API-CRAFT CONFERENCE

Detroit, MI July 29-31, 2013

Conference Proceedings

/sessions

GET	/sessions	The sessions we create together

/sessions

GET	/sessions	The sessions we create together

{API Craft Conference}

Detroit 2013

Where the community of API designers and programmers leads the APIs of tomorrow

Why the API Craft community?

The very active community of API crafters possesses the knowledge and experience to set the course for API best practices. Diverse backgrounds and perspectives came together in this face-to-face meeting to hash out a plan for pushing forward the state of the art.

- What happens when theory collides with practice?
- Is hypermedia more than just hype?
- How do we maximize our impact as API trendsetters?
- What's the big picture in this connected world?

Google Group: https://groups.google.com/forum/#!forum/api-craft

Facebook Group: https://www.facebook.com/groups/apicraft

/sessions

GET	/sessions	The sessions we create together

API Business Models: Creating Value from APIs

Overall guiding principal:

Have a clear understanding of how I can create value overall, and how I could create value by integration with others?

How to realize value

Direct Value

- Ads, listing
- Data sales
- Revenue share/affiliate
- License fees
- Service fees
- Selling transactions versus contracts

Indirect Value

- Discoverability
- Improve product (e.g. search) by increasing use
- Market research
- Leverage long-tail integration
- Increase customer file/acquire customers
- Prevent unsanctioned use
- Scaling the organization

Roadblocks to realizing value

- The overwhelming roadblock is the lack of conventions for exchanging value.
- This results in companies being too internally focused, not seeing opportunities, and killing opportunities by not capturing value
- Lack of data about what is working. Yet, over time, these frictions will diminish.

/sessions

GET	/sessions	The sessions we create together

How to get started without complete information

- Have a defensible business model with numbers.
- Don’t hurt the existing business (without good reason).
- Think not what is the business model for the API but what is the API for the business model.
- Measure and adjust, be agile, and start small.
- Think hard about partner/developer segmentation and focus.
- “Time is a feature” so go to market sooner with less.
- Market it!
- Sell value to endorsers, design for developers, and (above all) know your customers!

/sessions

GET	/sessions	The sessions we create together

API Evolution

Convener

Joe Betz (@jpbetz)

Notes:

Attendees briefly described how they currently manage API evolution, what they plan to do in the future, and issues they are currently facing.

The takeaways from this exercise were

- Many use a version number in the URL (e.g.: http://domain/v1/) for backward incompatible versioning, which was brought into question (see below).
- Some are using or considering using Content Negotiation. For example, a client includes an accept header with "application/json;version=1" when requesting a representation/ version of a resource (see below).
- A few have successfully used hypermedia and Loose Coupling, to eliminate the need for API and resource level versioning, and are considering using Content Negotiation for representation level versioning.
- A significant number of participants have a mixed environment of RPC frameworks and REST, so they're looking for processes that will ease the migration from RPC to REST.
- Some API upgrades (SOAP was mentioned) are done in lockstep with client upgrades. The upgrade approach described was to bring the server and clients out of rotation (because they cannot accept traffic during the upgrade), upgrade all of them, and put them back in rotation. If any of the clients or the server has a defect, then they must all be rolled back. This versioning strategy has a negative affect development and development methodology, is unscalable, and would simply not work in an open ecosystem.

/sessions

GET	/sessions	The sessions we create together

Questions asked by Attendees during exercise

- How to migrate from non-REST to REST?
- What Migration/Deprecation Strategies work? How do I notify a client?
  - How to incentivize clients to upgrade to a later version. Some migrations are to a newer version of a REST API, others are migrations from non-REST to REST.
- How does one manage lifetime for a version so that it doesn't need to be maintained forever?
  - One major restriction is contracts.
  - Knowing the cost to serve is used in some ecosystems to have the client pay the bill to keep the old version running.

Major Discussion Topics

While a theme in the discussion was how to evolve with minimal versioning, the major levels of versioning discussed were:

- API level - E.g. http://domain/v1/ style. The idea is that the version applies to the entire API, including all resources and representations. This approach was brought into question, see below.
- Resource level - There was a lot of valuable feedback on how versioning at the resource level can be avoided. However, use cases and patterns where backward incompatible changes to resource directly (i.e. not to the representation but to that actual resource, such as what methods are supported or what query params a method takes) were not discussed.
- Representation - There was a general consensus that this is where most changes happen and where versioning is most needed. Content negation was discussed as a mechanism to enable.
- Behavioral - e.g. core business logic has changed, this impacts not only the syntax of the API but also the expected behavior. This was also not covered in details, there were suggestions that this can usually be done in a backward compatible way.
- Not related to API evolution but mentioned was instance data versioning (e.g. versions of a wiki page, often using optimistic concurrency control).

/sessions

GET	/sessions	The sessions we create together

Concerns with using/v1/ to scope all resources under a version

- For APIs with multiple resources, a client may want to selectively upgrade from v1 to v2. In the presence of hypermedia, this would required all links be to the same version path that the resource they were requested from is in. But to get around clients have been known to hack the links (e.g. regex s/v1/v2) so that they can use v2 of the resources they have upgraded to and v1 for those they have not yet upgraded to. This highlights how violating the uniform identification property of URIs (there are now two URI representing the resource, one with /v1/ and another with /v2/) can be problematic.
- It was pointed out that often if an incompatible idea of just a resource, not it's representation, needs to be made, this may justify creating a new resource rather than attempting to version the entire API. Also, version numbers in URIs are not necessarily needed for representation versioning, which can be handled using content negotiation.

Details of using Content Negotiation to version representations

- Only need a new version when introducing backward-incompatible changes, all backward compatible changes can be made to the same version
- Client sends Accept-Content header with "application/json;version=1", server responds with content in that version of representation OR a 406 Not Acceptable if the resource does not support that version
- Discussed how a server can respond when a version number is not provided. Options: (1) Respond with oldest supported version, (2) Respond with newest supported version (3) Respond with 406 Not Acceptable and short human readable body explaining that version # is required. #3 is considered the most conservative, with #1 also being fairly conservative if old versions are supported indefinitely but just as risky as #3 if server deprecates versions over time. #3 is the most risky as a client may be unable to interpret a new version of a resource.
- Discussed level of granularity of representation versions.

/sessions

GET	/sessions	The sessions we create together

Visibility into how clients use APIs

- Metrics on what APIs are being called by what clients
- Eg. what resources they call, what methods they use, even what fields they access (can be gathered if clients use projections) are a core asset when evolving APIs. It allows the provider to determine what clients may be impacted.

Rate of change is high during early development

- Eat your own dog food.
- Keep internal long enough for the API to stabilize and mature.
- Consider initially providing the API as a "closed beta". Only select clients may use API, API is allowed to go through backward incompatible change until it's out of beta. Client should not use in production until out of beta.

Outcomes

- Lockstep is obviously horrible! This is the "anti-pattern" for API evolution. A tolerance for backward compatible changes (e.g. adding optional fields, new resource methods) is a must.
- Detailed Metrics on how clients call your API is key in making informed API evolution decisions, as well as for generally staying in touch with your clients needs and expectations.
- Content Negotiation for representation level versioning (of backward incompatible changes) has promise, not all details have been worked out.
- Placing all resources under a /v1/ was brought into question, strong arguments were made that this approach, while common, both breaks uniform identification properties of URIs and can also cause some non-obvious problem down the road. See above.

Outstanding Issues

- Content Negotiation for representation level versioning (of backward incompatible changes) has promise, not all details have been worked out. References or write ups of working strategies needed.
- The largest practical concern is how to ease migrations from non-REST to REST APIs. This was not answered.

/sessions

GET	/sessions	The sessions we create together

API Smells

A brainstorming session about the common failure modes of API design and execution, and about the warning signs to watch out for when designing APIs. There were also recommendations for preventing some common ways to fail, noted below.

Convener

Petre Popescu (@steepyirl)

Notes:

We've identified the areas into which API Smells may fall.

- Principle of Least Surprise
- Query string bloat (information in query string that is better suited to URI or request body)
- Lack of consistent representation and naming
- Exposure of internal data model
- Same data represented in multiple places
- Excessive abstraction
- No dogfooding (didn't try API on self before exposing it to customers)
- Heavy use of documentation (indicates that API is not intuitive)
- False impression of atomicity of operations
- False impression of synchronicity
- protip: when accepting a request that starts an asynchronous process, return 202-Accepted (instead of 200-OK) and include cache headers in the response to return an estimate of when the process might complete
- Hearing "Hmm, interesting way of doing that!" from other developers reviewing the design
- Non-atomic updates (that require multiple requests to fully update the state of one resource)
- Multiple ways, multiple end points to update the same information

/sessions

GET	/sessions	The sessions we create together

Shinto, or: assuming impermanence

- API changing faster than the documentation
- Outdated documentation
- Designing for eternity

Put your Customer First (and know who your customer is)

- Lack of well-defined scope
- Excessive customization for 1st use case

Metrics

- Poor design for traceability (impossible to determine who is making which requests)
- protip: track usage of docs to see how customers use them; this could indicate areas that are difficult to understand

Granularity Balance

- Too granular (but how do we decide the proper level of granularity?)
- Same set of calls invoked repeatedly in the same order
- Huge payloads
- protip: optionally auto-follow hypermedia links (automatically incorporate result from following those links in response to initial request) in order to provide multiple levels of granularity for the same resource

Reusing Existing Solutions

- A lot of verbs in the URIs

Other API Smells that could not easily be classified:

- Lack of plan
- Lack of extensibility
- Lack of security
- Following trends blindly
- protip: redact data based on permissions (sometimes, this will involve a separate subsystem for redacting data that was already retrieved, before returning it to the client)

/sessions

GET	/sessions	The sessions we create together

APIs and FP: BFF's

Conveners

Onorio Catenacci (@OldDutchCap) and Chris Roland (@ChrisRoland)

Notes:

We started by discussing which functional languages all of us are working/playing with:

- F#
- Erlang
- Clojure
- Elixir

No state

- The idea of coding without state maps more nicely to the FP paradigm than the OO paradigm.
- Try to push the management of state to a higher level
- We need a good way to model the finite state machine. FP provides this good way.
- OO seems to encourage thinking in terms of hanging on to state

Idempotent

Resources as actors

- What is the actor model?
  - Carl Hewitt on Actors: http://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

/sessions

- Properties of actors
  - Actors are addressable (Resources are addressable)
  - Therefore seems like there should be a good mapping here.
  - Erlang processes are actors.
  - Actors carry some sort of internal state. Actor can be thought of as state machine.
  - Discussion of investigating mapping resources to actors. If anyone is interested in pursuing this, contact @KevinSwiber
- How can we insure that everything happens in the order it needs to happen in?
  - This is a basic issue with all async programming.

Custom routing patterns map nicely to FP pattern matching/destructuring forms.

Is Hypermedia a good fit for FP? Yes, we feel that it is a good match.

Discussion of pagination vs. side-effect free functions.

- When discussing hypermedia we need to be careful about whether we're discussing client side or server side
- How do we maintain state on the server side with FP?
- Easier to think of pagination via the mental model of thinking about pages in a book.
- Discussing using Reactive Extensions as a potential solution to pagination issue.

/sessions

GET	/sessions	The sessions we create together

APIs and Node

Convener

Brian Mulloy (@landlessness)

Notes:

Expectations

- How to go from play to deploying?
- How to sell node internally to the enterprise?
- How are projects going for others?
- What are the pains using node for APIs?

Analysis of Node for APIs

Pros

- Easy to build
- Smaller footprint than JVM (~12MB)
- Solid support on MS Azure
- Can move server work to JavaScript developers
- Async model is powerful
- Oracle supports it (node.jar)
- Chrome profiler

Cons

- Hard to find right libraries
- Still rev'ing quickly, hard to figure out impacts of change
- Async model is a challenge for some developers
- Debugging (especially in production) can be hard
- Google (v8 engine) is typically not an enterprise standard

/sessions

GET	/sessions	The sessions we create together

Popular Frameworks

- Express (3 people with experience building APIs)
- Restify (1)
- Argo (1)
- HAPI (0)
- API Axle (0)

Framework Analysis

Express

Pros

- Well documented
- Small code base
- Modular
- DIY approach
- Stable
- Strong community adoption
- Leaders are open about shortcomings
- Performance is good
- All-in-one web UI and web APIs
- Part of MEAN stack: Mango, Express, Angular, Node

Cons

- Focus is UI (not API)
- Response object is not a first class citizen
- No pipeline metaphor

Argo

Pros

- Built around pipeline metaphor
- NPM packages for:
- Rate limiting
- Compression
- OAuth
- Caching
- Tight knot community

/sessions

GET	/sessions	The sessions we create together

- Fast bug fixes
- Proxy is a first-class feature

Cons

- New
- Buggy still
- Small community
- No support for regular expression routes (conscious design tradeoff for performance)

Restify

Pros

- Rapid development
- No UI stuff to distract from API focus
- NPM packages:
- XML to JSON
- Rate Limiting
- OAuth
- dnode
- Routes have versions

Cons

- Smallish community
- No response pipeline

HAPI

Pros

- Created by Eran Hammer (OAuth dude, Walmart Labs)
- Good tooling

Cons

- Config vs Code
- Slow
- Code does not follow .js or node conventions

/sessions

GET	/sessions	The sessions we create together

API Axle

Pros

- Generates proxies quickly
- redis works out of the box
- Key management
- Proxy approach
- API management focus
- Commercial support

Cons

- Small community
- Not an API server (focus is proxy)

Additional Resources

The Plight of Pinocchio: JavaScript's quest to become a real language - Brandon Keepers

/sessions

GET	/sessions	The sessions we create together

APIs for Robots

Convener

Alan Languirand - @13protons

Notes:

Robots are in the physical space. They read voltages from sensors and turn them into data. Things are sampled via analog interfaces and output over serial ports.

We write code to go into the cloud and show up on web interfaces, design APIs and work with nonlinear systems.

How to make these two worlds talk to one another?

I've had some experience with arduino and piecing together a complex (feeling) stack that:

- Takes a gps, reads the data over analog pins
- Runs it through a processing library in C
- Sends it to the arduino's serial port...
- Which uses an xbee as a wireless serial bridge to my laptop
- Using node.js and a serial port monitor package to intercept the data and then
- Finally - log it in a remote database through a web API.

I had a hunch that there could be a better way. Arduino is awesome for reading sensor data and getting to a serial port, but the Rasberry, PI (http://www.raspberrypi.org/) is a fully capable linux computer for $25 that has analog I/O pins. You can run node on it to write in JS (or run python or many other languages), and cut out the complicated steps of parsing serial messages and let your code talk directly to both your sensors and the internet.

Buy API, make robots. Simple, right? Obviously there is much more to physical computing, but this one device will bring API libraries and physical sensors/actuators right into contact with one another and get the robot of your dreams going faster.

/sessions

GET	/sessions	The sessions we create together

Links

http://www.raspberrypi.org/

http://www.arduino.cc/

http://www.digikey.com/

http://www.adafruit.com/

https://www.sparkfun.com/

https://npmjs.org/package/serialport

/sessions

GET	/sessions	The sessions we create together

APIs What Should I Log?

Conveners

Onorio Catenacci (@OldDutchCap)

Use Cases For Logging

- Debugging
- Measure Usage
- Auditing
- Throttling
- Defects
- Security
- Authentication
- General Stats
- Legal

Two Main Flavors of Logging

Built-in vs. Custom

What Data Should Be Logged?

- IP Address
- Headers
- Time Stamp
- Status Code
- URL
- Stack trace if possible
- Code from which the message is being generated
- Module
- Line
- Be careful not to log too much--if you log information you don't need, you'll get overwhelmed by unimportant details
- Some data should not be logged; confidential e.g. access tokens.

/sessions

GET	/sessions	The sessions we create together

Tools For Logging

Googling for "Open Source Splunk" and you'll turn up lots of good tools

- l2met
- pingdom
- splunk
- fluentd
- papertrail
- librato

Tools We Build For Custom Logging

We want to build our logging tools so they can emit logs that can easily be machine parseable.

How Long Should Logs Be Kept Live?

- Probably 7 days (or less if you have high volumes of data)
- After that point, goes into archival storage

Additional Reading

log2viz: Logs as Data for Performance Visibility

/sessions

GET /sessions — The sessions we create together

Batch changes requests to resources

photo by @CHRISMETCALF

Convener

Jeff Damick @jdamick

Notes:

Problems

- Want to be able to update multiple resources atomically
- Want to be able to retrieve multiple resources in a single call

Explored some examples

- https://developers.facebook.com/docs/reference/api/batch/
- https://developers.google.com/storage/docs/json_api/v1/how-tos/batch
- http://docs.neo4j.org/chunked/milestone/rest-api-batch-ops.html
- http://docs.aws.amazon.com/cloudsearch/latest/developerguide/DocumentsBatch.JSON.html

Question: Do you need the operations to atomic?

- One example came out
  - Asynchronous
  - Return 202, ticket to get result of each operation
- Rest.li - supports batch operations
  - They tolerate individual failures within the batch
- Another suggestion: Remodel data so that it doesn't span multiple resources or virtual models over both.
- Why are people using batch operations?

/sessions

GET	/sessions	The sessions we create together

        - Performance considerations for mobile or internally to relieve pressure on backend services
    - Caching: can't use HTTP caches, need to user server-side caching techniques for parts of the response
    - Cassandra 2.0 supports a light transactional model, as well as DBs, so consumers want parity in their APIs with their backends.
    - Suggestion:
        - Rather than support atomicity, write the children & children of children first, then remove the old ones.

- Rest.li Approach
    - Not strictly RESTful
    - Minimal envelope
    - Serial / ordered operations
    - 1 set of headers (can be an issue)
    - Acts on 1 Resource
    - Purely an optimization
    - GET Example: /url?ids=1&ids=2&ids=3
    - Response contains the id mapped to entity response & response code
    - Batch support for PUT, POST and DELETE as well

Protocol level support? using SPDY

- No one (admitted) to running SPY on the services
- Possible solution, since operations can be pipelined, cached, and other protocol level benefits

Why else to use SPDY?

- Header compression
- Multiple resource retrieval
    - Another use came out:
- Migrating data across entire systems
    - Do I create children and then children of children in the new system or just move the whole thing in bulk?
    - Is bulk loading a special case?
    - Create the new data, hyperlink as appropriate
    - Bulk loading

/sessions

GET	/sessions	The sessions we create together

- Most agreed that bulk loading should be treated as a special case, and you can do it async
- Could use an orchestrator like service-mix or mule
- Make sure your resources have unique id's
  - Question: Can you just hit the next resource up a level?
- We were exposing metadata at that level, otherwise the data size can get unwiedly

Building and documenting APIs for discoverability

Goals

- Findability - Developers must be able to find the initial entry point to your API docs and your actual API
- Discoverability
  - Human - Developers should be able to easily learn how to use your API through documentation, adherence to standards, and good intuition
  - Programattic - Applications should be able to programmatically determine the capabilities of your API through introspection

These same concepts adhere to both internal company APIs as well as open public APIs.

Tools

- Swagger
- mashery/iodocs
- Apigee
- Hypermedia Forms
- kevinswiber/siren
- Collection+JSON - Hypermedia Type : Media Types
- JSON Schema and Hyper-Schema
- JSON-LD - JSON for Linking Data
- Rest.li

Techniques

- Be opinionated in your frameworks to push developers down the best path, but allow them to use only the portions they find most useful, if they have their own frameworks
- Build docs automatically, but augment with tutorials and authored examples
- Make docs tutorial-oriented
- Provide a big, obvious "API HERE!!!" entry point to your dev docs
- Be consistent in your query parameters, data formats, and paging techniques
- Be consistent in your functional docs
  - Question: Do APIs need JavaDoc-style standards for doc formats?
- Provide documentation about design intents, to help developers understand why certain design decisions were made

/sessions

GET	/sessions	The sessions we create together

- Consider standards such as OData or SODA
  - Provide URI templates
  - Keep your documentation fun and engaging (http://httpstatusdogs.com)
  - Make your internal developers use your own documentation. Don't answer emails!
  - Link out to standards where appropriate.
  - Make your API friendly to experimentation with cURL and simple tools
  - Allow external developers to submit pull requests with edits or tutorials
  - Use developer feedback!

/sessions

GET	/sessions	The sessions we create together

Consuming Media Types: Making Life Easier for API Clients

In this (sometimes lively) discussion, we talked about how much people use non-generic media types, i.e., other than application/json,application/xml). Turns out that not many people are using more specific or structured media types. Some of the problems (or perceived problems) that were brought up are as follows:

Problems

- In denial about distributed systems (see, for example, Mike Amundsen's blog entry about "Interoperability, not integration")
- Optimizing for API consumers to help them get their app up-and-running quickly, and to heck with longevity.
- No parser (library) available
  - HTML can be hard to parse, too (Mike Amundsen suggested using application/xhtml+xml to make this easier, with the plus that it looks like plain-old-XML that might avoid triggering fear).
- Some ambiguous, amorphous fear of HTML variants
- Cheaper to just create new resources: instead of versioning the media types, and/or using a media type that is more flexible in the face of change, many companies seem to decide that it's easier to just create a new resource or set of resources. This seems to apply to the whole "version the whole API with a version number somewhere" issue as well.
- Chicken-and-egg problem: API consumers won't use more specific media types (e.g., HAL, Collection+JSON, SIREN, etc.) if there's no compelling reason. Server/API developers won't use those media types for fear of lack of acceptance. Libraries to make consuming those APIs easier won't be developed (or refined) if it's not used by a wider audience.

/sessions

GET	/sessions	The sessions we create together

I asked people what media types they use:

- application/json
- application/xml
- text/csv
- (a line was drawn here to indicate that media types that follow are harder to consume)
- application/vnd.github.v3+json
- application/hal+[json|xml]
- application/collection+json
- application/vnd.siren+json
- application/atom+xml
- application/xhtml+xml

Benefits

We also listed benefits and/or ways to make the use of the "below the line" media types more approachable

- Loose coupling: to me, this is a key benefit, though it doesn't always seem to be recognized as a worthwhile benefit (see "in denial about distributed systems").
- Clients can be more dynamic: once the client is built around a more structured media type, specifically one that has links, the client can display options available and/or form elements completely based on what's return in the response body. This, to me, is the clear benefit of HATEOAS (aka Hypermedia APIs).

Making it more approachable

- Serve multiple media types, e.g., application/json and application/hal+json
  - It was argued that this could be a bad idea as it'd make it very easy for clients to simply use the "easier" media type and not be able to take full advantage of a type such as HAL.
- Create a media type proxy (e.g., translate generic application/json from a specific API provider to, say,application/vnd.siren+json) to promote the use of the conventions described by the media type.
- Providing libraries (SDKs) that make it easier to interact with
  - It was noted that the SDKs shouldn't make things too abstract, i.e., make sure that it's still clear to the user of the SDK that they're communicating over an

imperfect network. This was also brought up in other sessions where SDKs were discussed, so seems to be a common theme.

Action Items

Brian (@landlessness) was pushing for some way to promote the usage of these more structured (note: I'd love a better term for HAL, SIREN, Cj, etc.) media types. Mike Amundsen (@mamund) suggested that putting community effort behind a single "winner" might not be appropriate, since that one might not be the right answer for everyone (or even the right answer for anyone).

We left with the idea of people "living with" a particular media type for 30-days and then provide feedback to the community (and library providers, if you used one). Since I (Ted Young) have already been "living with" Cj, and using a specific Java library, I'll be providing feedback on that, and will try living with HAL so that I can compare and contrast the experience.

If you are also going to give a media type a good solid try, please comment below.

Describing APIs: API Blueprint

Describing your API with API Blueprint. A structured human readable interface definition language that understand your API. API Blueprint empowers the tools so you can build better APIs of tomorrow. From rendering an interactive documentation to complete CI lifecycle.

Learn about the concept and benefits of using API Blueprint. Discuss API Blueprint and IDLs in general. Contribute to API Blueprint format and brainstorm about the endless possibilities of tools built on top of API Blueprint.

Convener
Zdenek Nemec, @zdne

Notes:

- Idea is to have a DSL/structured file that will describe your API
- Question: is there a repo where people share their Blueprint schemas so we can see them?
  - Not right now. There are definitely plans to make it really open
  - Some people are already sharing their blueprints on GH
- There are no other ways to collaborate on API design
- Describing, designing, documenting APIs
  - Idea is to have a workflow integrating Blueprint with development
  - Push new revision of the spec, have tests run? Another session re API testing is scheduled for 3pm.
  - Curl trace parser:
    - curl --trace generates a trace file
    - Apiary wrote a tool to parse that file and output the request in Blueprint format
    - This output can then be used with Apiary so you get documentation, a stub service, etc.

/sessions

GET	/sessions	The sessions we create together

- Suggestion: document a famous/real-world API with Blueprint
  - They're doing this with App.net's API
- Markdown itself is not a very consistent format. Different flavors, different parsers, etc.
  - Apiary's Markdown is compatible with Github's

- Question: how to keep the actual API implementation in sync with the specification?
  - Tests: the API spec is a test.
  - There are some projects integrating Blueprint with the actual code (someone hooked it up with Rails)
  - Seems like there's not best practice, though. First design and review it, then implement
- There's also an idea to take the curl parser and move it to the server side (eg: a proxy that documents the requests using that same format)
- Question: is this sufficient to document actual APIs?
  - With JSON Schema developers can use a different format to specify data types, etc
  - There's also support for URI templates
- With statically typed languages it's obviously easier to test/be explicit about types. Dynamic languages need another layer/more explicitly check for it.
- Future development on Blueprint
  - Better support for links
  - More DRY
  - Multiple files
  - What about curl examples for the docs?
  - What about authorization?
- Human readable makes sense, but it's nice to have Swagger closing the gap
  - Concerns re spec drifting from actual implementation
    - Blueprint is about documenting and testing
    - Apiary built tools on top of Blueprint
    - Users are working on their own tooling
    - Blueprint is open source, users can share their tools/ideas for it
    - One could convert from Blueprint to Swagger, etc

/sessions

GET	/sessions	The sessions we create together

- This is just part of the documentation?
- Markdown helps, when you write API endpoints Blueprint will kick in, but arbitrary content is still supported, etc.
- Is it possible to embed Blueprint in a block of comment that goes together with the code?
    - No official tooling for this today. But it's certainly possible to scrap these out of the code and merge into a Blueprint (markdown) file.
    - This new Blueprint format just came out yesterday. Still a lot to be discussed, etc.

/sessions

GET	/sessions	The sessions we create together

Domain Driven Design and RESTful APIs

Convener

Ted M. Young (@jitterted)

Notes:

In Domain Drive Design, Bounded Contexts (BC) are good candidates for API service deploment as the stuff inside changes together. However, there will also be APIs that cross BCs perhaps they are more facade-like in that they hide the BCs behind a single API.

/sessions

GET /sessions — The sessions we create together

From Governance to Chaos

Convener

Carlos Eberhardt (@carloseberhardt)

Notes:

What is governance? Super-high-level, it makes sure IT is aligned with what business wants to do. Regulatory and industry requirements are also a part of governance.

Governance is what you do before you change a contract.

- Does the change in scarcity challenge some of the reasons for governance?
- The cost of investigating new technologies has gotten much better due to incremental changes. Big bang investments are not required any more.
- The end-costs may not be cheaper but the cost of experimentation is much cheaper now.
- Possibly analogy to departments getting minicomputers instead of needing mainframe access
- Governance is often one-dimensional, it's driven entirely by IT and without business involvement
- Does the emergence of API programs as a product provide for new opportunities for governance?
- Traditional governance is very project driven. Does that align with product strategy?
- Governance for API as a product needs to have marketing, biz development, etc., at the table alongside security, info protection, and so forth.
- Roles and responsibilities need to be clearly defined.
- What is the scope of governance? How do we keep it from becoming too big?
- Heavy governance on the asset side, the SOA side, can cripple your API program.
- Is it a serial process or can it be done in parallel?
- Can you split your governance model in similar to how Netflix has split their API model?
- Business governance becomes a factor if so. There's a technical governance and a business governance side.
- Risk: people in governance roles become fossilized.

/sessions

GET	/sessions	The sessions we create together

- Assuming Governance is not going to be eliminated, how can you make it more agile? ** Identifying short-paths with governance roles to navigate quickly ** Where does the buck stop? ** Make slow governance responsible for core systems ** Multiple governance layers to mimic other layers
- Lens of contract thinking matches up with the lens of "pace of change" layer thinking
- Risk of change is another view that needs to be included
- Onion model
- Contracts must be well-written

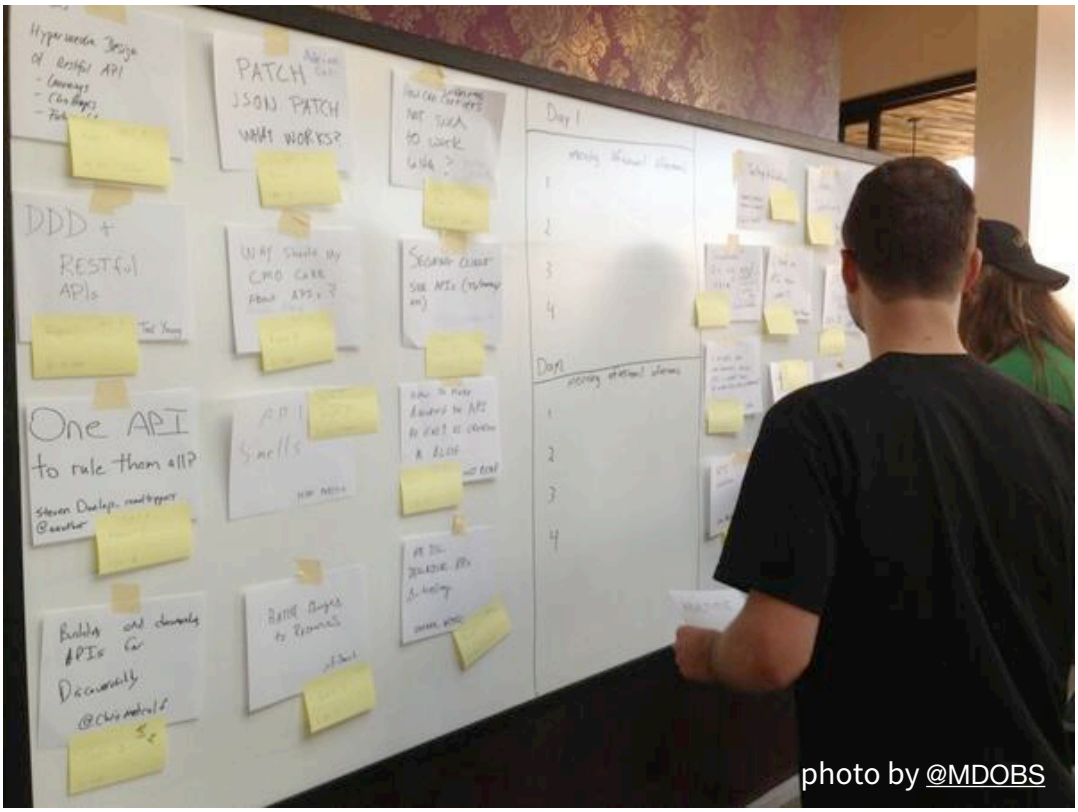


photo by @MDOBS

- Contract in API space mimics B2B with different contracts. Unless your API model has 1000s or orders of magnitudes more consumers, so contract becomes standardized.
- Could you govern the core systems and allow only "moderated" governance for internal developers to build anything they want?
- Seems yes, but how is it justified? Innovation!
- No! It's crazy. It's wild west.
- General consensus is that Carlos is crazy. ;-)
- Duplicating skins is easy, duplicating big stacks is expensive.

/sessions

GET	/sessions	The sessions we create together

- Tie in with costs. If costs are trending to zero, then can you have a more wild west style development process for APIs?
- Can you have multiple teams creating multiple APIs that do the same thing, allow the developer community to "vote up" the correct ones through usage?
- Can governance move to the lower level "platform" and allow ungoverned creation of APIs by project teams?
- If you let enough accidents happen, some of them will be happy accidents. Can you loosen governance in internal API development enough to allow happy accidents.
- Outcome: Depending on risk, there may be areas where governance can be loosened.

/sessions

GET	/sessions	The sessions we create together

Ground up API Design Workshop

In this workshop we designed an API from the ground-up, for a theoretical university course registration/student information system. The goal of the workshop was to determine the best approach for writing a useful API. Here's what we came up with:

1) Understand the business requirements

- User stories
- Use cases
- Visualization feedback
- ACL restrictions
- Existing data model

2) Extract nouns & verbs

- Nouns are your resources (e.g. students, courses, buildings)
- Verbs correspond to CRUD functionality & map directly to HTTP request methods (POST, GET, PUT, DELETE) note: keep your URIs short
  - /courses
  - /cources/{id}
  - /courses/{id}/students (this route represents a relationship between resources)

3) Handle exceptions using HTTP response codes

- 200 - General use, you may want to only use 200 errors and return an error message in JSON format
- 201 - Used in response to a POST, PUT or DELETE request to indicate successful execution of request
- 202 - Similar to 201, except indicates an asynchronous process is executing. Basically, this says "I successfully received your request. Give me some time to work on it."
- 301 - Redirect; used primarily to handle versioning. (E.G.: version 1 has /students and /professors, but version 2 only has/persons. A 301 redirect would send API requests to/persons)

/sessions

GET	/sessions	The sessions we create together

- 403 - This error code admits that the resource exists, but that the user is unauthorized to view it. Not the most secure - you're admitting the resource actually exists. (request: /students/12345 response: "You're not student 12345 so I'm not going to show you that information.")
- 404 - A bit more secure. Your system plays dumb. (request: /students/12345 response: "I have no idea what you're talking about, dude.")
- 503 - Service unavailable (for many reasons; maybe the user reached their request limit for the day/hour).

4) Handle querystring parameters

- Search: ?q=key:=value+key2:<value2...
- Returning only certain fields: ?fields=id,name...
- Pagination: ?offset=25&limit=5 OR ?page=3&records_per_page=10

5) Metrics & logging

Generally, you want to track:

- Who is accessing the API?
- How many requests is each user making?
- How many 404s are we returning?
- Do we need to use more 301s?
- Which endpoints are being used the most/least/not at all?

6) Authentication

OAuth is the best bet, but you can also choose to pass a non-oauth token through the Authorization header.

Additional notes

Use hypermedia to represent relationships:

- /courses/12345
  - {"id":"12345","title":"foo","students":"/courses/12345/students",...}

Version numbers can be passed either through HTTP headers or in the route (no consensus was reached in this regard).

/sessions

GET	/sessions	The sessions we create together

Hypermedia and the semantic web

- @mamund has a IDL proposal for hypermedia APIs and mapping to transports
- Hypermedia uses URNs and common Semantic types such as Schema.org and Dublin Core
- Hypermedia: Starting from the small use case and practical schema, working towards the big problem
- RDFa/Microformats: Great example of how simple formats can be powerful
- Semantic information as decorators
- Stop inventing new formats - no more snowflakes
- Hypermedia controls combined with semantics provide you a way to map data to actions
- "It's useless if you have a generic way to perform actions, but you can't describe what you're performing them on"
- You can apply multiple vocabularies to the same entities
- Everything doesn't have to be part of the same hierarchy, and that's OK. Agreement is the most important part

/sessions

GET	/sessions	The sessions we create together

Hypermedia API Design

Convener

Andreas Schmidt

Based off the design around the Nokia Places API

Notes:

Places API is an API for local discovery and information retrieval. Design goal was to build an API that provides entry points for various local discovery use cases. As local discovery is usually a multi step process, hypermedia links in the API responses correspond to the steps in such a use-case flow.

The API supports JSON representation only, using various json media types for different resource representations.

Hypermedia link representation

All json representations share a common representation for hypermedia links:

The minimal link object has two attributes:

- href : the URL to the linked resource
- type : the media type of the linked resource; either a RFC 2046 media type for links to external (web) resources, or for hyperlinks within the API a URN identifying the json media type of the linked resource (e.g. urn:nlp-types:place)

Additional link attributes:

- title : for hypermedia links intended to represent a step in the use case flow, this represents a title for that navigation element in a UI. The title is localized according to the apps language preferences (Accept-Language header); currently ~60 languages supported
- id : links to persistant resources contain a unique (within the resources media type) identifier which can be used for client side optimiziations (resource caching) or business logic (enabling app specific features where users have done something with those places within the app before)

/sessions

GET	/sessions	The sessions we create together

Question: Why id? Isn't the href the id?

- href is representing an actionable request, id identifies the resource
- href for the same resource might not be unique, as the URI also contains application (flow) state
- The id is not intended to be used for URI templating (although experience has shown that developers do it)
  - `method' : HTTP method for hypermedia links using other methods that GET
  - accept : for PUT/POST methods the URN of the supported media types expected in the request body
  - `labels' : for PUT/POST methods that expect to contain data entered by a user in some form, this dictionary contains localized labels / selection values for form elements.

Special links types and other things not going by the book:

- Some links for special link-relation types don't use the generic link object, but just a simple URL:
  - next : links to the next page within paginateable collections
  - icon : links to an iconic represenation for a given object
- Some URL are not generic hypermedia links, but developers see them as part of the content
  - view : a URL that can be used to share a resource between users (e.g. via email); when the user then opens that URL we will receive a UI with the best available representation of the resource for the given device platform
  - */src : the URL of binary media elements (images)
- Although the API encourages developers to use the appropriate HTTP headers, some client platforms only have limited access to those headers (e.g. JSONP). To support those clients, the API allows adding HTTP header also als query-string parameters to the hyperlinks; although kind of ugly, this has some nice side effects:
  - For demonstration purposes we can show every API behavior in any regular browser without need for special tools to manipulate HTTP headers (?accept=application/json for showing raw json representation in a browser)
  - Users can easily send us reports of any unexpected behaviour with a single URL, that is easy to embedd in any message container (IM, email, jira...) and immediatly actionable for the API developers.

/sessions

GET	/sessions	The sessions we create together

- Various resource support different representation flavors, that allow app developers to adjust the content (and its size) to their needs. Developers can chose those flavors by adding additional representation modifiers as query string parameters to the href of the link object.
  - Example: address is in html, but not all clients support html and need a plain text representaion of rich text elements
  - So clients can append a param tf (text format) to the object href and get the address rendered in plain text, for instance
  - Response is still a JSON, only the value of some attributes non contains plain text instead of html snippets

Question: Shouldn't that be conveyed in the accept header? Extension to the content type

- We have a hard time to convince developers to use different media types
- JSONP was a requirement.

Question: deviating from HATEOAS, from the developer perspective this seems more confusing? Links, params, href, etc.

- But it's easier to consume (easier to support a new param than parsing a bunch of links and picking the right one)
- It seems like a lot of people in the room are coming from the API server/backend perspective, presenter is had to take client-side demands into account

The API IS the primary documentation

- Without HATEOAS developers would get data, go to a documentation, and figure out how to build the next request
- With HATEOAS developers learn what are the available next actions
- Simple content negotiation can make the API when called from within a browser becomes a rich developer experience:
  - HTML serialization of json objects with extended functionality
  - Linking json media type URN to corresponding sections in the reference documentation
  - Simple form elements allow developers to learn/play with entry point request parameters and representation modifiers

/sessions

GET	/sessions	The sessions we create together

Iteration on API design

- In the beginning the API was more "academic", but application developers felt it was painful to consume
- Noticed developers enjoyed playing around with the HATEOAS APIs to learn, but once it was time to code they fell back to fixed object models/URIs
- Don't have the right tools available for consuming HATEOAS APIs?
- They had many sessions with ~20 different teams, they didn't care about links and semantic types!
  - There's no tooling for it
  - They also offer SDKs in c++, hard/expensive to parse semantic info

Question: What tooling is missing?

- Client side: support for something like JSON Schema link objects
- Jsonary is a showcase how this might work

Additional questions/discussions

- Inline JSON Schema in the response, or link to it?
  - Would be referenced from the content-type header, then clients can fetch and cache it
  - A lot of concerns related to the size of responses (again, re mobile clients)
- Question: What kind of semantic markup would you add to the JSON to make it more identifiable?
  - The rel attribute for instance is not present there
  - JSON format by itself doesn't handle links, there are extensions to the media type to support it
- Netflix had a big HATEOAS API, now going for device-specific APIs
- Clients need to understand the object types, eg what is a urn:nlp-types:place?
  - Introduced a nightmare: they told developers to always check the type, and ignore link objects when they didn't know it
  - Whenever clients saw a new type they freaked out
  - Is this because there was no content negotiation?
  - If they're having a hard time to convince clients about different types, pushing content negotiation would be really hard
  - One idea they had was to introduce noisy data into their API so developers are forced to respect different types/constraints
  - Like chaos monkey for APIs

/sessions

GET	/sessions	The sessions we create together

- href includes a binary context so they can do things like change the relevance of results
  - Like a google analytics cookie instead of the URI space
  - When developers hardcode URIs they bypass this context
  - They're considering making every request start through some specific public endpoints

- Why are developers not going for HATEOAS aware clients?
  - Pragmatism/urgency: they just want to get their stuff done
  - But that doesn't take into account the cost of maintaining this client over years
  - Some people are missing SOAP, better conventions to handle client specification and API changes
  - One idea to handle this is to take the Google approach, generate clients based off a JSON spec
- They extend the API by adding new link object types
  - Clients broke for instance when they introduced a "did you mean?" type to search results
  - Again, they tell clients to ignore types they don't know
- Some objects are dictionaries

/sessions

GET	/sessions	The sessions we create together

Like contacts (label: phone, value: "(123) 456789" / label: email, value: foo@bar.com / etc)

- Currently there's no way to fetch all type definitions from the API
  - There is a domain model and all documentation is generated off it, but they couldn't do that yet
  - Domain model is in Scala, gets rendered in different representations
- How do we classify this API in relation to HATEOAS?
  - This is a hybrid/there is tight coupling between concepts, etc.
  - How? All his resources are provided in the responses?
    - Because of ids and URL patterns. Client must fill template URIs with ids
  - Someone invested 4 months to get a proper client written to their HATEOAS API
- Filling ids in URI templates is harder than picking a URL from a list rendered by the API
  - But in practice there's still an effort into picking the right link
  - Say the API has 700 links for relevant information
- What's the example of the perfect HATEOAS API?
    - Hard question - there are two examples in Steve Klabnik's blog
    - Twilio
    - GitHub is close
    - But all of them with reservations
    - This is a cutting edge concept, they are being made at the moment

/sessions

GET	/sessions	The sessions we create together

Let's Talk About Our Relationship

Convener

[Matt Bishop (@MattBishopL3)]

Notes:

- Links between resources and different APIs will drive the Network Effect.
- Relationships are easy to identify--natural part of being alive.
- Relationships are stable over time.
- Location of relationships can change. My mom can move, but I still know what the name of our relationship is.
- Can we standardize the relationship graph? Not the data at each resource, but the names of the resources and their links to other resources?
- Semantic documents of prototype resources
- Risk here is stifling innovation. People won't participate if they cannot innovate on the resources.
- Maybe the names of the resources are really categories (like weather) that lead to specific providers (like weather.com).
- Search for resource by facet. I want weather in a zip code, or I want long term forecasts for Madagascar
- Dialects and L10N--can the resource provider support a resource dialect (certain sub-capabilities, like 10-day-forecast) in a particular Locale and present in a given language? This is about generating the right presentation data for the end consumer.
- Search needs to provide comparable facets to do apple-to-apple comparisons, while letting the providers surface their innovations in the search results.
- API needs to be browsable from there. HAL Browser is an example. Let the dev "poke around" in a sandbox would be great to see if they want to commit to binding to the resource provider. This means the media type is standardized to be browsable first.

/sessions

GET	/sessions	The sessions we create together

Amateur APIs

- To get to mass adoption of APIs that will drive the network effect, we need to make it easy for amateurs to build an API.
- Examples: A church, squash club, bowling league, bakery
- Today these amateurs go to blogger, or maybe weebly
- Do they know they need an API?
- Do they even know they produce data?
- RSS is similar to blogging, trick is capturing the event.
- Internet of Things can capture the events and generate the data. Bowling scoring desk, internet-connected oven. User manipulates Thing, which generates event, which populates a DB.
- @Include 'scores' in the website/blog page to draw the data into the presentation.
- Hypermedia Controls standardize the way an API can change state
  - Form
  - Selector (checkbox, radio buttons)
  - Date/Time picker
  - Authentication

/sessions

GET	/sessions	The sessions we create together

Offloading your API: a discussion about caching

Offloading your API

- Your mission is to maximize the caching of your API. If you choose to accept this mission you will be celebrated by your customers, industry application performance watchdogs (e.g. Gomez, Keynote) and praised by your Site / API Operations Manager.

Your Mission - Maximize Caching

- Avoid generating the object that compose your API responses repeatedly
- Allow these to age as long as possible
- Distribute objects to areas they are requested
- Enforce event driven object invalidation

Your Motivation - Customer Service

- Meet users expectation of a snappy response
  - User perception of performance is typically positive if the response below 250 ms
  - Conduct user sentiment analysis for keywords such as speed, slowest, frustrated, slow, etc
- Conserve resources for important tasks
  - Mission critical resources e.g. checkout, likes, follows or any other expensive transaction
  - Heavily repeated transaction conducive to caching is to be maximized.
- Enable consistent performance across http clients
  - Consider caching impacts on various http clients and proxies that can cache responses or objects.
  - Maximize caching on any level of your stack that supports. Carefully consider all opportunities from the browser to systems of record.
- Eliminate loss opportunities due to poor response times
  - Frustrated users typically equal dissatisfied API consumers.
  - Understand and measure performance impacts against conversion goals. Communicate this to business partners.

/sessions

GET	/sessions	The sessions we create together

For example:

- 0s - 1s response predicts a +%2 increase in conversion from average.
- 4s - 6s response indicates %1 drop in conversion from average.
- Greater than 6s response indicate a 3% drop in conversion from average.

Sentiment Analysis is key.

- Put your ears to groundswell and highlight user feedback on regarding performance.
- Use sentiment analysis to mine API consumer comments, forums and reviews.

Some Example Comments

- Customer Emails - "You definitely need to speed your mobile site up. It takes forever to see my friends updates and review my timeline."
- API Program Forums - "I think you may have the slowest API I've ever used. I seem to get a more timeout using your service compared to similar APIs."
- Customer Service Reps - "Customer reports that our site was painfully slow, forcing them use our competitors site."
- Site Reviews/Surveys - "25% users report that performance is tolerable. 10% report frustration, 3% report dissatisfaction"

Y**our Objective - Set Offload Goal for your APIs**

- Identify maximum offload during peak traffic
- Identify minimum offload during off-peak traffic
- Set a year-round average cache hit offload goal

A Strategy - Monitor Traffic Pathways

- Identify most popular resource traffic pathways
- Maximize caching for those resources
- Enable instant response on reverse paths (i.e. return to menu, back button, etc)

A Strategy - Forecast Potential Offload

- Adjust Time to Live to maximize offload
- Set different TTLs for each fragment of the response
- Determine TTL offload ratio e.g. 4h TTL = 70% offload

/sessions

GET	/sessions	The sessions we create together

A Strategy - Adjusting the Time to Live to "Pump up the Volume"

- Determine offload for most popular or mission critical resources
- Increase TTLs incrementally and observe and measure increase in offloaded traffic.

A Strategy - Consider Scheduled Invalidation

- Ensure scheduled content is up to date
- Enable predetermined future effective and expiration dates for relevant response fragments

/sessions

GET	/sessions	The sessions we create together

One API to Rule Them All?

Convener

Steven Dunlap (@aauthor)

Notes:

Flow of Thought

This might be able to be organized better (currently these points are chronological, not topical).

- Number of requests to the API can increase load time; motivation for splitting API could be to deliver more/less payload
- Versioning API could be seen as a form of API splitting
  - Versions do not need to be part of the resource path
  - Perhaps use headers or content-type
- **put part of what is being expected in header or query as part of the negotiation **
- Use more appropriate resources the get the exact data with more reasonable defaults (e.g. /area has a whole bunch of /pois instead of /pois defaulting to have nearby Pois)
- Stacking multiple API calls into one call; returning results in one large payload
- Netflix has a separate API for each device, but is that the norm?
- Having an internal API can lead to having "educated developers" in which API architects/developers aren't interested in keeping the API simple
- Code smell: data in API that can't be exposed to the public
- Only talk to client developer through documentation
  - Encourages better documentation and more though into API design instead of putting out fires
- SDKs
  - Thin wrapper vs. thick wrapper
  - A thick wrapper can disguise AJAX requests as simple getters/setters and include extra logic that makes the SDK work differently than an API
  - SDKs shouldn't hide a messy API; they should support and drive the API
  - Automatic SDK generation from API contract spec.
    - Smelly?
    - Don't have to maintain multiple code bases
    - Helps spur adoption of the API

/sessions

GET	/sessions	The sessions we create together

- Split documentation not implementation
  - Have the parts of the API that certain clients need exposed in their documentation instead of implementing splintered APIs for different use-cases
- "If you can do it in the app; you can do it in the API"
  - If the API is split, it will be very hard to keep that promise
- You can have more flexibility with a private API (make changes without worrying about breaking clients)
  - However, your flexible API could be your "beta" for the next public API
- State version vs view version
  - State version passed as content version HTTP header (reflect how the resource itself has changed)
  - View version passed in query or accepts header
  - Build things using your own API so that you know that it works
  - Push one API until become unfeasible; that that point you might have two different apps entirely
- Make sure you know your users use-case stories

/sessions

GET	/sessions	The sessions we create together

PATCH and partial updates

Folks and Context

- Adrian - change sparse data structure
- Steve - Investigating PATCH
- Brandur - Heroku
- Matt - JSON Patch for PHP
- Jeff - DNS
- Lukas - Tooling for APIs
- Rob - changing small items, or pieces of items from larger data existing data sets, while keeping an audit trail

JSON PATCH Overview

- Standardized in April
- Always an array of operations
- ex `{ "op": "add", "path": "/a/foo", "value" : any json
  - Path is a json pointer
- ops

  { "foo" : { "bar" : 42 } }

  - Test - ex. make sure something before proceeding
  - Add - only works if the object already exists
  - /foo/b succeeds, but /a/b fails, as the parent doesn't exist
  - /foo/bar will replace
  - Replace - differs from add where the entire path must exist
  - Move -
  - Copy -
  - Remove - for arrays, you need to know the index. ex. /foo/1
- Failure modes are not a part of the spec
- A little similar to XML patch or webdav ### Implementation notes
- About a month for Matt to write the pointer, then patch API
- Matt said he would open source the PHP lib in a non GPL license
- Add and remove are the only base ops, all others are derived from that
- Empty string in patch means the entire object

/sessions

GET	/sessions	The sessions we create together

- Watch out for pointer implementation wrt looking up parent (basename) to ensure replace is valid.
- Implement pointer spec first.
- Be careful with escaping, ex. resolve ~ last.
- Consider cucumber for language families
- Should be atomic (wrt the update)
- Use 202 like you do with normal REST, if the patch was accepted vs completed.
- It is possible to undo a patch
- If your HTTP response returns an "undo" patch ### Use cases
- Geo DNS: change territories as opposed to listing all in an update (more concise)
- Insurance language: hard to update what are otherwise very large strings (Emmanuel)
- Logging/Auditing:
- Mobile: GET a large structure is good for reads, but less so for PATCH
- Brandur: Should we use this for batch ops?
- Rob: watch out for business rules; how this object is composed on the backend
- Jeff: tried to do it, but the data size was too big.

Patch in the wild

Public APIs

- GitHub: PATCH, but not JSON patch
- Google Storage PATCH, but not JSON patch
- OpenStack

Public SDKs

- Java/Jackson: watch out LGPL
- Python: BSD
- Node/JavaScript: GPL
- Watch https://github.com/quickenloans for potential OSS
- Socialize your efforts if you make your own!

/sessions

GET	/sessions	The sessions we create together

Questions/Concerns

- How to deal with the relationship between front-end datastructure and backend resources Ex. when a subtree of the json can be updated atomically, but other pieces not A: (Matt) might lead to data model refactoring
- Rob: Why have a spec for this?
- Matt:
  - Helps to not have to know all the keys when doing an update
  - Helps with debugging or distributing change across nodes
- Brandur: Relationship between this spec and others?
  - Where url path and json path are related, this implies there are options for how PATCH is implied.
- Lukas/Jeff: queries are not stable. ex. list index or matching on values.
- Steve: point-cuts and patch. white list supported patch ops similar to HTTP 405

Interesting bits

- Arrays what happens when you add /foo/1?

  { "foo" : [42, 43] }

  inserts at position 1!

  { "foo" : [42, 44, 43] }

What if you want to affect the tail? Use index - ex. /foo/-

- Unknown keys are fine Ex. you could add a reason for the change

/sessions

GET	/sessions	The sessions we create together

Standards: Do we need them?

There are standards emerging that compensate for many of the perceived missing elements from previous API offerings -- JSON-LD, HAL, RDF, WADL, etc. All solve different problems.

The question: Do we need to reintroduce many of the complexities that these standards bring along? Are they problems they solve worth the impact to delivering good APIs?

Conveners

Chris Hatton

Todd Fredrich

Time

Day 2 (10:30-12:00)

Attendees

Carl Zetie

Agenda

- Contracts - WADL, JSON Schema
- Versioning - URL, Header, Content-type
- Linkability - ATOM, JSON-LD, HAL
- Content-Type Negotiation
- Authentication - OAuth, SAML, OpenID

Notes:

Preamble

Swing of the pendulum from SOAP => REST => {swinging back}?

/sessions

GET	/sessions	The sessions we create together

Contracts

- WADL - Been around the longest, lack of broad adoption
- JSON Schema - Validation, handwritten, more weakly typed, start minimally
- No tooling yet ... WSDL used to be auto-generated because SOAP had the tooling
- Depends on number of consumers and stability requirements
- Introspection could reduce need for contract
  - Interface is contract
- Value in contrac-first design (e.g. protocaol buffers)
- Doesn't capture call order (semantics) of contract. Definition might not be enough
- Tool generation and testing (after-the-fact ... code is always right)

Versioning

- Needed for backwards compatibility
- In header: mechanism for load-balancing
- In URL:
  - Negative effect on linkability
  - Good for versioning entire service/app
- Content-type negotiation:
  - Versions payload, still the same resource
  - Proposal for versioning in the content-type
- Default Version:
  - Default to latest stable (common)
  - Default to oldest (safest)
  - No default ... require a version from Day 1?
  - Provide deprecation warnings ... email to registered developer

/sessions

GET	/sessions	The sessions we create together

Linking

- Depends on service consumer library
- Graph w/nearest neighbors
- "self" -> identifies where obtained (in case of a save)
- w3c Linked Data spec w/ IBM at OASIS OSLC
- JSON-LD
  - Schema + Extensions + Linking
- HAL
- Siren
- Collections.js

Content-types

- More than 'application/json'
- Requires tools to support
- If domain-specific, types are more valuable
- Less-constrained domain -> less value from content types
- Use registered types for common elements

Authentication

- Despite problems, OAuth seems to be the best current option (broad general adoption)
- For SSO, SAML works well
  - Ping Federate good COTS offering
- HttpSignature from Joyent ?
- Auth.io ?

Mobile

- Most using bearer tokens
- Also Hashed user|password|salt as token

/sessions

GET	/sessions	The sessions we create together

Systems Interfaces

Convener

Noah Zoschke

Hangout

https://plus.google.com/hangouts/_/66d64f30e1f51869afa07fa8b589b648bd481a7a

Notes:

- Good Morning!
- Noah sharing about himself. Works for Heroku.
- "I'm very interested in APIs. I'm a systems guy."
- Lots of linux, and works on the application server
- Layers of APIs going deep down into the system
- There are lots of questions about how we should use APIs, but there are a lot of APIs established in the 60s that we use, and build on top of. They are just mostly operating system APIs.
- Let's try to get web interfaces to that stage.
- Nick is showing some slides concerning systems and APIs.
- User space is a metaphor for APIs in general. You can create an account and start using the userspace API. For example, the userspace for twilio is sending text messages.
- Matt from quicken loans speaking.
- "We have a mobile app. The users are anyone who is going through the process of getting a mortgage."
- Jeremy, "works on a product that does image sizing".
- Most of the front-end is designer people.
- Very different domains. There is some system that has control, and that imposes a lot of constraints on the user. But you still give the user a space to do something.
- It's this thing you build for people to play in but you can't get outside of it.
- The point is to make it easy for your users. You hide the complexity for the user. Make it a fun sandbox.
- Nesting doll of system interfaces. Let's talk about that.
- Giving example of the Sinatra app. Language virtual machine.

/sessions

GET	/sessions	The sessions we create together

- MRI Ruby VM - bytecode. Language VM is pretty well established. Something takes your code, parse it, and converts it into a set of valid bytes/tokens.
- The next layer in heroku. Heroku LXC container - exec. LXC is some of the most interesting technologies right now.
- (docker.io was also mentioned)
- LXC stands for Linux containerization.
- It's isolation of process tables, isolation of security purposes, and more.
- Dotcloud is a platform as a service.
- Within the next few years, everything will probably be running in containers. Really strong interfaces for isolating. Daemon can't go crazy and use all the resources.
- LXCs are really really fast.
- Off that tangent. So in heroku every single app is in one of these LXC containers.
- All this is running on ubuntu. Ubuntu is providing a file system for what your userspace could use.
- ubuntu does release management really well.
- As API developers we should look and figure out how this distributed company manages to build and release this operating system, give it to users and never break anything.
- The operating system maintainers have figured this out already.
- It's small tools that you can put together. Narrow but powerful interfaces. And let users put those together how they see fit.
- Decoupled
- Combinable
- One way to link to libraries
- Dependencies
- Thinking about the operating system influences.
- How do we approach that with our own APIs?
- Many endpoints? One way to link to libraries. Resources + Links
- Secret Admirer. Prices can attach a link to a line item.
- Encapsulation, where you are isolating.
- RBAC
- At heroku we use Amazon Web Services
- Xen - virtualization. Hypervisor. You are inside this userspace doing low level authorization - this process can get direct access to a CPU.
- Amazon Linux - somewhere there is a data center and they give you an API to start/stop their machines.

/sessions

GET	/sessions	The sessions we create together

- Paravirtual API. A CPU exposes it. Xen, or VMware uses it. It's a way for things down in the stack to get access to the CPU. Software virtualization and APIs it exposes are still important for software work.
- We want some alignment with how the APIs work under the hood. Let's try and align them to use the libraries that are powering them under the hood.
- Talked briefly about Erlang, and also about Node.
- CPU - one of the last levels. There is some piece of silicon inside with the x86 instruction set with crazy extensions to it for virtualization.
- The disruptor ring - example of knowing what is actually happening under the hood helps you improve speed.
- You have to in some cases know more than what the interface tells you to really do things very well.
- We are lucky we don't have to worry about all these interfaces. The complexity of some of the code under some of these interfaces is so complicated that you can't know it. The interface must be good.
- It can be hard to line these things up. We need to get as crazy aligned as these hardware developers. Things generally don't break for them. Our APIs break a lot.
- Every level is hiding some degree of ugliness. That's why we need abstractions. It hides complexity.
- Physics - the interface to rule them all.
- Example: Transistors - it's a physical thing, layered the right way with electrons and does some sort of gate.
- Responsibilities are clear on the systems' stuff, but not clear on the API stuff. There is no common 'open source' API to rule them all. There is a point where you don't want to share anymore because it hurts you against your competition.
- Should we have some sort of standardization on these APIs? There are good engineering practices that we aren't using in our software space.
- You have the public facing APIs of your own, but you also have a lot of 3rd parties. Thinking about the 3rd parties would help come up with some sort of standardization. Setting the standard.
- Layers of abstractions make it black box to black box to black box. The abstractions could always change too.
- Deep linking is a way to find what you need.
- All virtualization is not user friendly but it is operator friendly.
- Security: You have to have a boundary.

/sessions

GET	/sessions	The sessions we create together

- Expressiveness. That's how we get things done. Things need to eventually be expressive.
- That's the tradeoff for all these abstractions.
- We go into more group discussion now:
- We are talking about adding endpoints - evolving those over time. Evolving internal and public.
- We are thinking that discovery and documentation of those APIs is important. Some sort of directory service of your APIs.
- Service discovery in hypermedia might have some identifier. Maybe just linking.
- Maturity vs Discoverability. Noah, brought up how in linux though things don't change all the time. These internal APIs don't use a hypermedia approach. It's not discoverability. It's maturity.
- Large enough audience then you need to keep your API consistent. It shouldn't change.
- The lower you go on the stack the more well defined the interfaces are.
- The difference between going level is that the semantics have been defined and accepted.
- How about package managers? It is a way to manage versioning and keep things reliable.
- Tradeoff for package managers is the isolation route.
- "You have to be able to innovate". Microsoft never deprecates anything and then they can't innovate as fast because they are stuck with old stuff.
- In the server level you are empowered to go back and use an old version, but on the API level you really can't do that.
- Versioning is at the heart of this all. You need a way to deprecate.
- Containers also.
- The metaphor of container is really important and really apt. Shipping containers changed shipping.
- Example of hardware APIs breaking Hardware for a while was always about making the transistors smaller. Then it went to more cores on the machine. Now as software developers we have a whole new job managing those multiple cores.

/sessions

GET	/sessions	The sessions we create together

Takeaways:

- What does this mean for how we are developing stuff nowadays?
- You need a way to deprecate things - just like systems do. Because if you can deprecate then you can move forward and innovate.
- But you also need an ability to stay on something old - just like systems do. This allows people to continue working with old things.
- Versioning can solve that. This is how systems solve it. With versioning generally.
- And you move people onto the newer systems by giving them the new hotness. This is what ubuntu does as well. 10.04 is supported for like 5 years, it's stable, but you won't get some of the stuff in 12.x.

/sessions

GET	/sessions	The sessions we create together

What can we use from HTTP?

Convener

Mitch Oliver (@mitchvoliver)

Notes:

- PLEASE use error codes
- Perhaps a better question is, what should NOT be used? (no answer was proposed)
- Use cache control headers
- Refer to RFC 2616
- Custom "X-" headers
  - Not all clients get all headers
  - Indicate with special parameters or a PRAGMA
- Reusing error codes may violate the spec
- It was noted that the 30x range is chaotic
- Redefining errors will result in unexpected behavior
- Reusing errors in a sensible manner to add value is recommended
  - Keep in harmony with original meaning
  - Take time to understand the actual error that occurs
- For authentication, require that a separate resource be created
  - Incude above resource in AUTH header after resource is created
- Use of OPTIONS verb
  - Expose list of available resources
  - Hint about how to use resources
- WARNING headers
  - Indicate obsolescence
  - Indicate deprecation
- Could also use 302 to indicate deprecation
  - HOWEVER: must use versioning in URL and requires request is serviced in the same manner

/sessions

GET	/sessions	The sessions we create together

- Advantage of using HTTP is that tools are well spec'ed; proxies and frameworks understand HTTP already
- Advantage of simplified HTTP with custom error handling is that the learning curve is easier
- Question was posed: Are there any references for using HTTP effectively?
  - Perhaps a heat-map of what is actually used
  - Something that lists beginner, intermediate, and advanced features
- "Overloaded POST" can be used to support actions that do not map cleanly to a resource
  - Examples: search and submission
  - "Purchase order" resource has an "action" subresource
  - /purchase_order/action
- It seems that if your API is REST, you should use HTTP; if it isn't rest, take what you like and roll your own
- Are there any mainstream clients that misbehave wrt HTTP spec?
  - Proxy servers are troublesome
- ETags are easy for static content, however they are difficult for dynamic content
  - Clients can be flaky with ETags

Common Headers and Expected Results:

- Location header points to newly created resources
- Prevent cookie caching with NO CACHE SET-COOKIE
- Use 202 for asynchronous responses
  - Provide an indication of expected completion time, perhaps with RETRY AFTER

CONCLUSION!! Read the HTTP Spec.

/sessions

GET /sessions The sessions we create together

Why should my CMO love APIs?

Convener

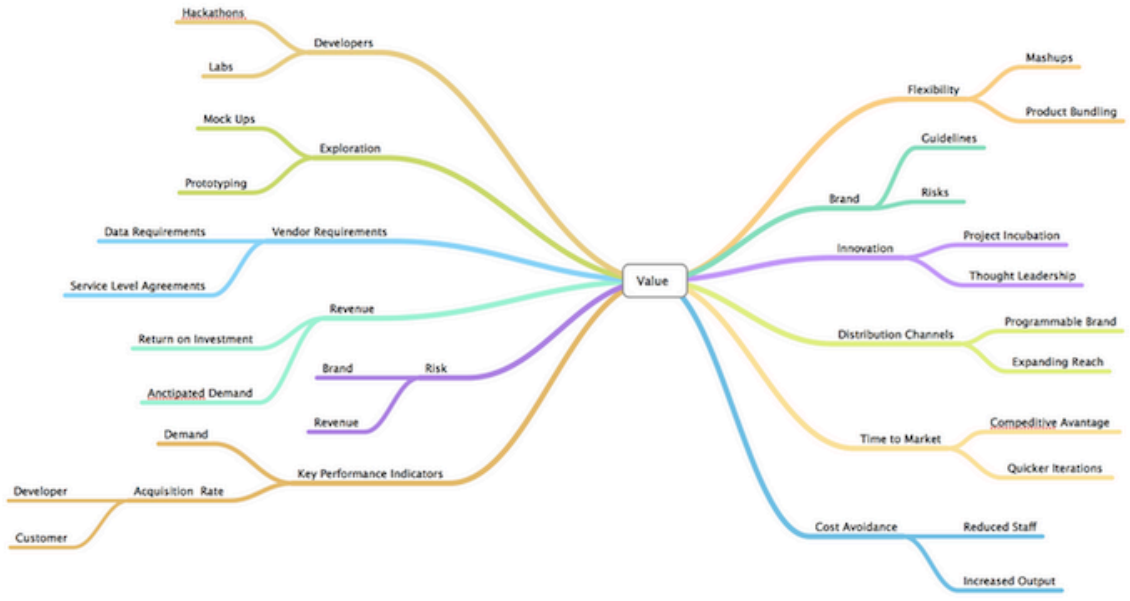
Kristopher Kleva (@klevland)

Notes:

APIs mean a lot of different things to different roles in the enterprise. This often exposes the question about who or why key business stakeholders should be on board with your API program. Considering this, I think of leadership at the c-level and how should ultimately have the most skin in the game. The CIO? - Yes, of course, this is a given.

But what about the CMO? What does he care about APIs? Why should he love them? This talk will prompt ideas and discussion of the best communications tools we have to evangelize to the Chief Marketing Officer.

Why should my CMO love APIs?



/sessions

GET	/sessions	The sessions we create together

APIs provide value

- Vendor Requirement - Vendors can more easily integrate to meet business service level and data requirements. Service levels can measures the performance of a system by defining goals and the percentage to which those goals should be achieved.
- Exploration -
- Developers -
- Flexibility -
- Brand - Extend your Brand by enforcing design and/or integration guidelines reducing risks incurred by unsanctioned scraping.
- Innovation - APIs can foster innovation by supporting business incubation programs designed to support the successful development of entrepreneurial companies through an array of business support resources and services, developed and orchestrated by incubator management and offered both in the incubator and through its network of contacts.
- Innovation - May position you as a thought leader recognizing you as an authority in a specialized field and whose expertise is sought and often rewarded.
- Distribution Channels - APIs can create new distribution channels. Allowing the Brand or Enterprise to become "Programmable" expanding the reach of your product and/or services.
- Time to Market - Get out into the market faster creating a competitive advantage by developing an attribute or combination of attributes that allows you outperform competitors.
- Cost Avoidance -
- Key Performance Indicators -
- Risk Management - Without an API, people will find unofficial points of integration that you have no control over, e.g. screen scraping. Better to have people "inside the tent"
- Revenue -
- Discovery - Once you offer an API you create the opportunity for 3rd parties to creatively bring value in unexpected ways.

/sessions

GET	/sessions	The sessions we create together

Key Points

1. They are the business of highlighting value. APIs provide value.
2. Because there is nothing worse than missing out on a trend.
3. APIs (in some form or another) have historically enabled multiple distribution channels.
4. Providing APIs can transition a traditional web site into a platform.
5. They are in the business attracting new customers or leads.
6. To bind or lock-in existing customers better providing service integration they are less likely to switch.
7. API consumers are likely different from the rest of your customer base, but have a powerful multiplying effect.
8. Because you don't want to get your lunch eaten by a competitor who did a fantastic job of reaching out to this segment and engaging them as customers.
9. Because the value of your product and/or service increases with the number of tools people can use with it.
10. The API is a means of building long-term customer relationships.
11. Puts you on the radar of technical people by sponsoring hackathons, writing technical papers, producing technical content.
12. Customers who integrated our APIs have stayed customers for longer on average and compared to non-integration customers.
13. Building a community of developers around our API will function as a sales multiplier in our market.
14. Creates a software halo around our core system that complements the out-of-the-box functionality.
15. Enables collaboration with integration partners (who rely on our API) in niche markets that we would not address by ourselves cost/benefit-wise.
16. Provides marketing with a new target audience - the integration developer to market the product and/or service and the complementing API.
17. Conveys a message to end customers that you are committed to meet their specific needs and are satisfied with a leading system and a smart integration.

/sessions

GET	/sessions	The sessions we create together

The impetus for creating APIs for an existing product can be entirely of a marketing nature (sometimes championed by engineering, but always from a marketing-as-perceived-by-engineering slant):

- This would be useful?
- People would use it and be happier?
- People will pay for this, or buy our core product in greater numbers?

Do's and Don'ts when pitching APIs:

1. Never tell anyone "they don't get it"... because they may understand the concepts using different nomenclature. Providing APIs are sometimes represented as "**Turning your site into a Platform**".
2. Don't debate the value or impact to the Brand with marketing. They are the SME in this area.
3. Don't overdose on the technical terminology.
4. Replace the API acronym with what it enables.
5. API must not be solely the domain of the developer evangelist, CMO needs to be in on this, but your dev evangelist / API support team should be their shepherd
6. Be aware of the customer segment actually consuming your APIs. But don't over do it
7. Be aware of the customer segment actually building (or just using) the applications on top of or integrating with your API.
8. Ideal use case is to provide a comprehensive product and/or service including an API, not just a pure API.

Use cases

- Dropbox and Twitter (before it changed tactics) grew because of third-party integrations
- Nike created Nike+ to prevent user for scraping content
- IBM Rational Software creates API into products to extend functionality

/sessions

GET	/sessions	The sessions we create together

Links

- http://adcontrarian.blogspot.com/
- http://prezi.com/rt07gxj02hf8/open-api-economy-ii/
- https://blog.apigee.com/detail/the_business_value_of_apis_pwc_apigee_webcast_video_slides
- http://www.google.com/?q=API%20Economy
- http://blog.programmableweb.com/2013/03/11/is-the-cmo-now-the-chief-api-officer/

/sessions

GET	/sessions	The sessions we create together

Post Conference Report

Discussion here: https://groups.google.com/forum/#!topic/api-craft/v7qpUPaIcUo

Attendee Report

Attendees came from the following cities

Ann Arbor	Phoenix
Austin	Prague
Berlin	Salt Lake City
Boston	Minneapolis
Chicago	New York
Cincinnati	Paris
Denver	Phoenix
Detroit	Prague
Freeport, Maine	Salt Lake City
Hong Kong	San Francisco
Los Angeles	Seattle
Minneapolis	Vancouver
New York	Washington, D.C.
Paris	Waterloo

104 Total Attendees
93 Paid Attendees
6 Organizers
5 Scholarship Attendees
30 Self-Organized Sessions

Financial Report

Total Revenue: $28,605
Ticket Sales: $16,275
Anonymous Sponsors: $12,330
Total Cost: $28,568
Food & Drinks: $15,280.37
Facility: $9,527.63
Swag: $2,724.14
Supplies: $536.84
EventBrite Fees: $499.41

/sessions

GET	/sessions	The sessions we create together

Retrospectives

What went well:

- Open space format worked surprisingly well
- Food and drink were good and there was plenty of it
- Venue was cool-ish (a few shortcomings below)
- The law of mobility kicked in on day 2 with good results
- Support team at the Madison Building was very helpful
- EventBrite was reliable
- Swag was well received

What did not work:

- The organizers were not as well organized as they would have liked for morning session on day 1
- Organizers did not clearly communicate the fluid nature of session times and session topics on day 1, which caused confusion
- There were not enough breakout areas
- The larger breakout areas were too noisy
- The organizers did not emphasize that sessions can and should end early if the topic has been covered
- Using the auditorium for evening news was the wrong space: broadcast mode instead of interactive mode

What to improve:

- Create a better circle for the morning planning session
- Due to lack of space, our circle collapsed into a lopsided blob
- Quieter breakout areas and more of them
- Encourage butterflies and bumblebees on day 1
- Bigger wall for the space-time matrix and the community bulletin board
- Communicate breakfast times in the event API and on the event website
- Ask for actionable outcomes for each session
- Create signs with API Craft values (represent thyself, not thycompany, etc) along with Open Space values
- Save money by ordering a little less food and drink