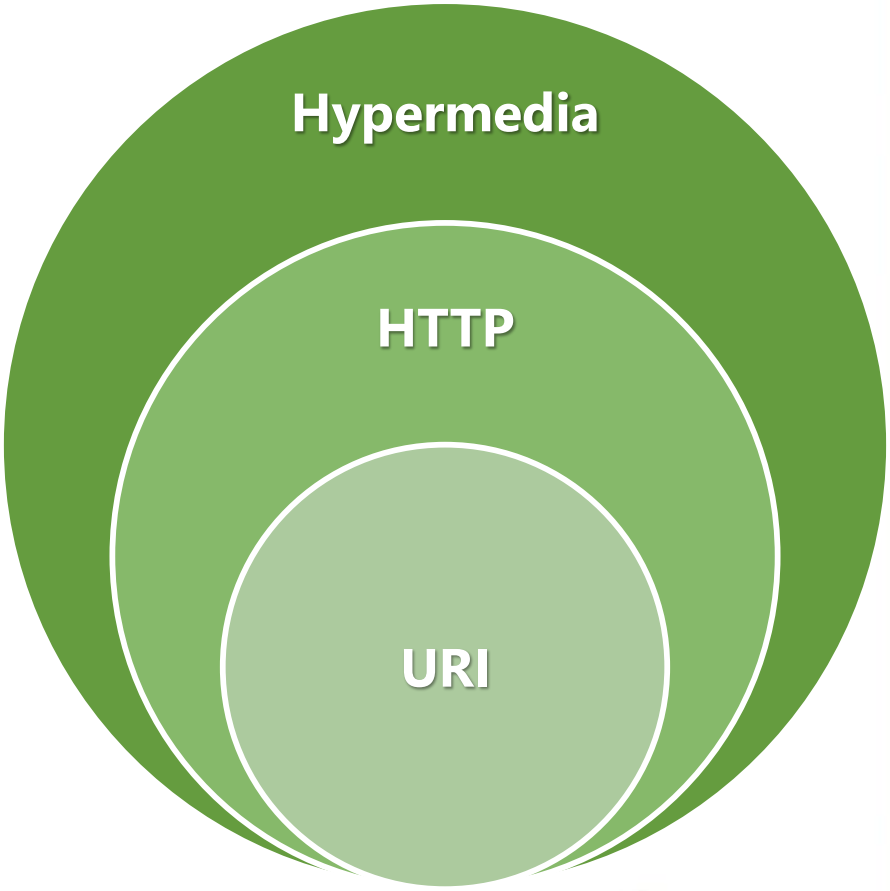
- Many user agents handle 302s the same was they do 303s
- 307 introduced to allow proper temporary redirects
- Avoid 302s in favour of 307

versioning

- **Version Header** –RFC 4229 identifies Version header to describe what caller is expecting
  - Proxies may strip out header values
- **Content Negotiation** – Accept header describes structure version (e.g. Accept: application/vnd.tasks.v1+json)
  - Could end up creating a lot of media types, unregistered ones discouraged
  - Have to handle ordering, wildcards, and quality factors
- **URL Versioning** – Bakes version into resource locator
  - Makes “permalinks” are impossible to store for other systems
  - How do you redirect outdated clients?
  - Omitting version = latest version?

are we RESTful yet?

- Leonard Richardson's model helps us think about the tenants of resource APIs
- Use it as reference point rather than dogma
- Martin Fowler explains: colinb.me/RESTModel
- Adding hyperlinks to Web API resources: colinb.me/HATEOSWebApi



DevTeach

thinking about access controls

DevTeach

securing the transport channel

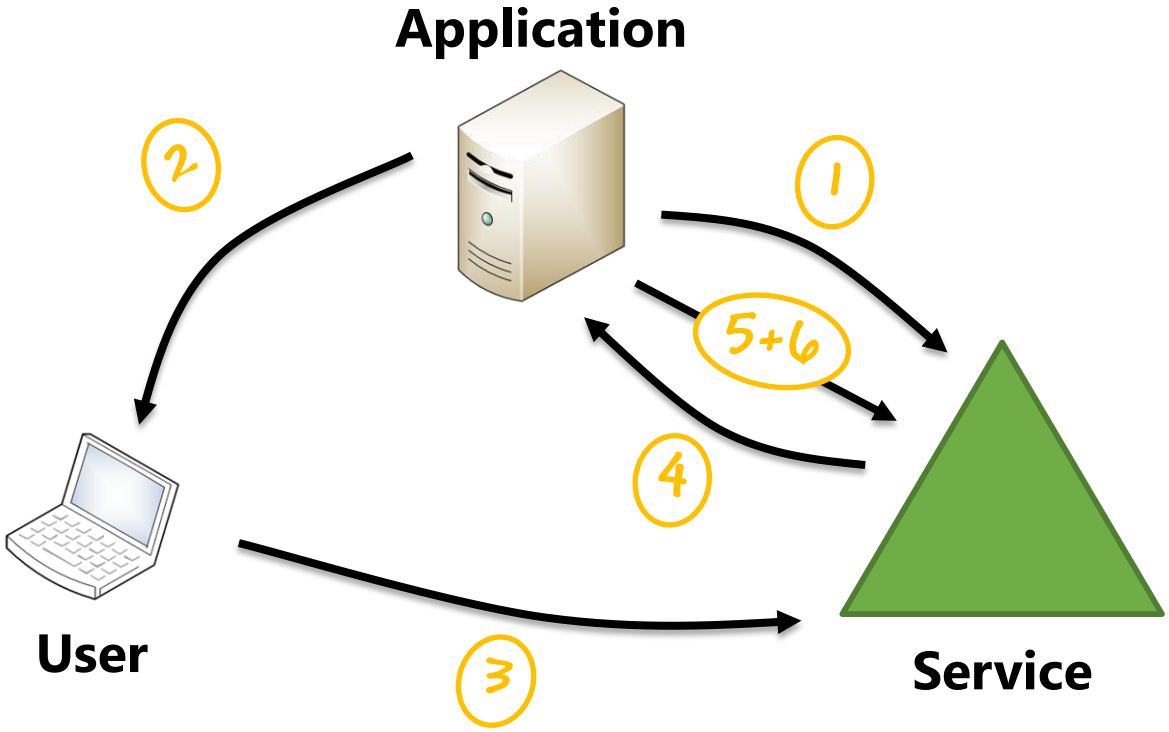
If you API needs authentication, then make TLS (aka SSL vNext) required.

Period.

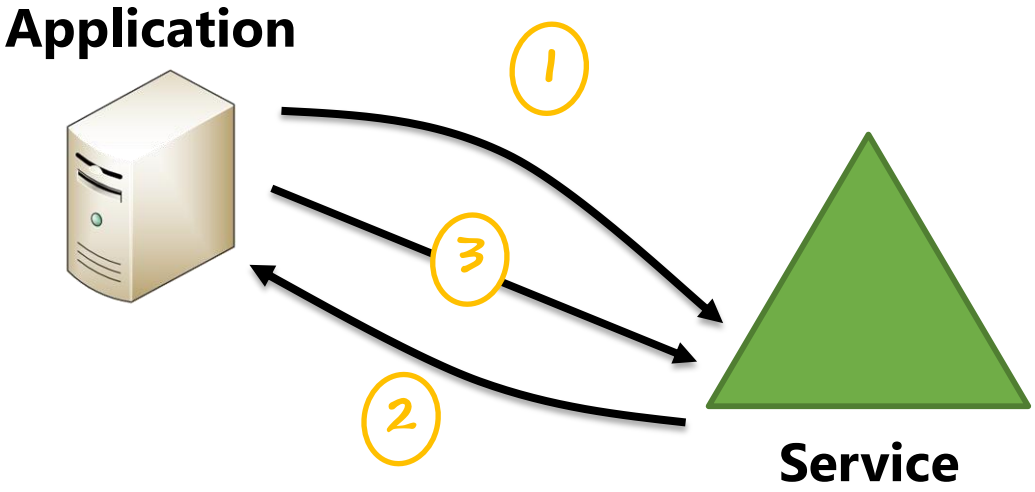
session security

- **Basic** – username and password
  - Base 64 encoded in the HTTP header – not encrypted!
- **API Key** – shared secret
  - Proprietary, simple, revocable
  - Can be stored in header, query string, or body
  - Works great for service-to-service interaction
  - HMAC-based keys discussed on StackOverview: colinb.me/HMACWebApi
- **OAuth** – delegation of service access for applications
  - The "valet key" of authentication - designed for user-app-service interaction
  - Watch for Open ID Connect to enrich OAuth 2.0 for additional scenarios
- Go deep on authentication : colinb.me/SecureWebApi

three legged auth: user-app-service



two legged auth: application-service



DEMO

SECURING YOUR API

the request/response highway

strategies for caching http responses

1. Do Nothing

2. Expiration
   - **Expires** – absolute date/time in response
   - **Max-Age** – number of seconds to wait between requests

3. Validation
   - **Last-Modified** – compare last state change date and only send full response if it has been modified since the absolute date
   - **ETag** – compare unique identifier for resource state and only send full resource if it does not match the current state

Better explained at colinb.me/HttpCaching

controlling agents along the way

- Cache-Control response header dictates how to handle future requests
  - **Public** – anyone can cache the response
  - **Private** – only the user agent can cache the response (i.e. response is specific to a single user)
  - **No-Cache** – always check in with origin server before assuming cached copy is valid
  - **No-Store** – don't store the response under any conditions
  - **Must-Revalidate** – check in with server once freshness expires
  - **Proxy-Revalidate** – similar to above, targeting proxy caches
- Remember - caches aren't required to obey headers. ☺

DEMO

ENCOURAGING RESPONSIBLE CACHING

API Tooling Ecosystem

API Building Blocks for the Open Enterprise

resources

- ASP.NET Web API
  - asp.net/web-api
- REST in Practice
  - restinpractice.com
- Service Design Patterns
  - servicedesignpatterns.com
- How to Design Good APIs
  - colinb.me/DesignGoodAPI
- API Best Practices
  - colinb.me/APIBestPractices
- Usable APIs in Practice
  - colinb.me/UsableAPIs
- The Web API Checklist
  - colinb.me/ApiChecklist