Firebase Security 2.0:

A Smörgåsbord of evolutionary and revolutionary changes for your consideration

Tom Larkworthy

Wed 9th, 2014

Outline

Critique of Existing Rules

- Syntax
- quotes and comma and fiddly
  - no comments or multi-line string in JSON
- Reuse
  - of model classes for denormalization
  - of common phrases in rules
- Semantics
  - write: false ... allows writes if child overrides
  - mix of concerns, access control, integrity constrains and schema
- Extendability
- Testing is hard work

Basic YAML

- YAML superset of JSON
- comments, unquoted keys, unquotes strings, multi-line strings, no commas
- indentation if not using JSON syntax

```
circle:
  position: [5, 6]
  color: red
```

YAML → JSON

- trivial transformation, pure JS implementation, js-yaml for node, CLI or Browser
- suggestions: display documentation in YAML and JSON?

```
{
  "circle": {
    "position": [
      5,
      6
    ],
    "color": "red"
  }
}
```

Advanced YAML

- block string literals, >
- merge keys operator, <<
- anchor and reference, & and *
- source maps (not available, but column and row tracked in js-yaml)

```
circle: &MYSHAPE
  position: [5, 6]
  color: >
    red &
    yellow

GoButton:
  shape:
    <<: *MYSHAPE
  isPressed: false
```

Advanced YAML

```
{
  "circle": {
    "position": [
      5,
      6
    ],
    "color": "red & yellow\n"
  },
  "GoButton": {
    "shape": {
      "position": [
        5,
        6
      ],
      "color": "red & yellow\n"
    },
    "isPressed": false
  }
}
```

YAML Summery

- well known (esp. python or ruby or GAE)
- solves all current syntax wrinkles
- easy to read
- ugly, but functional reuse
- Entire spec is massive but dedicated to types, many implementations do not support all of it
  - !!js/function
- Internally everything is JSON, but write the front end YAML (or write in JSON if you prefer).

Rule Variables

old names	data	newData	root	(newRoot)
temporal	before	after		
	prev	next		root_next

client/server	ref	request		
	db			
	server	client		

Rule Predicates

- Boolean functions: $x \mapsto \mathbb{B}$
- Encode the function declaration in the key
  - prevents users defining a procedural body

```
predicates:
  isLoggedIn():    auth.id !== null
  isUser(userid): auth.id == userid && isLoggedIn()
...
write: isUser($userid)
```

Child Dereferencing

```
#       |<---------- TOO LONG---------->|
- read: data.child("notifications").val()

#valid JS alternative
- read: data["notifications"].val()

#even denser for literals
- read: data.notifications.val()

#forget JS
- read: data->"notifications".val() #C++ DSL pointer style
```

Access Control

```
predicates:
  isUser(id): auth.id === id
schema:
  ...
access:
  - location: /users/$user/*
    read:  isUser($user)
    write: isUser($user)

  - location: /users/$user/public/*
    read:  isLoggedIn()
```

- Obvious place for triggers
- Would be nice to override predicates here too if possible

What is Schema?

- The static structure
- the integrity constraints
- JSON Schema, YAML, Finitio (was Q)
- **Not** the access permissions

JSON Schema (in YAML)

```
Person:
  type: object
  properties:
    name: {type: string}
    age:  {type: integer}
    sex:  {enum: [male, female]}
    employer: {$ref: "#/definitions/Company"}
  required: [name, sex]
  additionalProperties: false
```

Inheritance

```
defs:
  Shape:
    type: object
    properties:
      postion: {$ref: "#/defs/Point"}

  Polygon:
    allOf:
      - $ref: "#/defs/Shape"
      - points:
          type: array
          items: {$ref: "#/defs/Point"}
```

$vars

- wildcards are supported out-of-the-box through regex matching
- but named groups are verbose to bind the match, so we should facade patternProperties

```
users:
  type: object
  #regex match the allowed properties
  patternProperties:
    ^[\[\]$\.]: {$ref: "#/defs/User"}

  #But it loses the $variable name
  #so I suggest we:
  $userid: {$ref: "#/defs/User"}
```

Integrity Constraints

- ALL global constraints must evaluate to true to write, NO inheritance

```
predicates:
  inMaintenance(): root.system.maintenance.exists()

schema:
  definitions:
    Log:
      type: object
      $entry:
        type: object
        constraint: >
          !prev.exists() ||
          prev.val() === next.val() ||
          inMaintenance()
```

Extending: Meta Schema

level	schema	language
data	user-schema	JSON
user-schema	meta-schema	JSON
meta-schema	meta-schema	JSON

- What constitutes a valid user-schema is described by the meta-schema (which is also a valid schema!)
- The meta-schema states the allowed keywords, e.g. “type” or “properties”

JSON Schema's meta-schema

```
{
    "id": "http://json-schema.org/draft-04/schema#",
    "$schema": "http://json-schema.org/draft-04/schema#",
    "description": "Core schema meta-schema",
    "type": "object", ...
    "properties": {
        "maximum": {
                "type": "number"
        },
        "properties": {
                "type": "object",
                "additionalProperties": { "$ref": "#" },
                "default": {}
        }, ...
    }
}
```

ADT

- Meaningful keywords are created by extending the meta-schema (e.g. constraint)
- Keywords affect how security rules are generated (e.g. require)
- Existing keywords have to affect the rules anyway (e.g. additionProperties = false)
- The validity of a keyword is dependent on type (e.g. minimum)
- So bottom up traverse the schema tree, looking at the "type"

Queue

```
Message:
  required: [to, from, msg, id, before, after]
  properties:
    id:     {type: string}
    before: {type: string} #linked list
    after:  {type: string}

messages:
  type: Queue
  put_condition: isManager()
  get_condition: isSelf()
  properties:
    head:  {$ref: "#/definitions/Message"}
    tail:  {$ref: "#/definitions/Message"}
    $item: {$ref: "#/definitions/Message"}
```

Queue

```
Message:
  required: [to, from, msg, id, before, after]
  properties:
    id:     {type: string}
    before: {type: string} #linked list
    after:  {type: string}

messages:
  type: Queue
  put_condition: isManager()
  get_condition: isSelf()
  properties:
    head:  {$ref: "#/definitions/Message"}
    tail:  {$ref: "#/definitions/Message"}
    $item: {$ref: "#/definitions/Message"}
```

Meta-schema

- The same mechanism can be used to decide how “minimum” gets mapped into constraints
- As schema are URL references, you can import other peoples ADTs!

VM

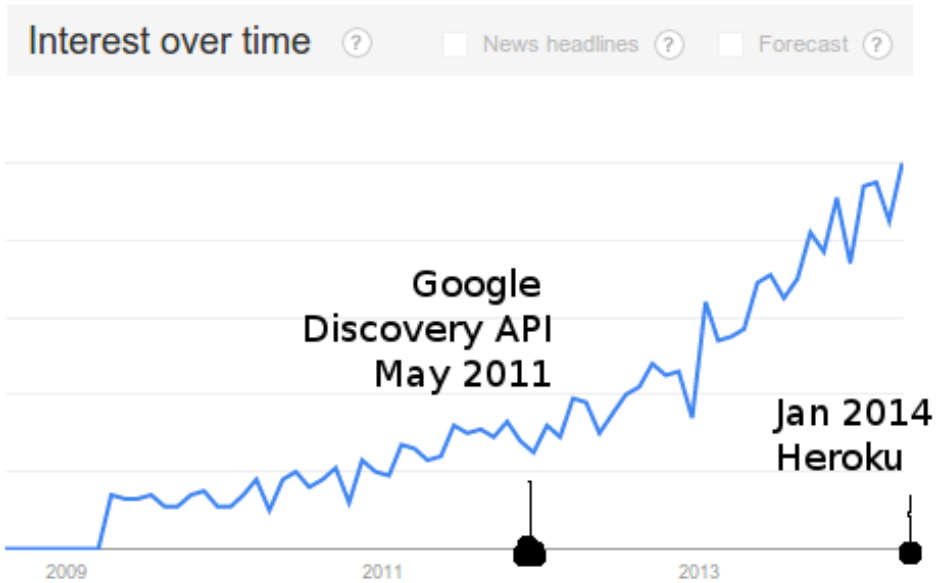
- What about a VM for a ADT to verify a user has run some byte code correctly
  - cross compile C++ to Firebase for serverside verification
  - missing >> and >>> bitwise operators & |
- explicitly represent the VM state (program counter, program, registers, memory)
- Client steps through bytecode like a debugger
  - server notifies client only if desyncs
  - client gets results before the server is aware!
  - server can be sure client has run the correct code!

Initial Firebase

```
predicates:
  isLoggedIn(): auth.id != null

schema:
  definitions: {}

  type: object
  properties: {}

access:
  - location: /*
    read: true
    write: true #&& isLoggedIn()
```

- would be nice to one click to a basic template, e.g. federated users

JSON Schema adoption

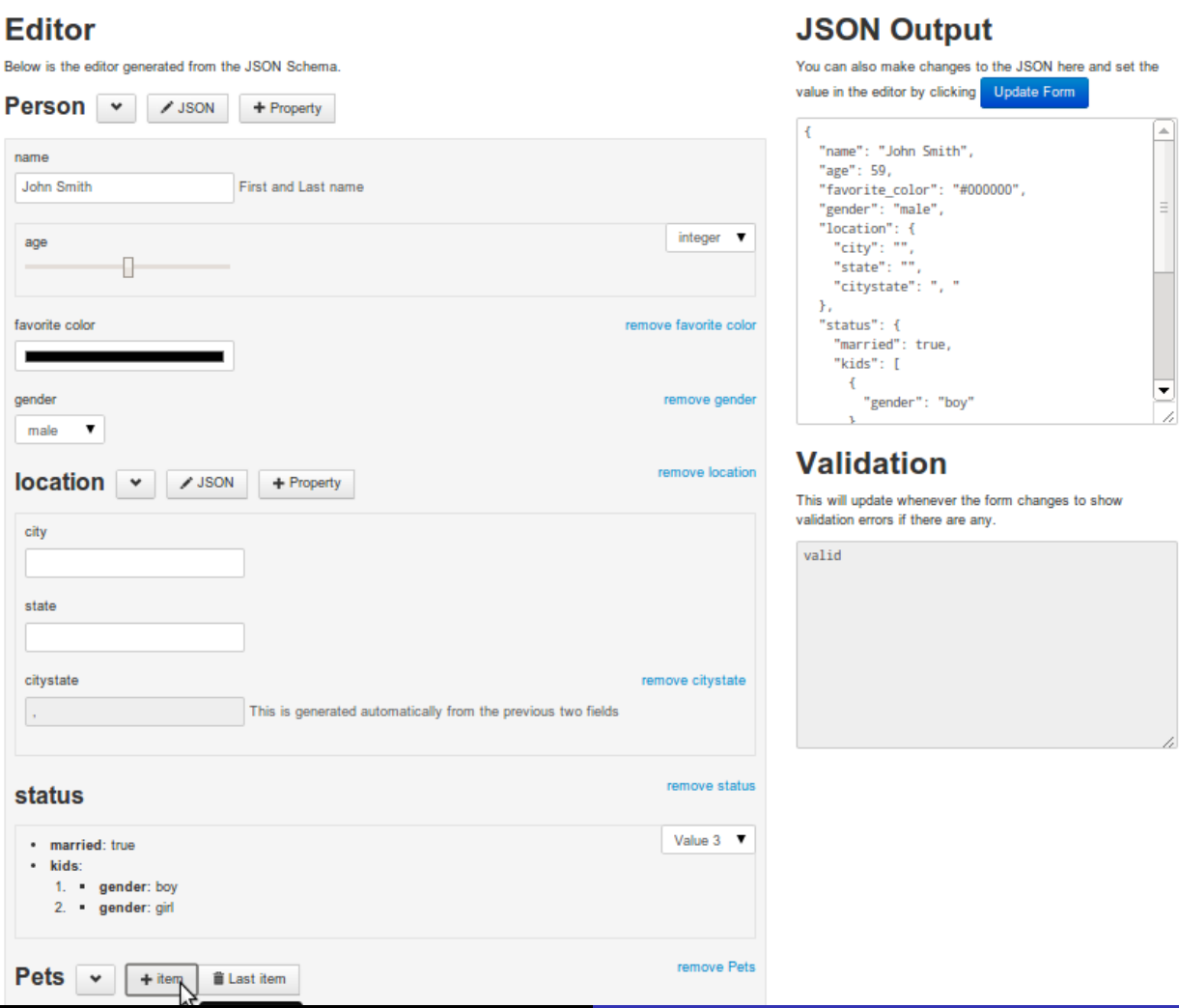


- IETF draft v4, v5 fixes unioned meta-schema with control flow statements. v5 is last draft
- v1-3 written by different people than v4 onwards

Tooling

- existing documentation on how to write JSON schema
- Validators in every language
- Hyper-schema describes the REST interface (which contains schema)
- Heroku clients (ruby, scala, node) dynamically generate clients from hyper-schema
- static code generators not mature
- UI generation from Hyper-schema
- documentation generation
- *interoperability*

JSON-editor


Editor
Below is the editor generated from the JSON Schema.
Person
JSON
+ Property
name
John Smith
First and Last name
age
integer
favorite color
remove favorite color
gender
remove gender
male
location
JSON
+ Property
remove location
city
state
citystate
remove citystate
This is generated automatically from the previous two fields
status
remove status
Value 3
married: true
kids:
gender: boy
gender: girl
Pets
+ item
Last item
remove Pets
JSON Output
You can also make changes to the JSON here and set the value in the editor by clicking
Update Form
{
"name": "John Smith",
"age": 59,
"favorite_color": "#000000",
"gender": "male",
"location": {
"city": "",
"state": "",
"citystate": ", "
},
"status": {
"married": true,
"kids": [
{
"gender": "boy"
Validation
This will update whenever the form changes to show validation errors if there are any.
valid

JSON Schema Good bits

- $ref (& and * in YAML)
- required
- additionProperties = false
- enum
- allOf (merge key in YAML)
- properties
- type

Testing

- development of schema/rules is error prone
- inline tests can catch semantic errors at "compile" time

```
Room:
  properties:
    createdByUserId:
      constraint: next.val() == auth.id
    numUsers: {type: number}
    type: {enum: [public, official, private]}

  required: [id, createdByUserId, numUsers, type]
  examples:
    - {createdByUserId: j43, numUsers: 10, type: private}
  nonexamples:
    - {createdByUserId: j43, numUsers: s, type: private}
```

Verification

- Dynamic behavior of database even harder to predict
- Computational Tree Logic (CTL) is a formal language for describing temporal evolution of logical systems
  - **A** means all paths
  - **E** means there exists a path
  - **G** global modifier (always true)
  - **F** finally modifier (the path leads to ...)
- SPEC AG EF users.fred.state = IDLE
- SPEC !EF (users.john.item = gold & users.fred.item = gold)

Verification

```
specification:
  - initial:
      users:
        fred: {item: gold}
        bill: {item: null}
        greg: {item: null}
    substutions:
      $user: [fred, bill, greg]
    specs:
      - >
        !EF (root.users.fred.item.val() != null &&
        users.fred.val() = null)
```

Suggestions

domain	suggestion	implications	hard?	USP?	TAM
language	use YAML	convertion step	1	2	100%
	support YAML/JSON	website bifocal	2	1	60%
		source maps	2	2	100%
security expr.	change newData		1	1	100%
	predicates	parsing key	2	2	80%
		scoping	3	1	50%
	dereferencing	API clashes	1	2	90%

Suggestions

domain	suggestion	implications	hard?	USP?	TAM
schema	support v4	expr. generation	5	3	90%
	meta-schema	extendability	5	4	40%
		VM	9	10	3%
	ACL	expr. generation	4	2	100%
		triggers	1	2	60%
	hyper schema serving	hosting integration	3	4	40%
		UI generation	3	4	50%
Testing	inline (non) examples	example	1	3	40%
	CTL	serverside processing	10	10	5%
		new billing	6	1	5%