tubi

Building next-gen APIs

What do good APIs look like?

- Github API?
- Google Cloud APIs?
- Amazon AWS API?

Traits of Good APIs (for users)

- Easy to learn and intuitive to use
  - **full-fledged and good quality** docs / playground
- Hard to misuse
  - type safety and proper error responses
- Powerful enough to drive business requirements
  - flexible, and performant
- Easy to evolve as the products grow
  - backward compatibility
- Opinionated
  - don't make me think

Traits of Good APIs (for devs)

- Easy to read and maintain existing code
- Easy to write new APIs / extend existing APIs
- Easy to generate client SDKs
  - less complains from client teams
- Great coverage of tests
  - more confidence

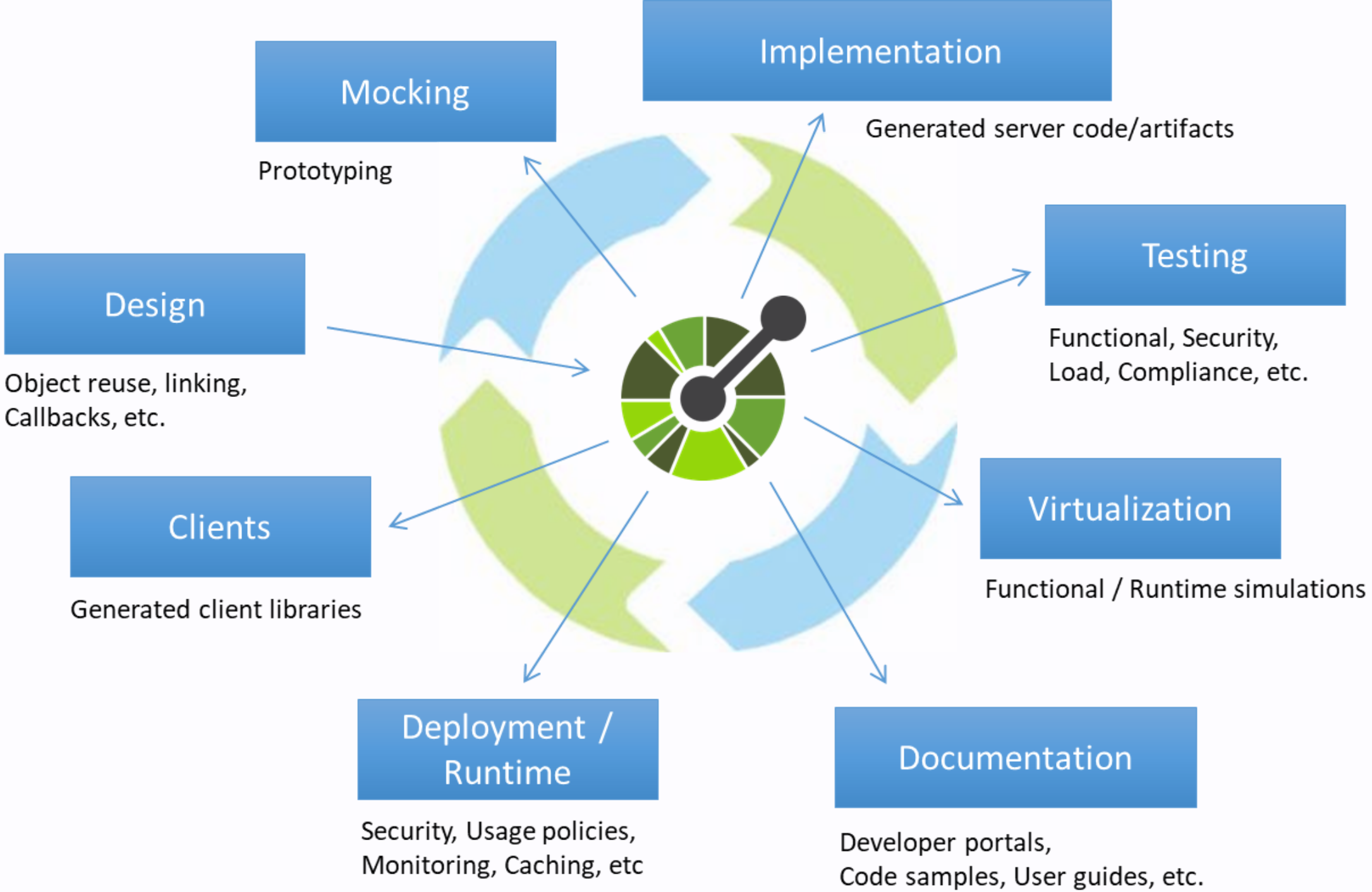
Scope of this talk

HTTP-centric API design

Challenges

- HTTP-centric impl leads to HTTP-centric design
  - hammar and nail
  - HTTP-elements become design elements (URIs, Methos, Headers, etc.)
- HTTP-centric design leads to HTTP-centric definitions
  - OpenAPI
  - gRPC
- HTTP-centrtic definitions lead to HTTP-centric governance
  - monitoring, rate-limiting, authentication, etc.
- Introduce other implementations (graphQL, etc.) breaks everything
  - different design/review rules, different tools, different monitoring etc.
  - slow experimentation, exploration, and roll-out of innovative solutions

API lifecycle

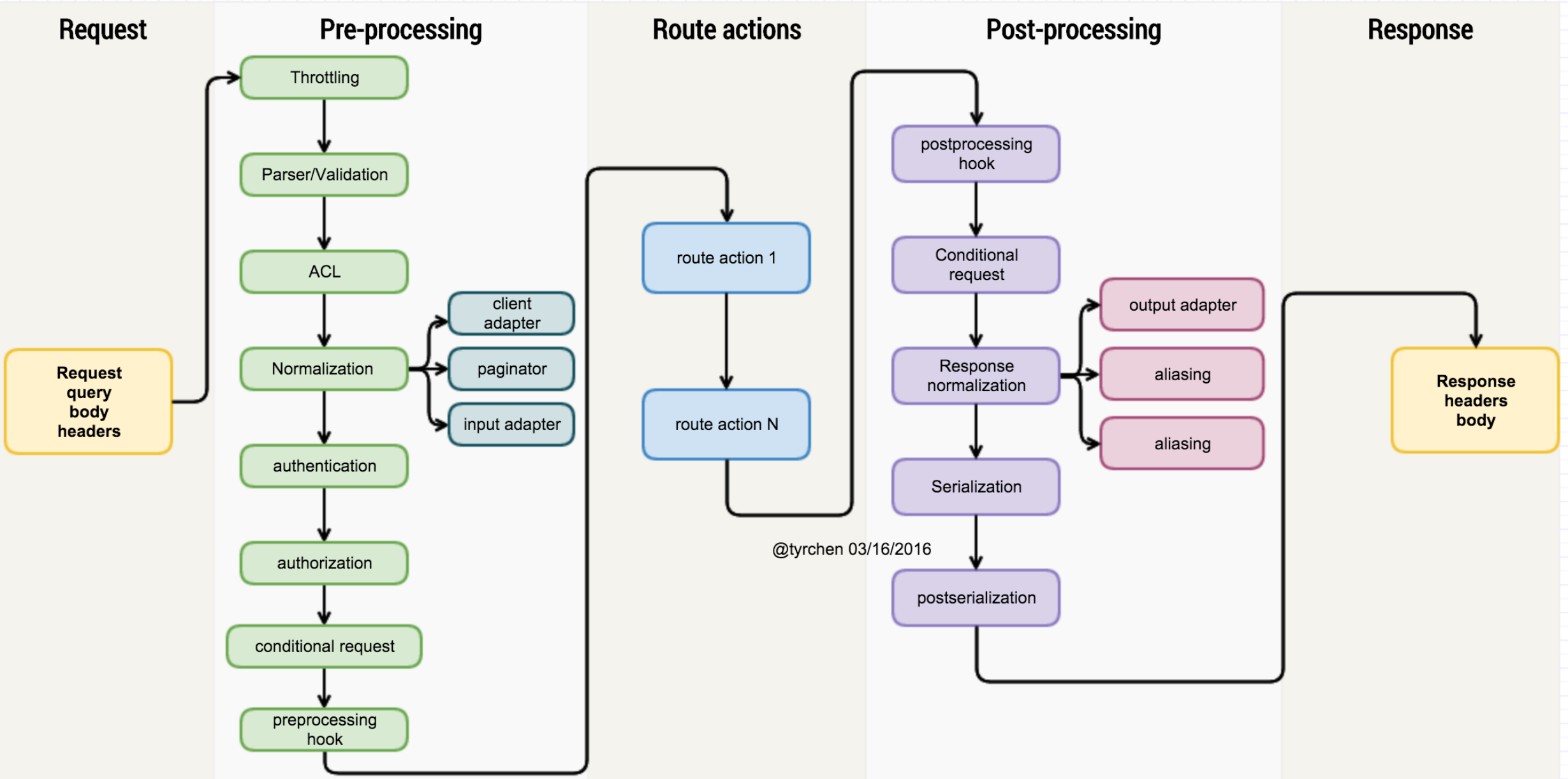

Mocking
Prototyping
Implementation
Generated server code/artifacts
Design
Object reuse, linking,
Callbacks, etc.
Testing
Functional, Security,
Load, Compliance, etc.
Clients
Generated client libraries
Virtualization
Functional / Runtime simulations
Deployment /
Runtime
Security, Usage policies,
Monitoring, Caching, etc
Documentation
Developer portals,
Code samples, User guides, etc.

rule 1: Spec Driven Development

rule 2: Pipeline as much as possible

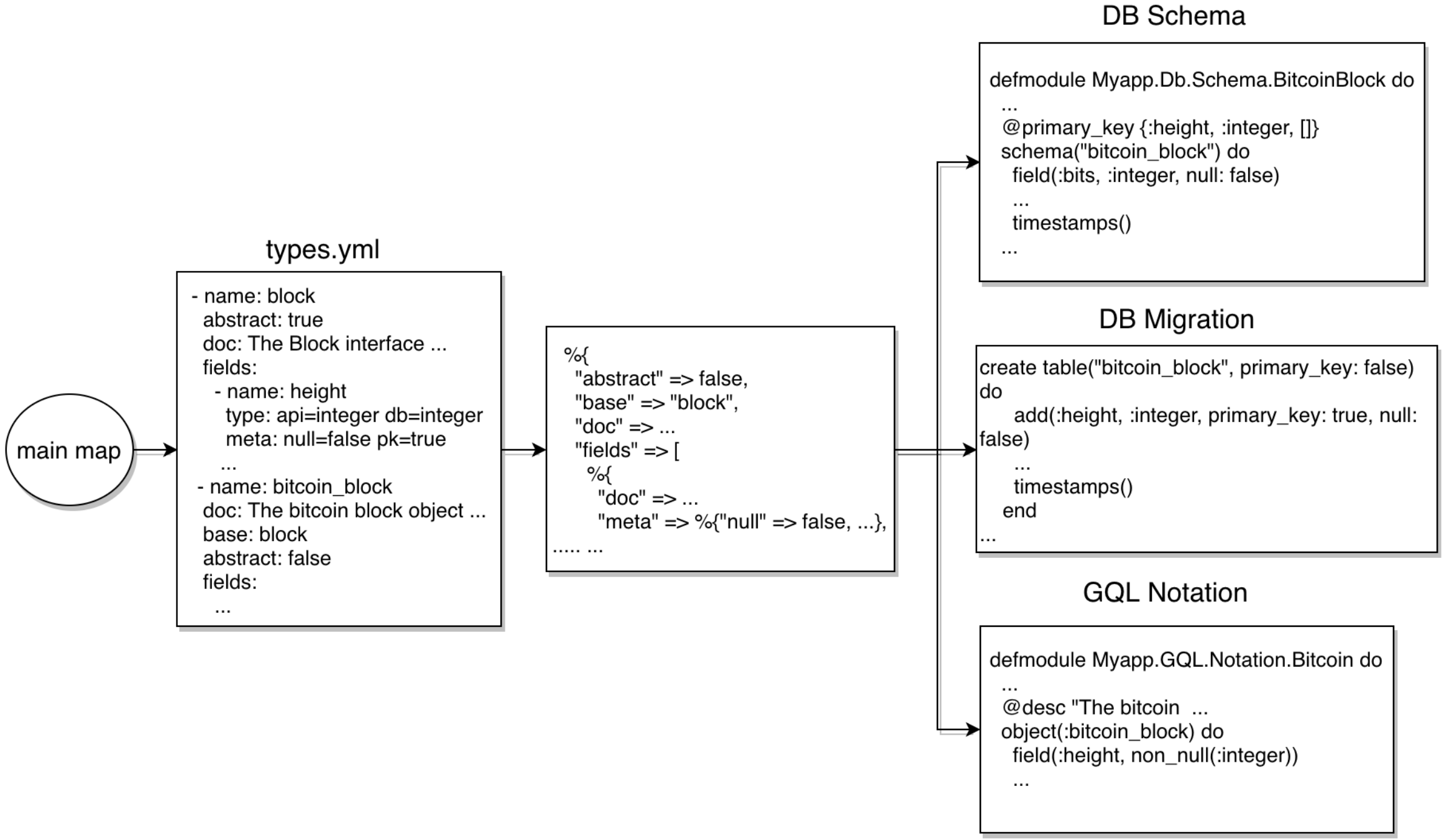
Why?

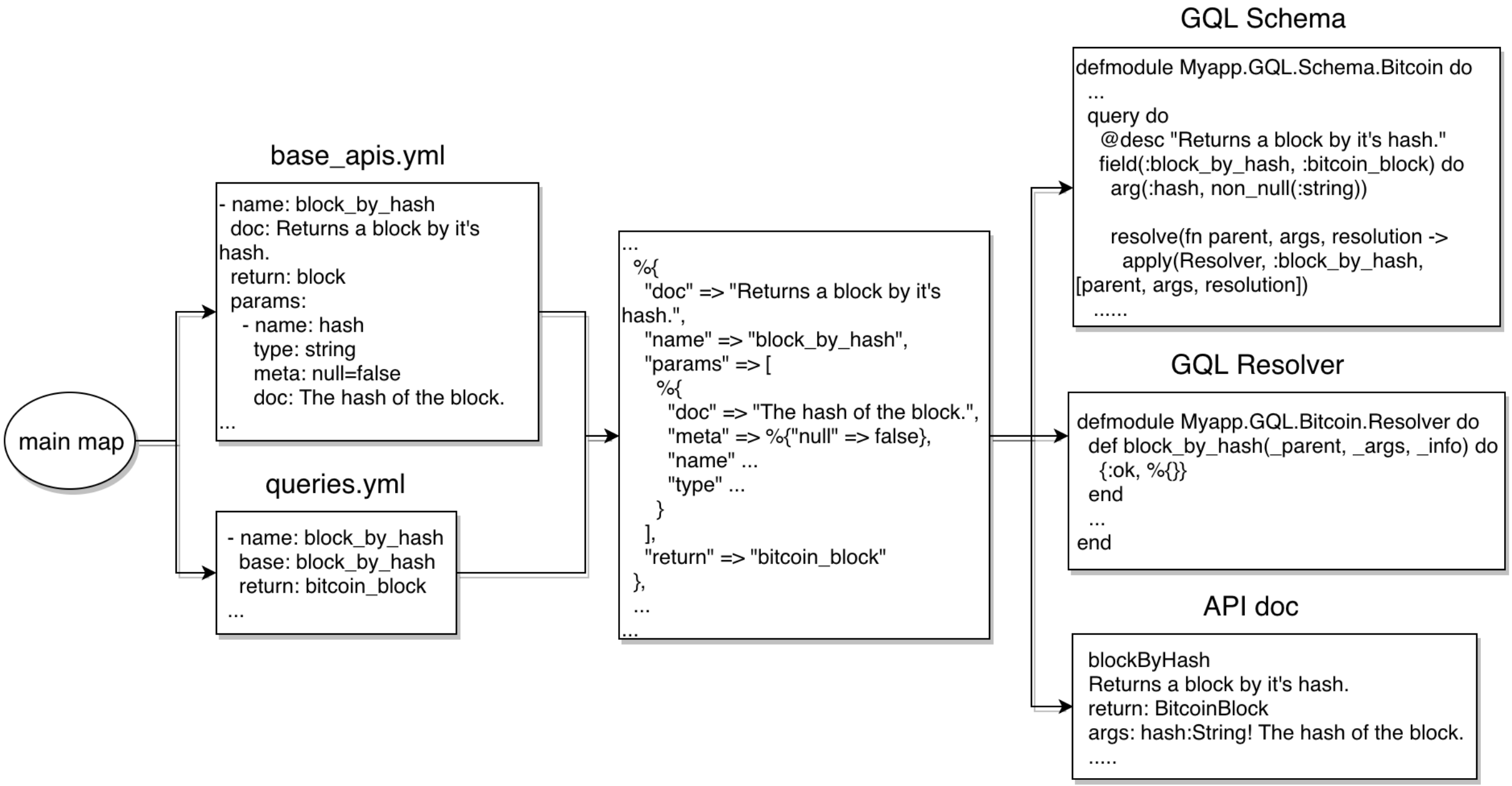
Pre-work: UAPI (tubi)


Request
Pre-processing
Route actions
Post-processing
Response
Request
query
body
headers
Throttling
Parser/Validation
ACL
Normalization
client adapter
paginator
input adapter
authentication
authorization
conditional request
preprocessing hook
route action 1
route action N
postprocessing hook
Conditional request
Response normalization
output adapter
aliasing
aliasing
Serialization
postserialization
Response
headers
body
@tyrchen 03/16/2016

Pre-work: goldrin (arblock)

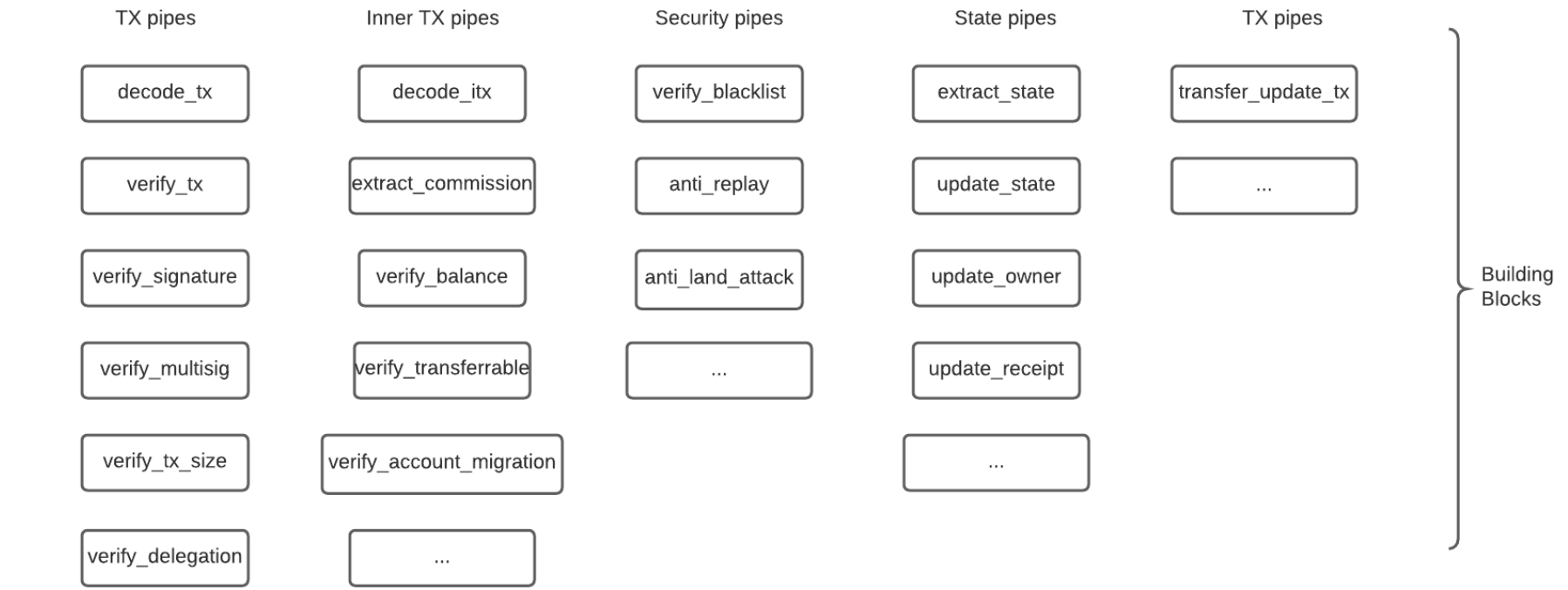
- YAML-style API definition for GraphQL
  - why not using GraphQL schema directly?
- Generate code for API layer
  - plugin: Absinthe (Notation and Schema)
  - generate CRUD resolvers / faker resolvers
  - generate runnable projects from API definition
- Generate code for DB layer
  - plugin: Ecto (Schema and Migration)
- Generate API documentation
  - plugin: slate (for human readable docs)
  - plugin: GraphQL Schema (leverage Absinthe)
- Generate client code (leverage apollo)
  - iOS / android / Web

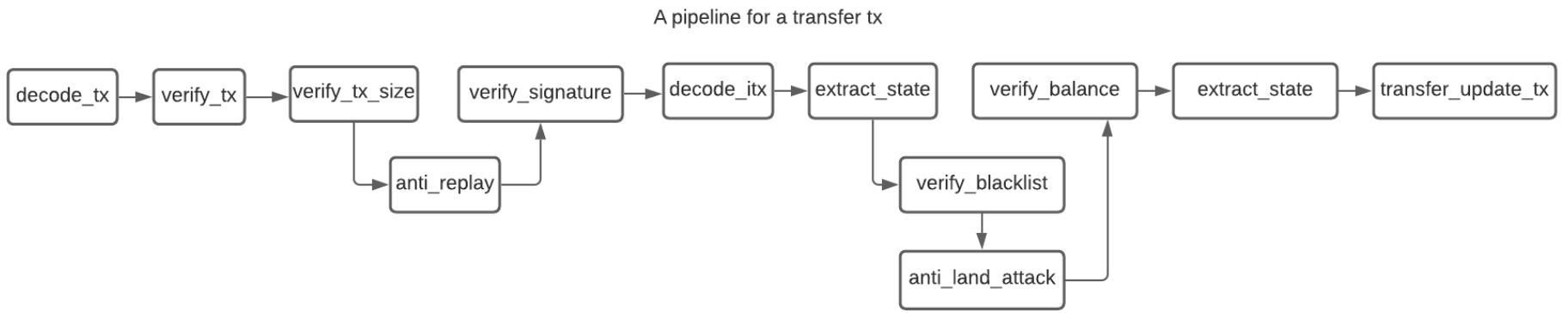

main map
types.yml
- name: block
abstract: true
doc: The Block interface ...
fields:
- name: height
type: api=integer db=integer
meta: null=false pk=true
...
- name: bitcoin_block
doc: The bitcoin block object ...
base: block
abstract: false
fields:
...
%{
"abstract" => false,
"base" => "block",
"doc" => ...
"fields" => [
%{
"doc" => ...
"meta" => %{"null" => false, ...},
....
DB Schema
defmodule Myapp.Db.Schema.BitcoinBlock do
...
@primary_key {:height, :integer, []}
schema("bitcoin_block") do
field(:bits, :integer, null: false)
...
timestamps()
...
DB Migration
create table("bitcoin_block", primary_key: false)
do
add(:height, :integer, primary_key: true, null:
false)
...
timestamps()
end
...
GQL Notation
defmodule Myapp.GQL.Notation.Bitcoin do
...
@desc "The bitcoin ...
object(:bitcoin_block) do
field(:height, non_null(:integer))
...


base_apis.yml
- name: block_by_hash
doc: Returns a block by it's hash.
return: block
params:
- name: hash
type: string
meta: null=false
doc: The hash of the block.
...
main map
queries.yml
- name: block_by_hash
base: block_by_hash
return: bitcoin_block
...
...
%{
"doc" => "Returns a block by it's hash.",
"name" => "block_by_hash",
"params" => [
%{
"doc" => "The hash of the block.",
"meta" => %{"null" => false},
"name" ...
"type" ...
}
],
"return" => "bitcoin_block"
},
...
...
GQL Schema
defmodule Myapp.GQL.Schema.Bitcoin do
...
query do
@desc "Returns a block by it's hash."
field(:block_by_hash, :bitcoin_block) do
arg(:hash, non_null(:string))
resolve(fn parent, args, resolution ->
apply(Resolver, :block_by_hash, [parent, args, resolution])
.....
GQL Resolver
defmodule Myapp.GQL.Bitcoin.Resolver do
def block_by_hash(_parent, _args, _info) do
{:ok, %{}}
end
...
end
API doc
blockByHash
Returns a block by it's hash.
return: BitcoinBlock
args: hash:String! The hash of the block.
.....

Pre-work: forge pipes (arcblock)

- Break complicated blockchain TX logic into small, chainable logics
  - Steal the idea from Elixir Plug
- YAML-style TX pipeline definition
  - execution of a blockchain TX is described as a set of ordered pipes (think plugs)
- Generate code for the pipeline execution
  - execution model: run-to-completion (bail out if error)
- Generate client SDK for sending TX (partially)
- Easy interface to writing pipes
  - Each pipe gets data to manipulate and a context, then return the data and context
- Pretty easy to add new TX
  - Tens of common pipes already there to be used
  - adding new transaction support mostly just glue common pipes and TX specific pipes in YAML


TX pipes
Inner TX pipes
Security pipes
State pipes
TX pipes
decode_tx
verify_tx
verify_signature
verify_multisig
verify_tx_size
verify_delegation
decode_itx
extract_commission
verify_balance
verify_transferrable
verify_account_migration
...
verify_blacklist
anti_replay
anti_land_attack
...
extract_state
update_state
update_owner
update_receipt
...
transfer_update_tx
...
Building Blocks


A pipeline for a transfer tx
decode_tx
verify_tx
verify_tx_size
anti_replay
verify_signature
decode_itx
extract_state
verify_blacklist
anti_land_attack
verify_balance
extract_state
transfer_update_tx

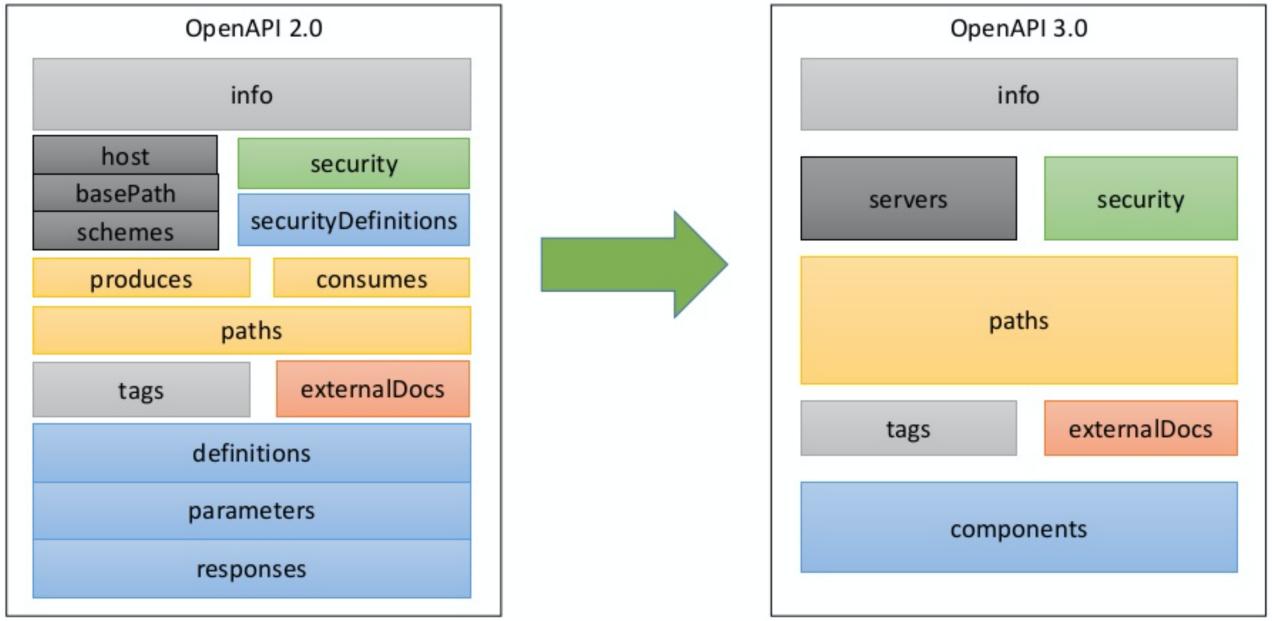
My learnings

- Spec, spec, spec!
  - programming language agnostic contract
  - single source of truth for BE, clients, PMs and tools (mocking server, playground, etc.)
- Type safety for input & output
  - eliminate potential data problems
  - made code generation much easier
- Whenever possible, generate the code!
  - sever code (adding new APIs is more like filling in the blanks than writing an essay)
  - client SDKs (lower the bar for various clients)
  - docs, tests, playground, etc. (lower the bar of consuming APIs)
- Thinking in pipeline (nested pipelines)
  - break actions to smallest execution unit (think about particles, elements)

OK, what will I look at on building next-gen APIs?

Type	REST	Streaming	Bi-directional
OpenAPI	Y	N	N
GraphQL	N	N	Y
gRPC	N	Y	Y
gRPC w/ grpc-gateway	Y	N	N
...	..	..	..

OpenAPI(v3)



OpenAPI intro

- Follow up version of swagger 2.0
- Defines specs with YAML/JSON
  - You can learn the grammer in a few hours
- Strong tooling support
  - playground (swagger), codegen, docgen, etc.
  - test case generator: tcases
  - Other tools: openapi.tools

Example

```yaml
components:
  schemas:
    status:
      type: string
      enum: [active, completed]
    todoId:
      type: string
      format: uuid
    todoBody:
      type: string
      minLength: 3
      maxLength: 140
    dateTime:
      type: string
      format: date-time
    Todo:
      type: object
      properties:
```

```yaml
paths:
  /todo/{todoId}:
    get:
      operationId: getTodo
      tags: [todo]
      description: get a todo item
      parameters:
        - name: todoId
          in: path
          required: true
          description: The id of the pet to retr
          schema:
            $ref: "#/components/schemas/todoId"
      responses:
        "200":
          $ref: "#/components/responses/Todo"
        default:
          $ref: "#/components/responses/Error"
```

OpenAPI demo

Pros / Cons

- Pros
  - Easy to use (swagger playground is a big plus!)
  - Pretty mature tooling to reduce lots of (manual) work
  - Good ecosystem (e.g. AWS API gateway support it)
  - Can generate GraphQL schema if you want to move to GraphQL one day
- Cons
  - A small learning curve (mainly learn to write OpenAPI spec)
  - No elixir spec -> code generator (erlang one is pretty odd)
    - Code generated for server stub miss all kinds of validations
  - General REST API drawbacks
    - no streaming support, not bidirectional
    - not flexible for clients (multiple calls to render a page, unnecessary data returned, etc.)

GraphQL

GraphQL intro

- Created by Facebook
- Gateway to all APIs
- Client focused
  - Easy to map to presentation logic
- Tooling
  - Apollo
  - AWS amplify
  - godrin (private unfortunately)

Example

```
{
  search(query: "tyrchen", type: USER, first: 10) {
    edges {
      node {
        ... on User {
          id
          email
          bio
          name
          pinnableItems(first: 10) {
            nodes {
              ... on Repository {
                id
                name
                url
                updatedAt
                createdAt
              }
```

GraphQL Demo

Pros / Cons

- Pros
  - Easy to use (GraphQL playground is even greater than swagger!)
  - Flexible query makes client easy to get the right amount of data it needs
    - Save client API requests round trips
  - Subscription is great for building event driven apps
  - A good aggregate layer to micro services
  - Apollo toolchain is pretty decent
  - Good ecosystem (e.g. AWS AppSync support it)

- Cons
  - A big learning curve for both client and server devs
  - Breaks general HTTP ecosystem
    - Every request is a POST (breaks caching)
    - Every response is 200 (breaks the HTTP semantics)
    - All APIs inside a schema point to same API location (breaks URI based routing, and monitoring ecosystem)
  - Complexity of a query sometimes pretty tricky
  - N + 1 problem (dataloader fixed part of the issue)
  - Need extra work for logging, monitoring, caching, etc.

gRPC (w/ grpc-gateway)

gRPC/grpc-gateway intro

- Write proto with gRPC (and certain extentions)
- Serve API with grpc-gateway, as a proxy to gRPC server
  - need some knowledge in golang
- Could generate swagger or openapi spec

Examples

```
option (grpc.gateway.protoc_gen_swagger.options.
  info : {
    title : "User API",
    version : "0.1",
    description : "User API"
  };
  security_definitions : {
    security : {
      key: "ApiKey";
      value : {
        type: TYPE_API_KEY;
        in: IN_HEADER;
        description: "access token generated by
        name: "Authorization";
      };
    };
  };
  schemes : HTTP;
```

```
service UserService {
  // user signin
  rpc Signin(RequestSigninOrRegister) returns (R
    option (google.api.http) = {
      post : "/api/v1/users/signin"
      body : "*"
    };
    option (grpc.gateway.protoc_gen_swagger.opti
      summary : "User signin"
      description : "Sign in a user by using an
      tags : "Users"
    };
  }
  ...
}
```

gRPC/grpc-gateway DEMO

Pros / Cons

- Pros
  - Generate OpenAPI/Swagger Spec from existing gRPC services
  - Auto generate the proxy code
  - Setup once
- Cons
  - Need to write and maintain a little bit golang code
  - Need to build client-facing APIs completely in gRPC (and then proxy it)
  - Need to learn grpc-gateway protobuf extensions
  - Certain annotation is done in comments (e.g. "Output only"), not easy to maintain

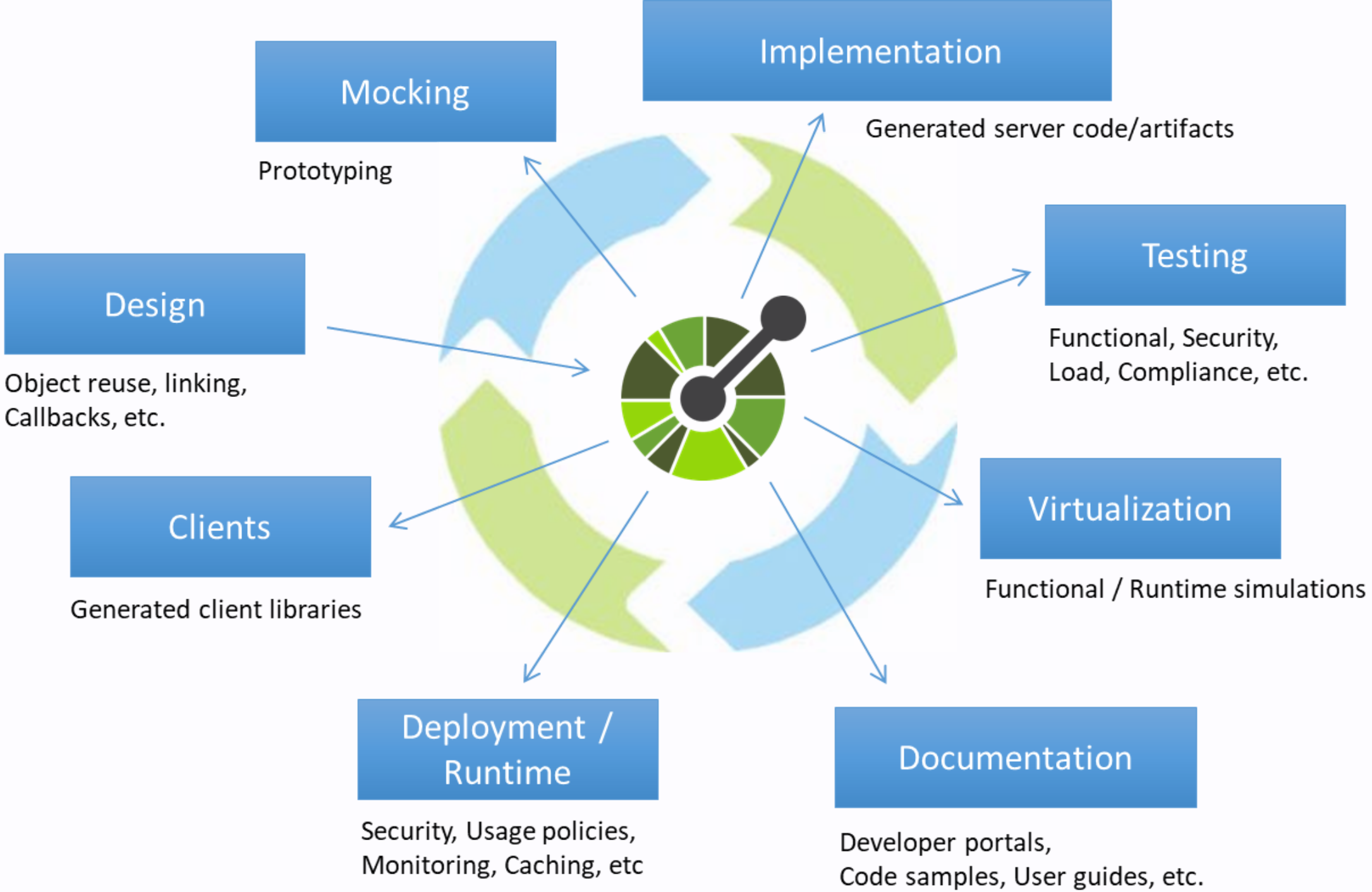
Other solutions

- Smithy
  - the tool backing aws APIs, IDL pretty powerful
  - support tagged union for data structure
- rest.li
  - linkedIn API tool, optimized for client use
  - API can be mapped to presentation layer pretty easy (e.g. AttributeString)
- AsyncAPI
  - good for event-driven arch, e.g. websocket, mqtt
- Build your own IDL

```
{
  "title": {
    "text": "Tyr Chen liked this",
    "attrs": [
      {"type": "PROFILE_FULL_NAME", "start": 0,
    ]
  }
}
```

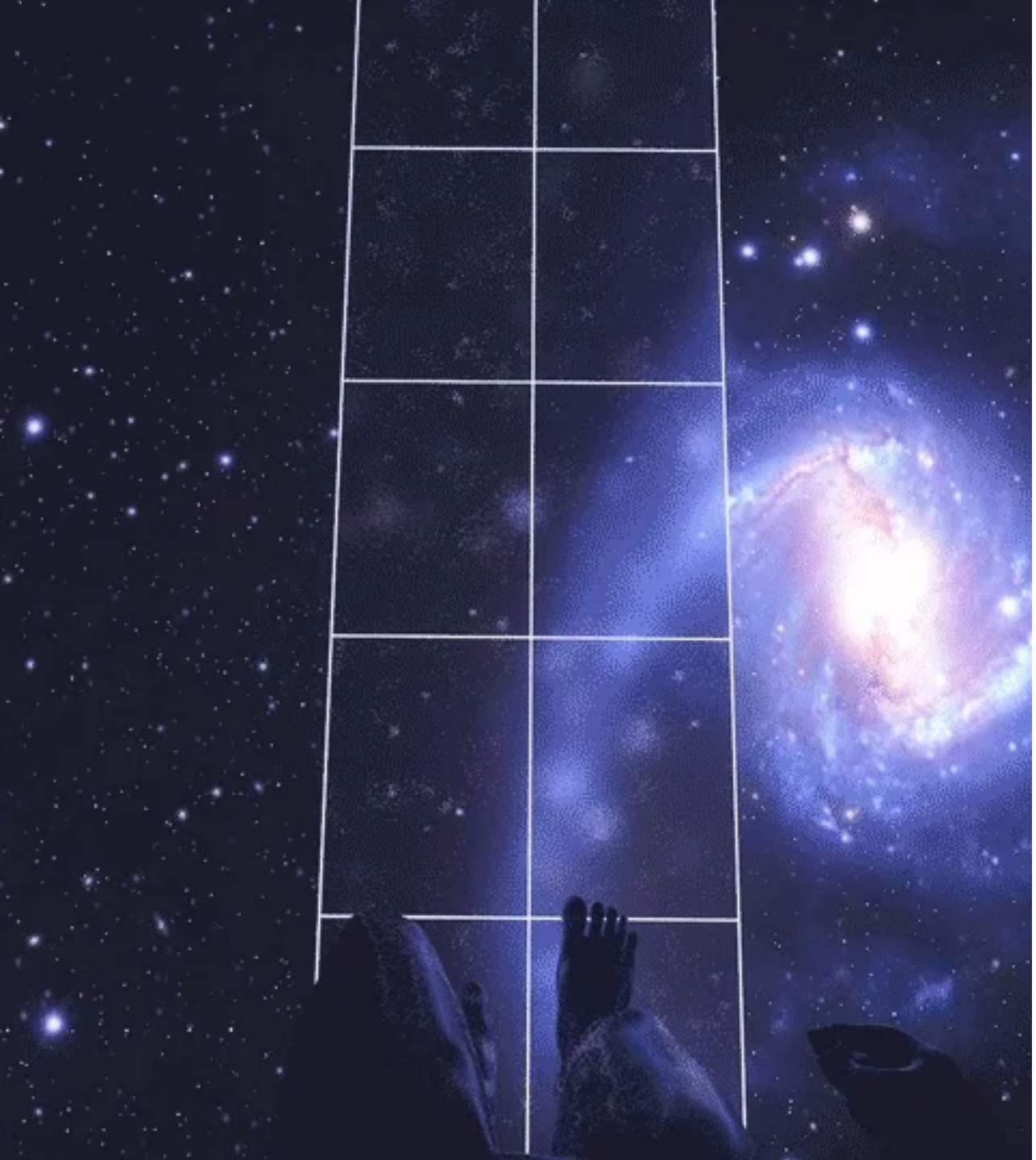
Build your own IDL

- Methods
  - seek first to extend existing IDLs - OpenAPI, Protobuf or GraphQL
  - if not, try to make something with yaml or toml, if you don't want to define new grammars
  - if you want to define your own grammers, please study existing solutions first
- Strategy
  - build a parser to parse your IDL to OpenAPI / protobuf / GraphQL IDL
  - leverage existing toolchains to support code generation, playground, documentation, etc.
  - Focus on usability


Mocking
Prototyping
Implementation
Generated server code/artifacts
Design
Object reuse, linking,
Callbacks, etc.
Testing
Functional, Security,
Load, Compliance, etc.
Clients
Generated client libraries
Virtualization
Functional / Runtime simulations
Deployment /
Runtime
Security, Usage policies,
Monitoring, Caching, etc
Documentation
Developer portals,
Code samples, User guides, etc.

Trends in serverless era

- AWS API gateway (OpenAPI)
- AWS AppSync (GraphQL)
  - managed GraphQL gateway
  - realtime databroker (subscribe to any mutation out of the box)



Q&A

The best way to predict the future is to **invent** it.

— Alan Kay