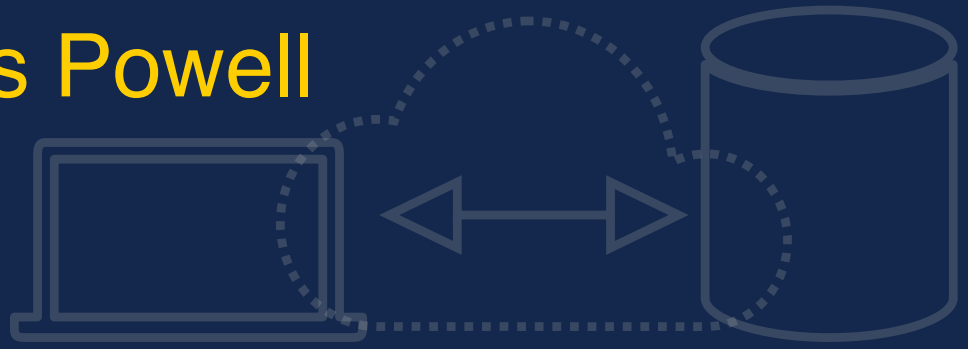
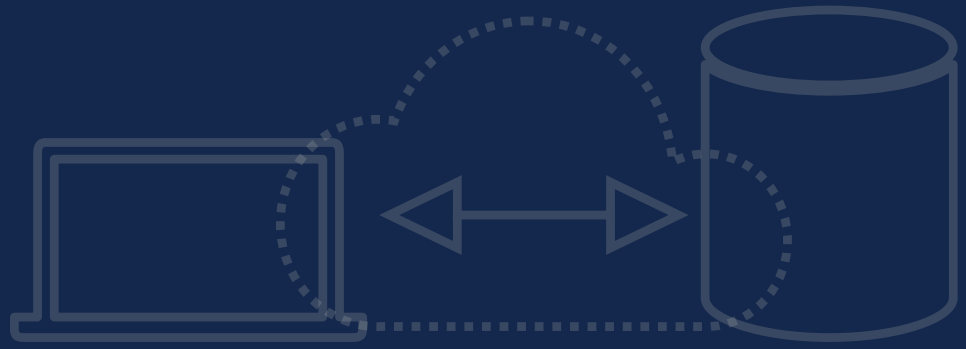
REST, CRUD, MVC Architecture and More

with Thomas Powell



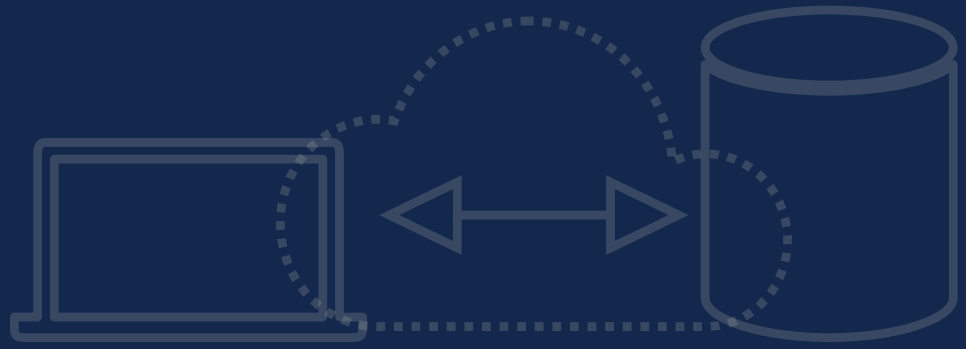
REST

- REpresentational State Transfer
  - Roy Fielding PhD work around 2000
  - Describes method of defining resources as URLs and actions on those resource with HTTP methods
    - URLs as nouns
    - HTTP methods as verbs on those nouns



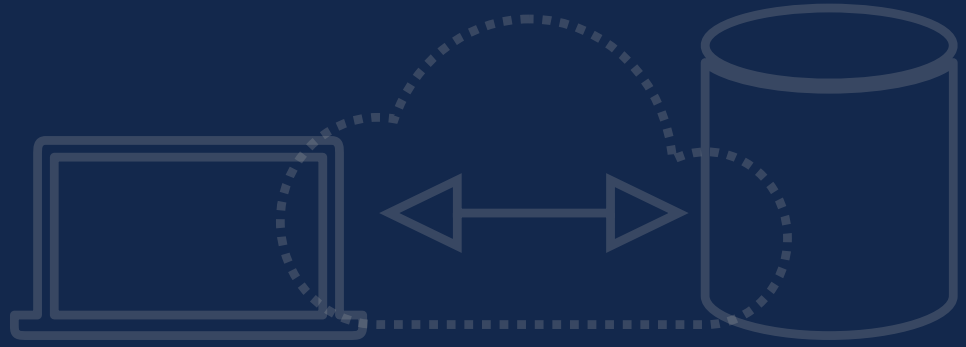
CRUD

- CRUD = **C**reate **R**ead **U**pdate **D**elete
- Pattern of the actions we might perform on a resource or piece of data
- A simple idea but often the heart of apps built for business applications



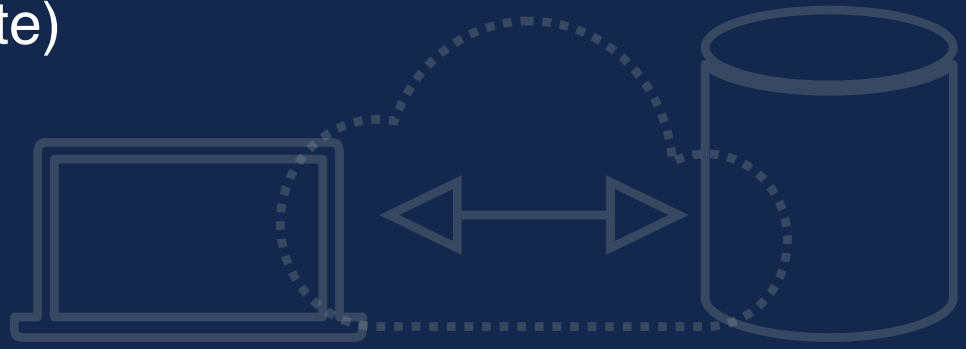
RESTful vs REST

- There are specific definitions to being REST
- RESTful = in the spirit of REST than strictly conforming
  - Go too far though and it becomes non-RESTful
  - No or few HTTP verbs (mostly GETs for example)
  - Verb/Action focused URLs as opposed to entity/resource URLs
  - Endpoints that “do” more than the typical CRUD actions or consolidate actions



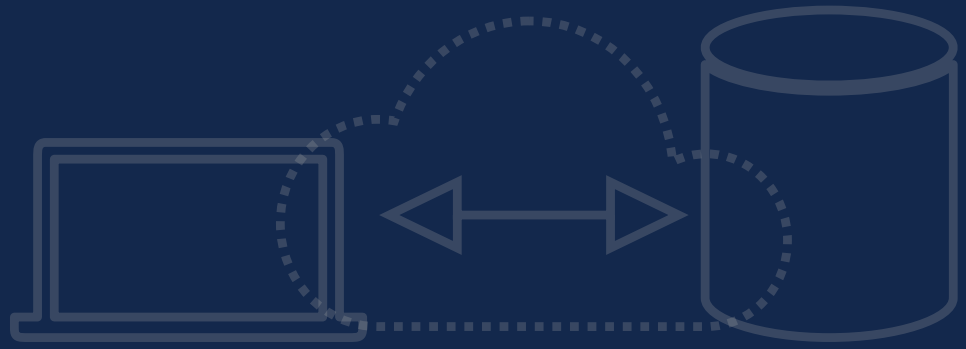
Put Another Way

- REST has you think of the URL as a command line
- The resource is the “file” or entity
  - http://example.com/dogs/1 (an object)
- The HTTP methods used to access the object correspond to the actions to perform on the entity
  - POST - makes a new entity (**C**reate)
  - GET - fetches the entity (**R**ead)
  - PUT, PATCH - update an entity (**U**pdate)
  - DELETE - delete an entity (**D**elete)



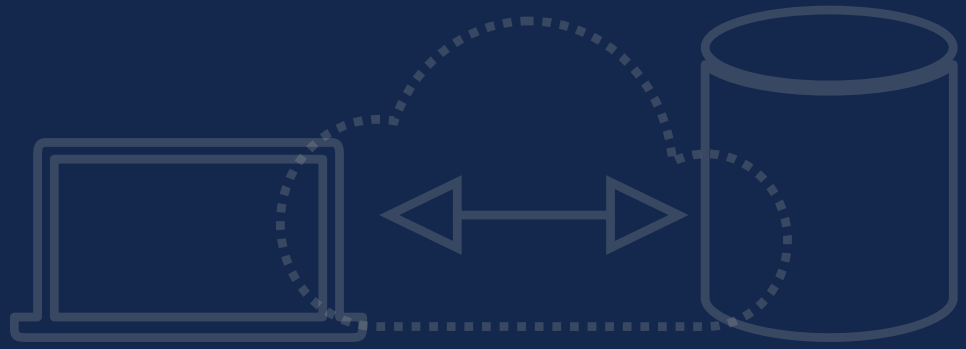
Read Examples

- `GET /api/books` *get all the books*
- `GET /api/books/1` *get book of id = 1*
- `GET /api/books/:1` *same but more common id identification*
- `GET /api/books?id=1` *same but not path based*
- Watch out! Getting less RESTful
  - `GET /api?record=books&id=1`
- Too far, not RESTful for multiple reasons
  - `GET /api?id=1&action=delete`



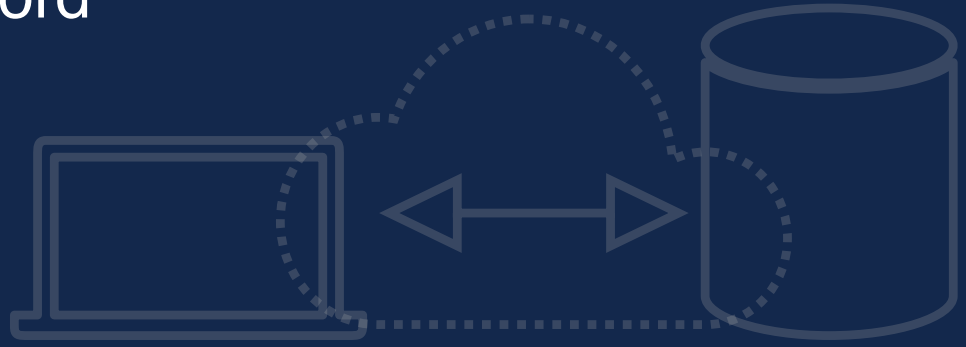
Create Example

- **`POST /api/books`** *Creates a new book*
- Submitted values should be sent in message body with appropriate Content-Type header
  - **`x-www-form-urlencoded`** or **`JSON`**
- Do not send the new entity in query string!
- Response with created id or full record created including the id



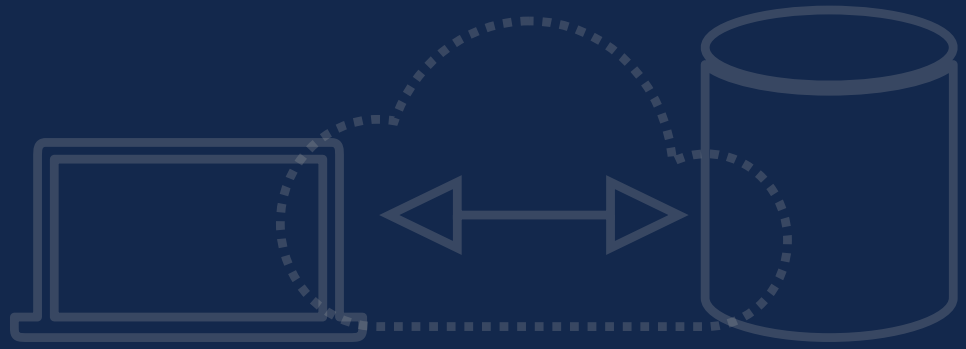
Update Example

- **`PUT /api/books/30`** *Modify book with id = 30*
- Data is sent in message body
  - Same format consideration as create (POST)
- Use **`PATCH`** if partial update / **`PUT`** for full update
  - **`PATCH`** is not always used
- Return is often 1 or 0 for accomplishment.
  - Might also be modified record id value or 0
  - Might also be full modified record



Delete Examples

- `DELETE /api/books/30` delete book with id=30
- No payload required
- Return 1 or 0 for Boolean status
  - Sometimes return record if deleted or 0 if fail

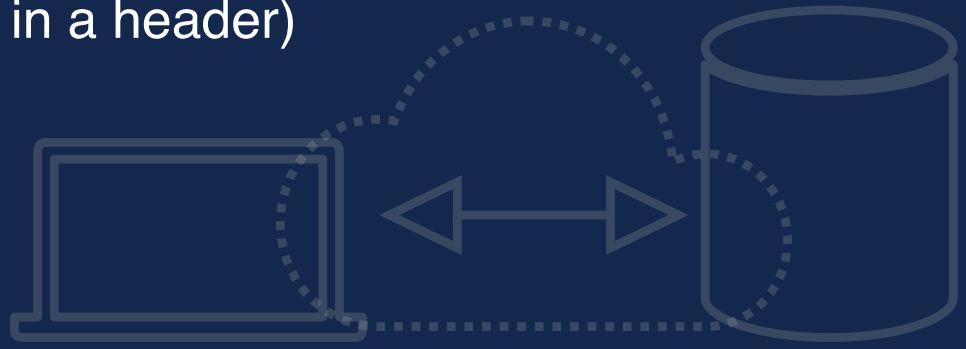


Summary

GET	/movies	Get list of movies
GET	/movies/:id	Find a movie by its ID
POST	/movies	Create a new movie
PUT	/movies /:id	Update an existing movie
DELETE	/movies /:id	Delete an existing movie

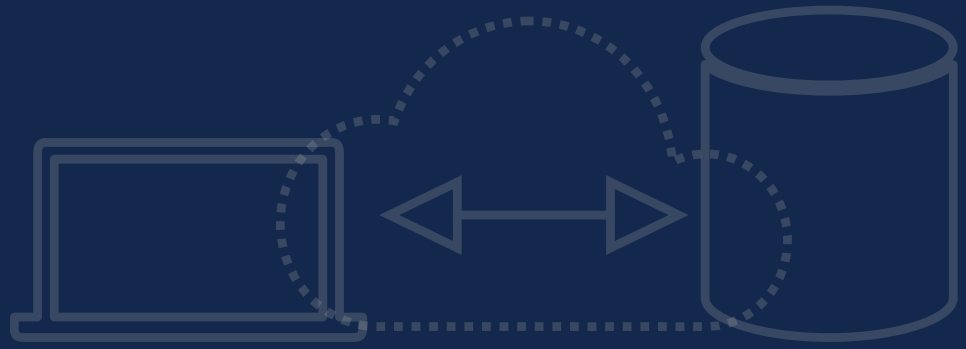
Method Danger?

- It is indeed possible that HTTP verbs required for REST are not allowed
  - Proxies
  - Security?
  - Coding Approach
- Solutions
  - Encrypt whole way
  - OPTIONS method and negotiate
  - Fall back to GET and POST
    - This means READ = GET all other actions are POST with some flag indicating actual action (often in a header)



Request and Response Variation

- For each of the actions you'll notice there seems to be some ambiguity in what is sent and received
- Data Format - JSON, XML, Other
- Response Style - Simple Boolean vs Records
- HTTP Response Codes are often overlooked but make things more RESTFul

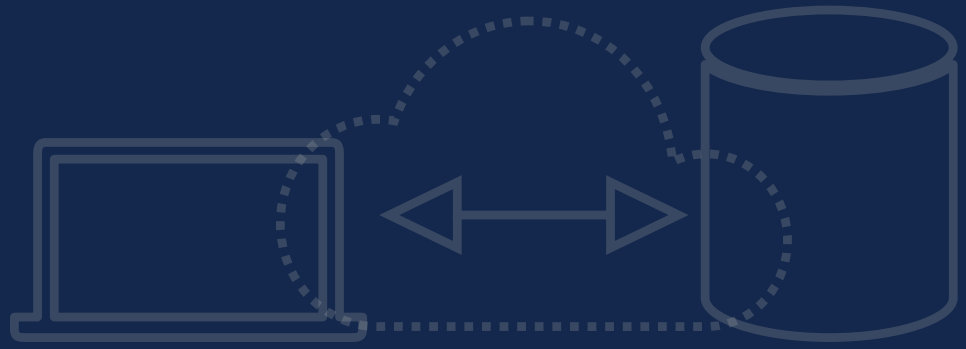


Possible Response Codes

- 200 Ok - Return if the action was ok
- 201 Created - Useful to return if something was created (POST). Often we might return a Location header of the newly created object. Some people echo the newly created object. Upon an update (PUT) we might do the same
- 204 No Content -Might be useful on Delete where we don't return a data status code.
- 400 Bad Request - Malformed requests
- 401 Unauthorized - Access Control
- 403 Forbidden - Access Control
- 404 Not Found - Resource wasn't there
- 405 Method Not Allowed - Misuse of an HTTP verb example create on a /:id URL
- 409 Conflict - May result when doing an update when resource can't be modified
- 500 Internal Server Error

Data Type Selection

- Choosing a data type should be driven by
  - Ease of encode/decode (use) of data type on client and server
  - Integrity or Reliability Concerns
  - Performance Concerns
  - Security Concerns
- In practice choosing a data type is often driven by convention or pattern

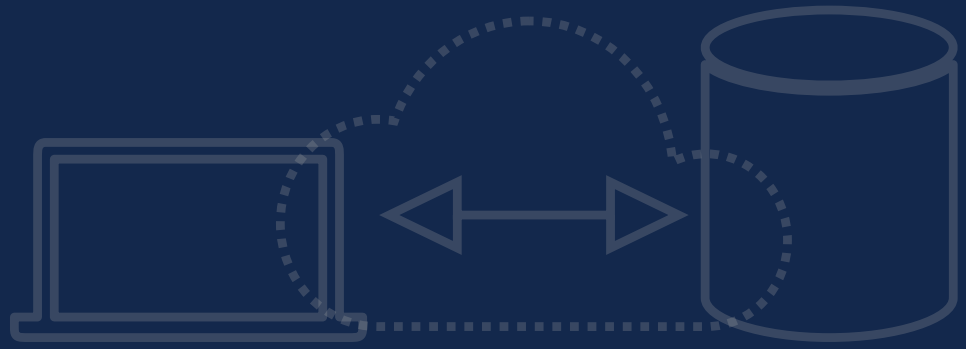


Understand Context of Choice

- Designing your end point and choosing your data types really does depend on context
- Public API vs Private API
  - Change your API if public at your PERIL
  - Even versioning is not a fix but maybe a headache with little chance for upgrade
    - `POST /api/Books` vs `POST /v1/api/Books` vs `POST /v2/api/Books`
    - Try getting people to change this
  - If you control client as well this may not be such a big issue

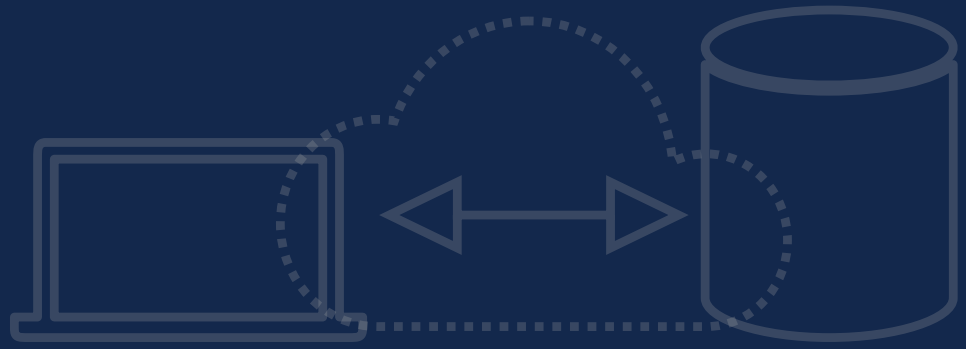
Public vs Private Points

- Public
  - Relatively unchanging and when changes likely can't force upgrade
  - If changes it follows Open-Closed Principle
    - https://en.wikipedia.org/wiki/Open%E2%80%93closed_principle
    - This principle tends to lead to code/syntax bloat though if my experience is any guide :-(
  - Has to be understandable and documentation may be considerable
  - Has to assume malfeasance inherently
    - Malformed submissions, rate abuse, API key share, etc.



Public vs Private Points

- Private API endpoints should not necessarily be designed the same as public ones because
  - Change may be easier if you control the client code which might allow for less design up front
  - Efficiency or security changes may be less impacted because ease of use for outside API users is not a consideration
- Some might argue that all APIs be designed as if they are public … it seems to have been good?



Bezos API Principle

THE API MINDSET FOR IT AND BUSINESS

THE JEFF BEZOS MEMO, 2002

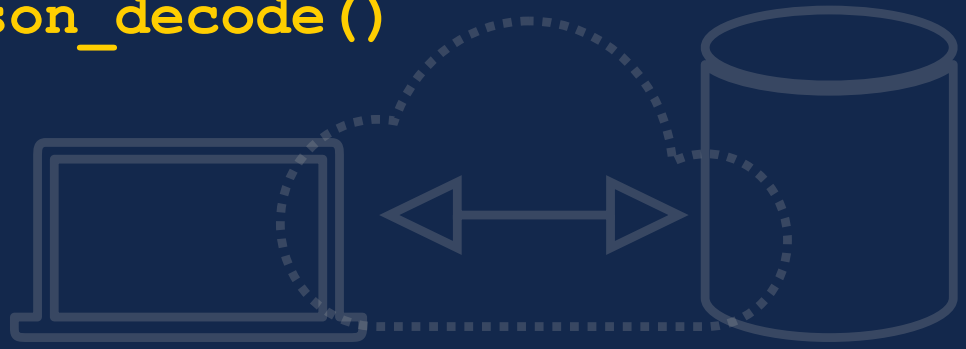
- All teams will henceforth expose their data and functionality through **service interfaces**.
- Teams must communicate with each other **through these interfaces.**
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. **The only communication allowed is via service interface calls over the network**.
- It **doesn't matter what technology** they use.
- All service interfaces, without exception, **must be designed from the ground up to be externalizable**. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

Bezos API Principle

https://apievangelist.com/2012/01/12/the-secret-to-amazons-success-internal-apis/

JSON

- JavaScript Object Notation
- JS type subset that is serialized
  - Strings, Numbers, Booleans, Null, Arrays and Stringified Objects (string keys)
- Super easy to use
  - JS object -> JSON `JSON.Stringify()`
  - JSON -> JS ~~`eval()`~~ `JSON.Parse()`
- Most languages have built-ins or easy access to libraries
  - PHP - `json_encode()` and `json_decode()`

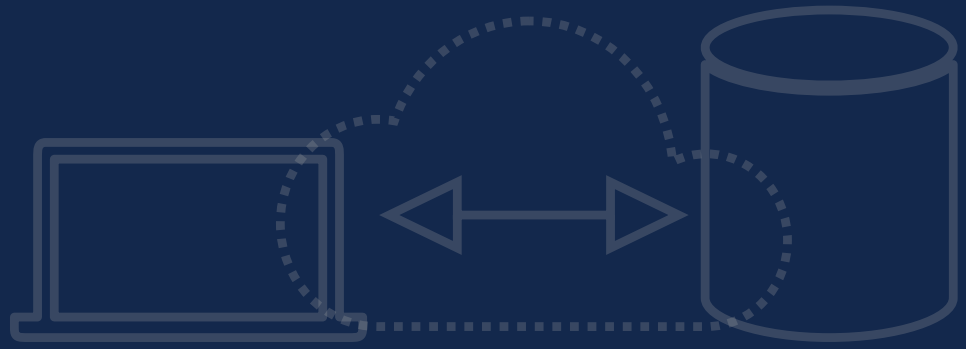


JSON Example

```
let obj = {
  name : 'Zelda',
  type : 'Dog',
  breed : 'Schnauzer',
  age : 2,
  friendly : true
};

// Serialize this object

let payload = JSON.stringify(obj);

// result
{"name":"Zelda","type":"Dog","breed":"Schnauzer","age":2,"friendly":true}
```

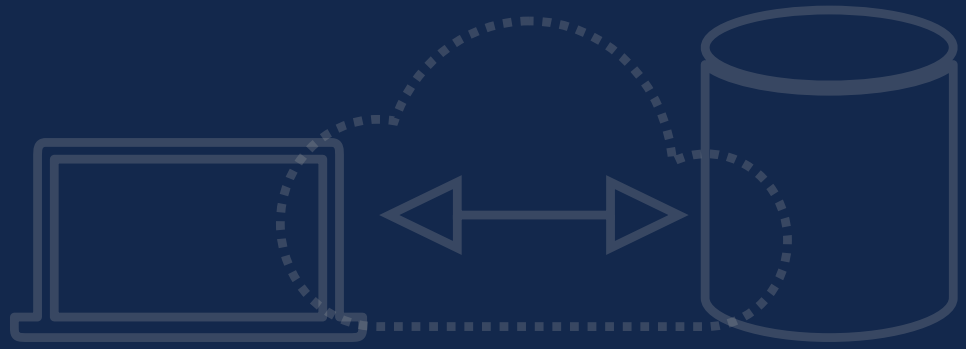
JSON Points

- Super popular right now
- Easy to use but on wire a bit wordy
- Issues
  - Malformedness issues
  - Dangling commas
  - Comments?
  - Schemas and Enforcement of Schemas



XML

- e**X**tensible **M**arkup **L**anguage
  - Language in which you can define other languages
  - The modern dependent of SGML
  - Define markup languages and data packets with it
    - Formal definitions - Document Type Definitions (DTD) or Schemas
    - Informal / Ad-hoc Definitions - just use XML syntax in well-formed fashion
  - Well Formed - syntax < / “ is right
  - Valid = Well Formed + Semantic Use (DTD/Schema is adhered to)
  - HTML use informs this sadly where well formedness isn’t even always met



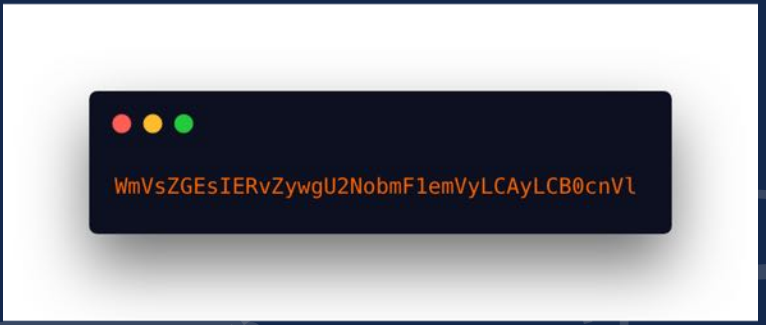
XML Example

```xml
<payload>
  <name>Zelda</name>
  <type>Dog</type>
  <breed>Schnauzer</breed>
  <age>2</age>
  <friendly>true</friendly>
</payload>
```

Other Formats

- We can send just about anything
- CSV - terse but brittle!

- Encoded - visual security
  maybe size



Demo: **http://ajaxref.com/ch4/responseexplorer.html**

No Major Reason to Be Format Dogmatic

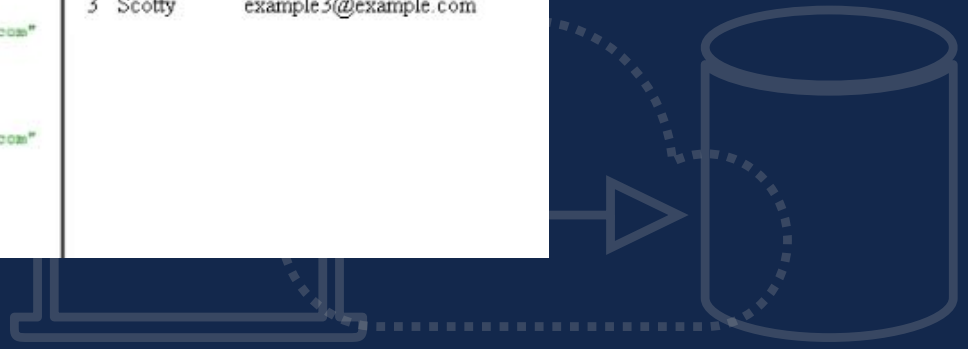
- For public APIs multiple response formats might be possible
  - Determined by query string or `Accept` header or `X-`*`something`* header or even path?

XML	JSON	HTML
users	users/format/json	users/format/html

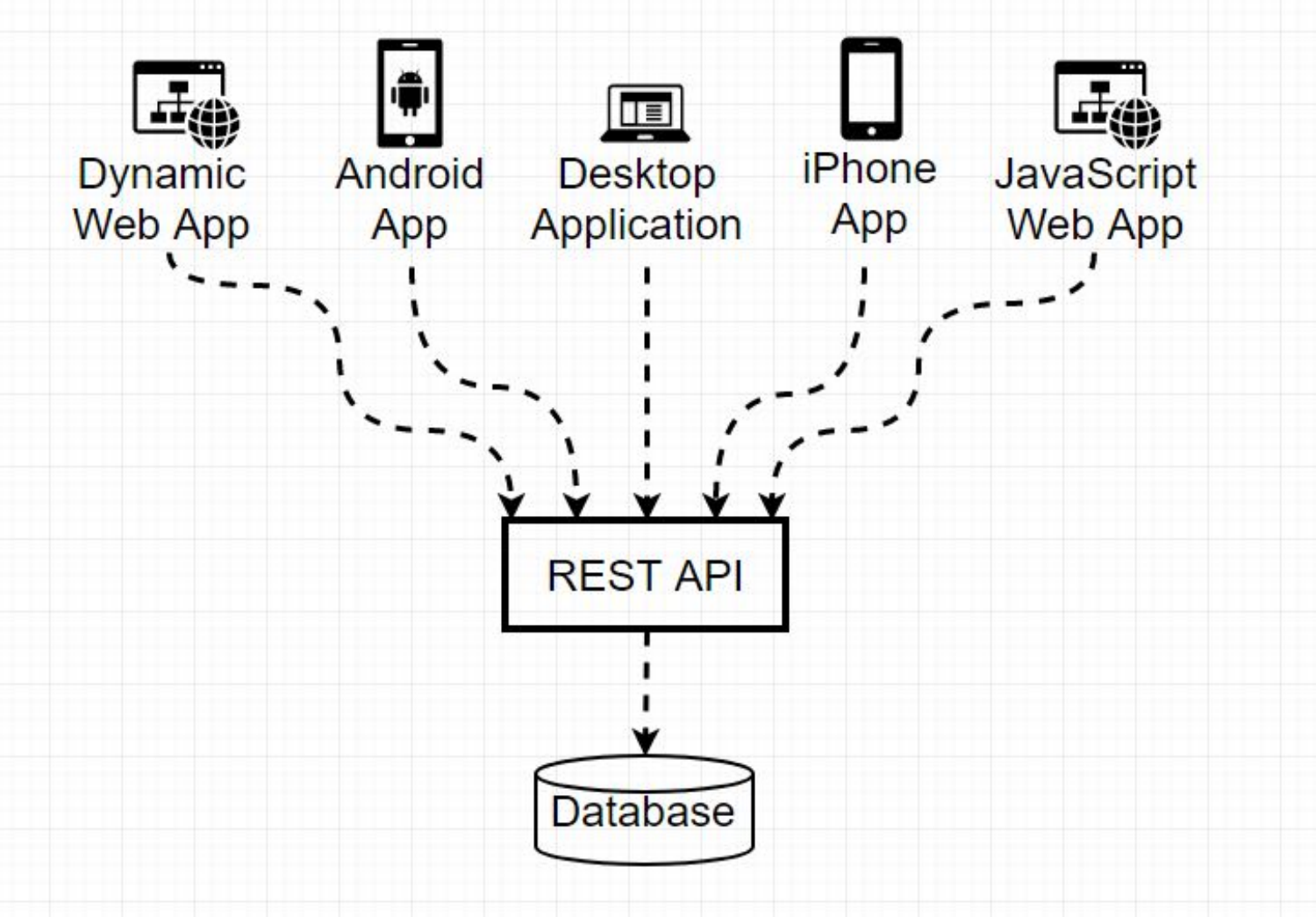
```
- <xml>
  - <item>
      <id>1</id>
      <name>Some Guy</name>
      <email>example1@example.com</email>
    </item>
  - <item>
      <id>2</id>
      <name>Person Face</name>
      <email>example2@example.com</email>
    </item>
  - <item>
      <id>3</id>
      <name>Scotty</name>
      <email>example3@example.com</email>
    </item>
</xml>
```

```
[
  - {
      id: 1,
      name: "Some Guy",
      email: "example1@example.com"
    },
  - {
      id: 2,
      name: "Person Face",
      email: "example2@example.com"
    },
  - {
      id: 3,
      name: "Scotty",
      email: "example3@example.com"
    }
]
```

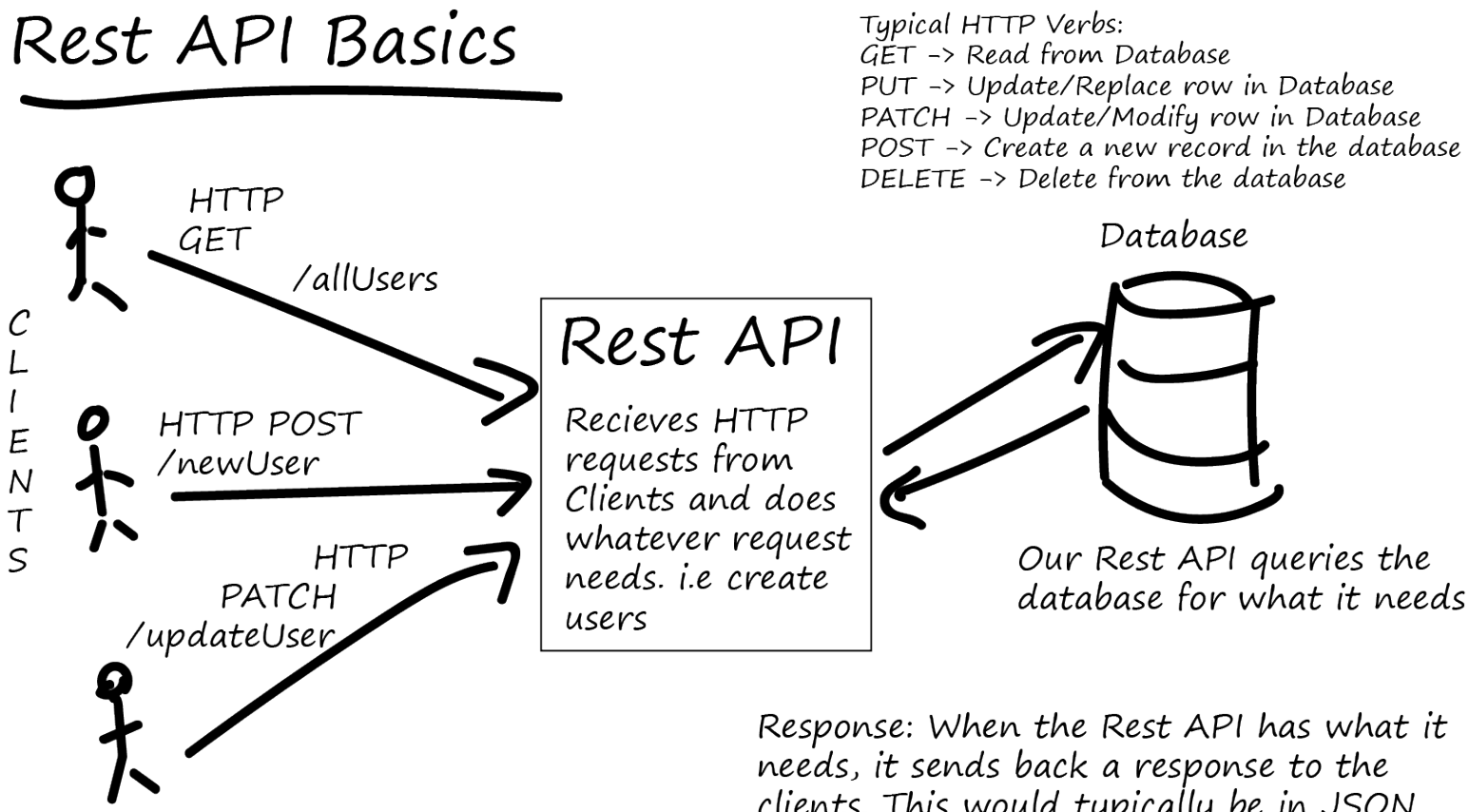
id	name	email
1	Some Guy	example1@example.com
2	Person Face	example2@example.com
3	Scotty	example3@example.com



REST is Also Independence of Clients

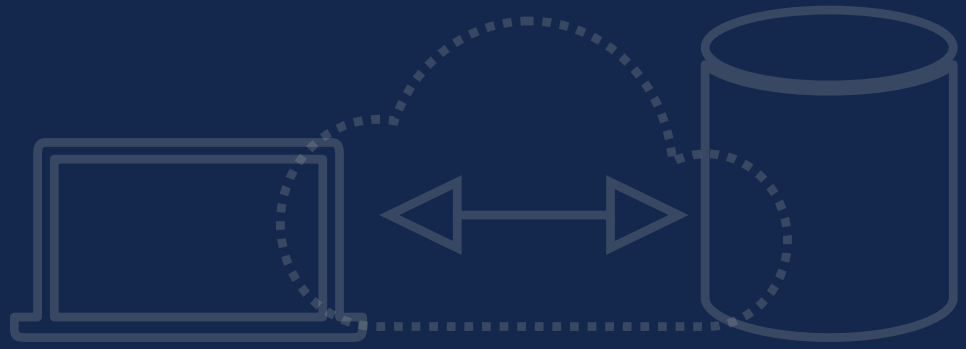


REST Overview Redux

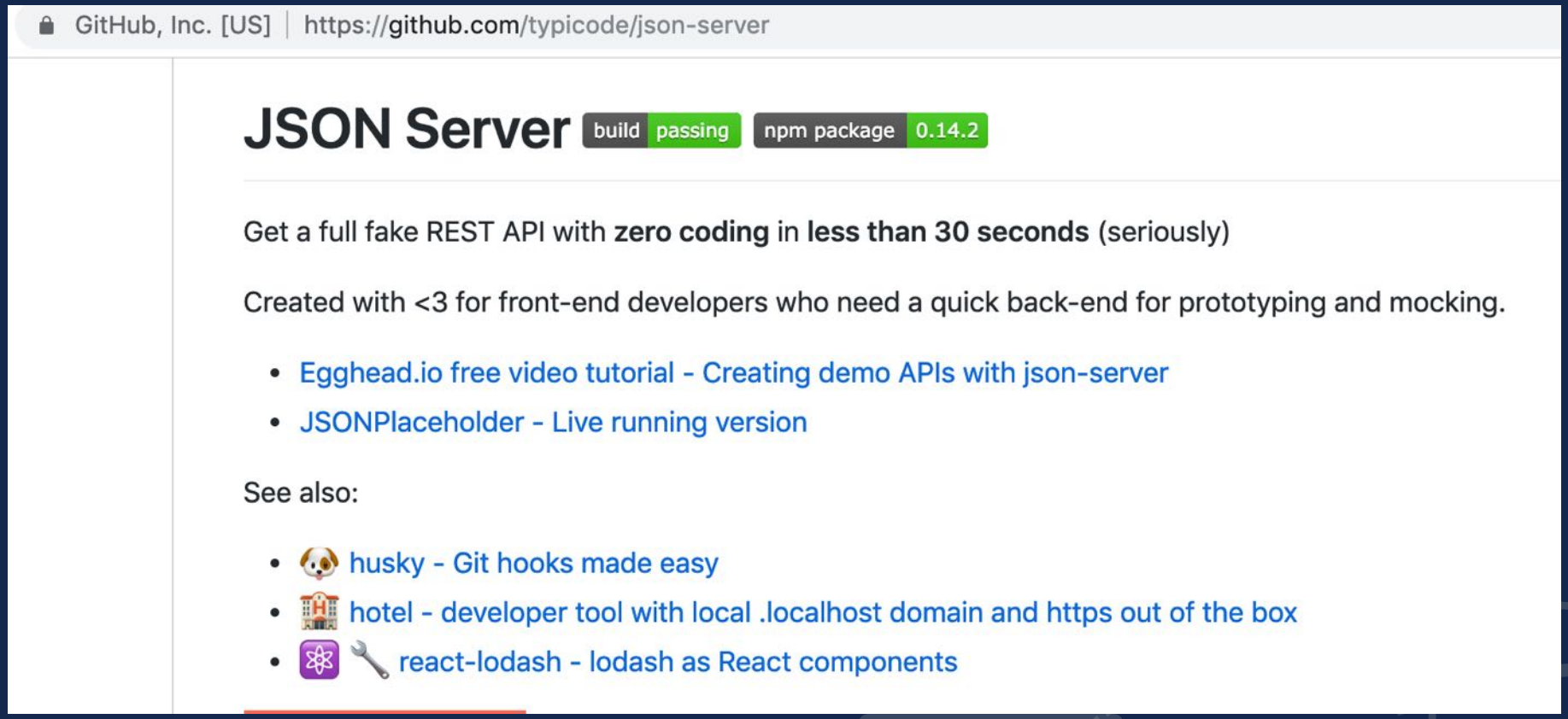


REST is Not the Backend

- It is the API to the backend!
- The database really doesn't need to be anything in particular
  - Structure in memory
  - File(s) in filesystem
  - Database
    - SQL (Relational Style)
    - NoSQL (Document Style)
  - The end point itself may encapsulate many systems themselves that do other things



REST Mocking



POSTMAN Demo

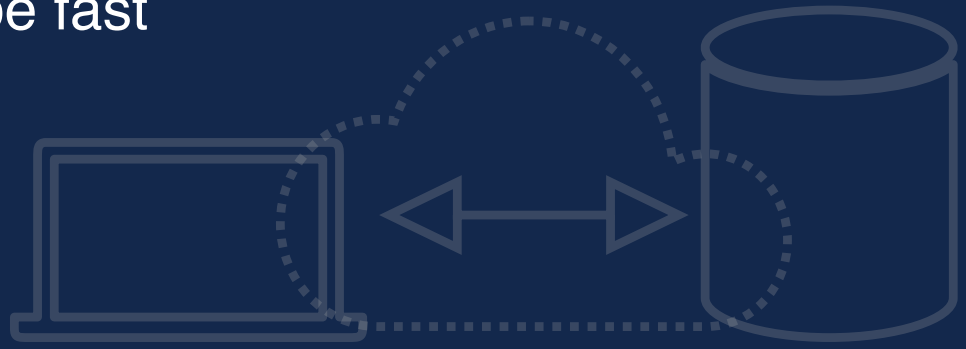
- https://github.com/typicode/json-server

Why Mock?

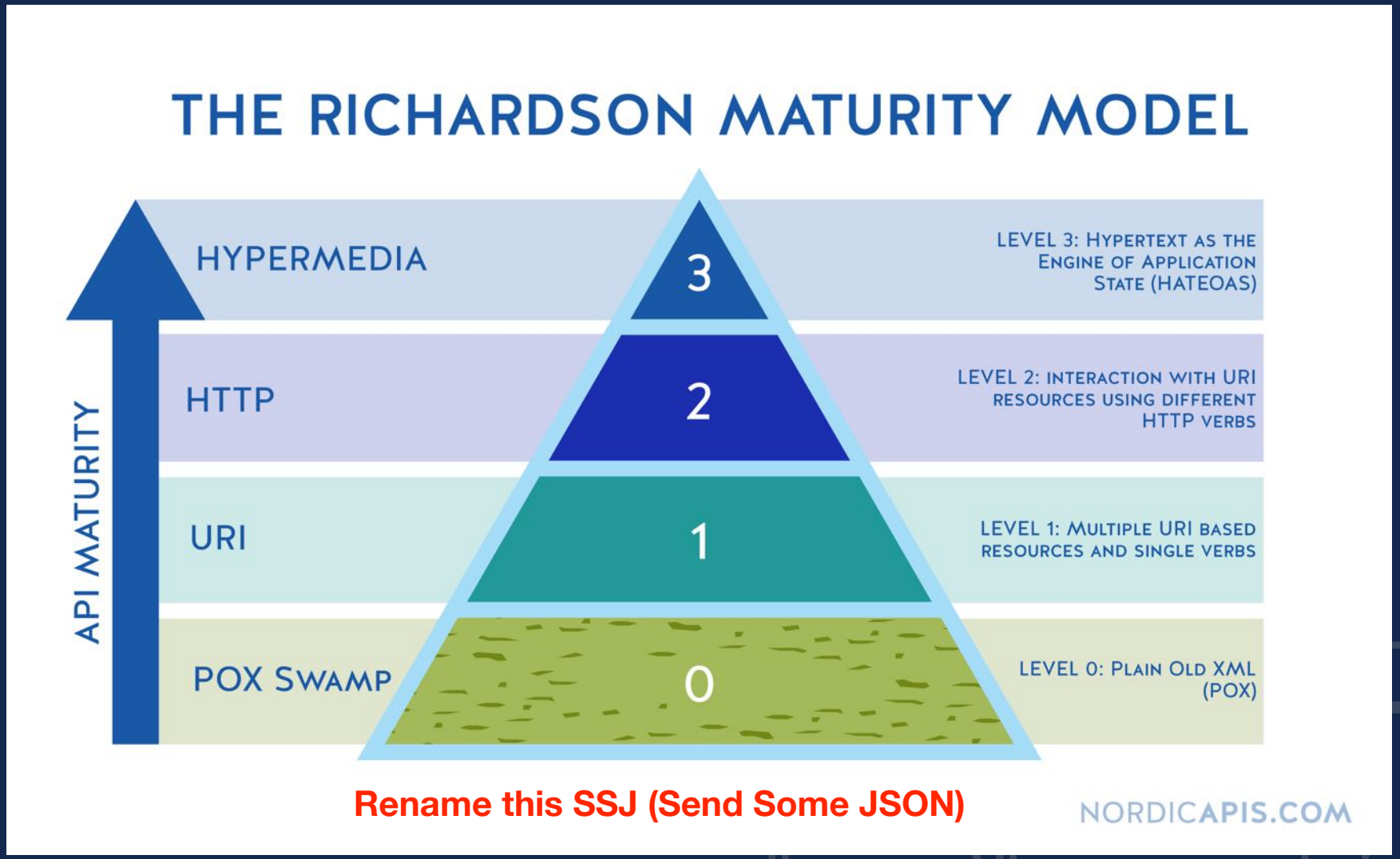
- Front End First Development! - UI Down
  - If we can count on one thing is that satisfying users isn't necessarily that easy
  - They often don't know what they want until they "see it"
  - Mock your UI (sometimes on paper, get into context quickly), test, and iterate until matches need
    - Prototypes probably need mock data and interactions and you don't want to code all that if it will change!

Why Data Model

- Model the data first to drive the design - Data Up
- Allows us to get the backend right which may help for performance, space efficiency, and ultimately cost (delivery and of course maintenance)
- But…
  - Backend is private, we can change it without people knowing
  - What we do here isn't noticeable unless it gets slow (could be at scale) or is buggy (we did a bad job)
  - It is expected to work right and be fast



Climbing API Pyramid

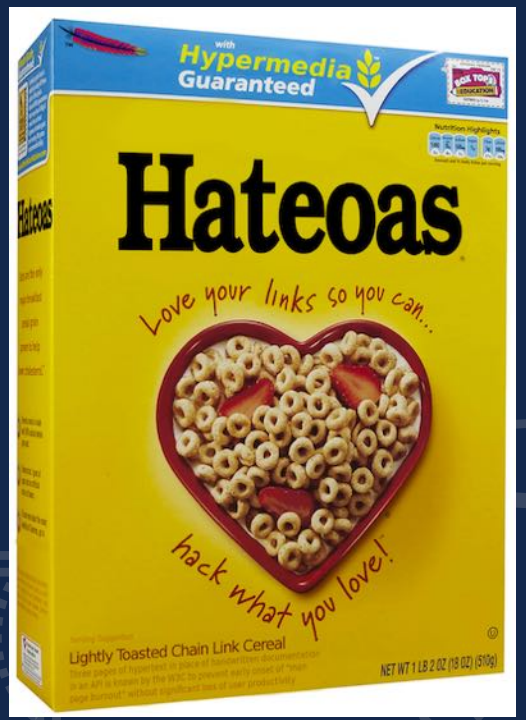


Rename this SSJ (Send Some JSON)

https://martinfowler.com/articles/richardsonMaturityModel.html

Go Higher!

- Hateos - Hypermedia As The Engine Of Application State
- Endpoints return data + links which lead to discovery and further actions upon the data
- Useful for public APIs so that the endpoint supports interactive development



https://en.wikipedia.org/wiki/HATEOAS

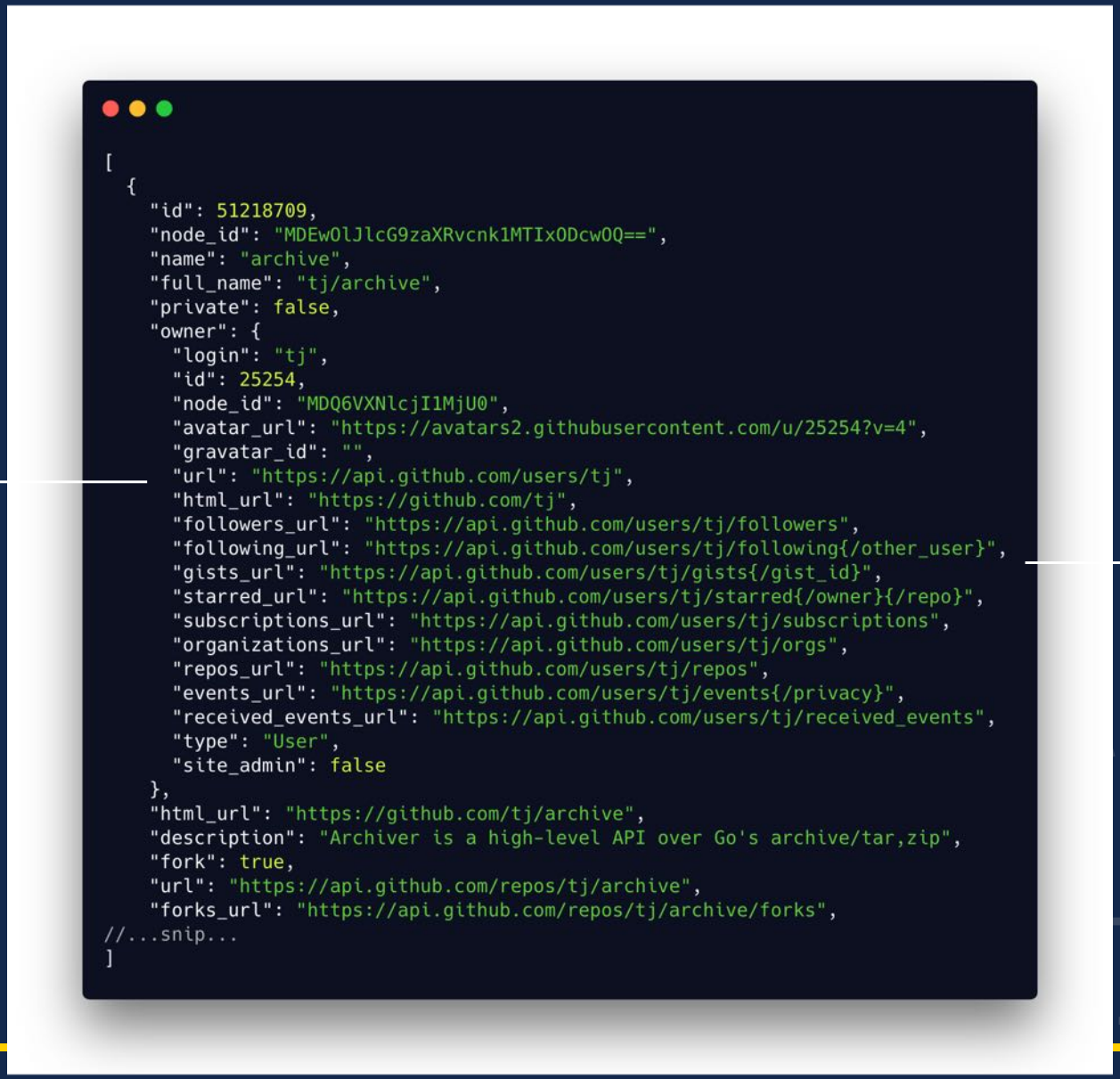
Hateos-ish


```
{
  "login": "tj",
  "id": 25254,
  "node_id": "MDQ6VXNlcjI1MjU0",
  "avatar_url": "https://avatars2.githubusercontent.com/u/25254?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/tj",
  "html_url": "https://github.com/tj",
  "followers_url": "https://api.github.com/users/tj/followers",
  "following_url": "https://api.github.com/users/tj/following{/other_user}",
  "gists_url": "https://api.github.com/users/tj/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/tj/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/tj/subscriptions",
  "organizations_url": "https://api.github.com/users/tj/orgs",
  "repos_url": "https://api.github.com/users/tj/repos",
  "events_url": "https://api.github.com/users/tj/events{/privacy}",
  "received_events_url": "https://api.github.com/users/tj/received_events",
  "type": "User",
  "site_admin": false,
  "name": "TJ Holowaychuk",
  "company": "Apex",
  "blog": "https://apex.sh",
  "location": "London, UK",
  "email": null,
  "hireable": null,
  "bio": "Founder of Apex\r\nhttps://apex.sh, a non-startup.\r\nmedium.com/@tjholowaychuk •
twitter.com/tjholowaychuk • tjholowaychuk.com",
  "public_repos": 274,
  "public_gists": 544,
  "followers": 38870,
  "following": 46,
  "created_at": "2008-09-18T22:37:28Z",
  "updated_at": "2019-01-17T19:06:32Z"
}
```


Follow this

https://api.github.com/users/tj

Hateos-ish Contd.



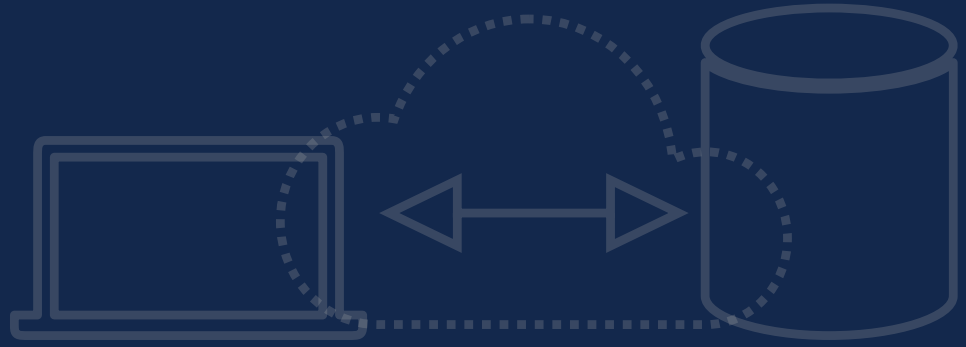
```
[
  {
    "id": 51218709,
    "node_id": "MDEwOlJlcG9zaXRvcnk1MTIxODcwOQ==",
    "name": "archive",
    "full_name": "tj/archive",
    "private": false,
    "owner": {
      "login": "tj",
      "id": 25254,
      "node_id": "MDQ6VXNlcjI1MjU0",
      "avatar_url": "https://avatars2.githubusercontent.com/u/25254?v=4",
      "gravatar_id": "",
      "url": "https://api.github.com/users/tj",
      "html_url": "https://github.com/tj",
      "followers_url": "https://api.github.com/users/tj/followers",
      "following_url": "https://api.github.com/users/tj/following{/other_user}",
      "gists_url": "https://api.github.com/users/tj/gists{/gist_id}",
      "starred_url": "https://api.github.com/users/tj/starred{/owner}{/repo}",
      "subscriptions_url": "https://api.github.com/users/tj/subscriptions",
      "organizations_url": "https://api.github.com/users/tj/orgs",
      "repos_url": "https://api.github.com/users/tj/repos",
      "events_url": "https://api.github.com/users/tj/events{/privacy}",
      "received_events_url": "https://api.github.com/users/tj/received_events",
      "type": "User",
      "site_admin": false
    },
    "html_url": "https://github.com/tj/archive",
    "description": "Archiver is a high-level API over Go's archive/tar,zip",
    "fork": true,
    "url": "https://api.github.com/repos/tj/archive",
    "forks_url": "https://api.github.com/repos/tj/archive/forks",
//...snip...
]
```

Back

Deeper

API Consumption

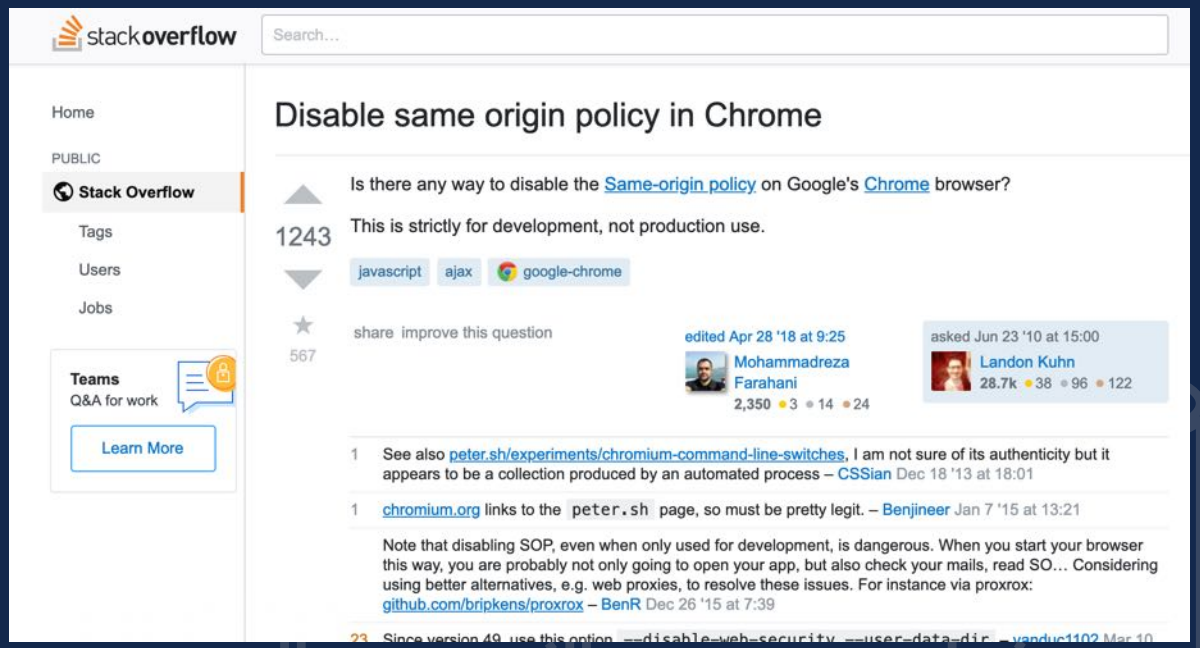
- Discovery
- Testing
  - Tools: Browser, Postman, Paw, https://insomnia.rest/, etc. etc. etc.
- Cross Domain Drama
  - Same Origin Policy
    - JSONP - JSON with Padding
      - Or how to run raw script responses like a crazy person!
    - Self Proxy
    - CORS



What the…!

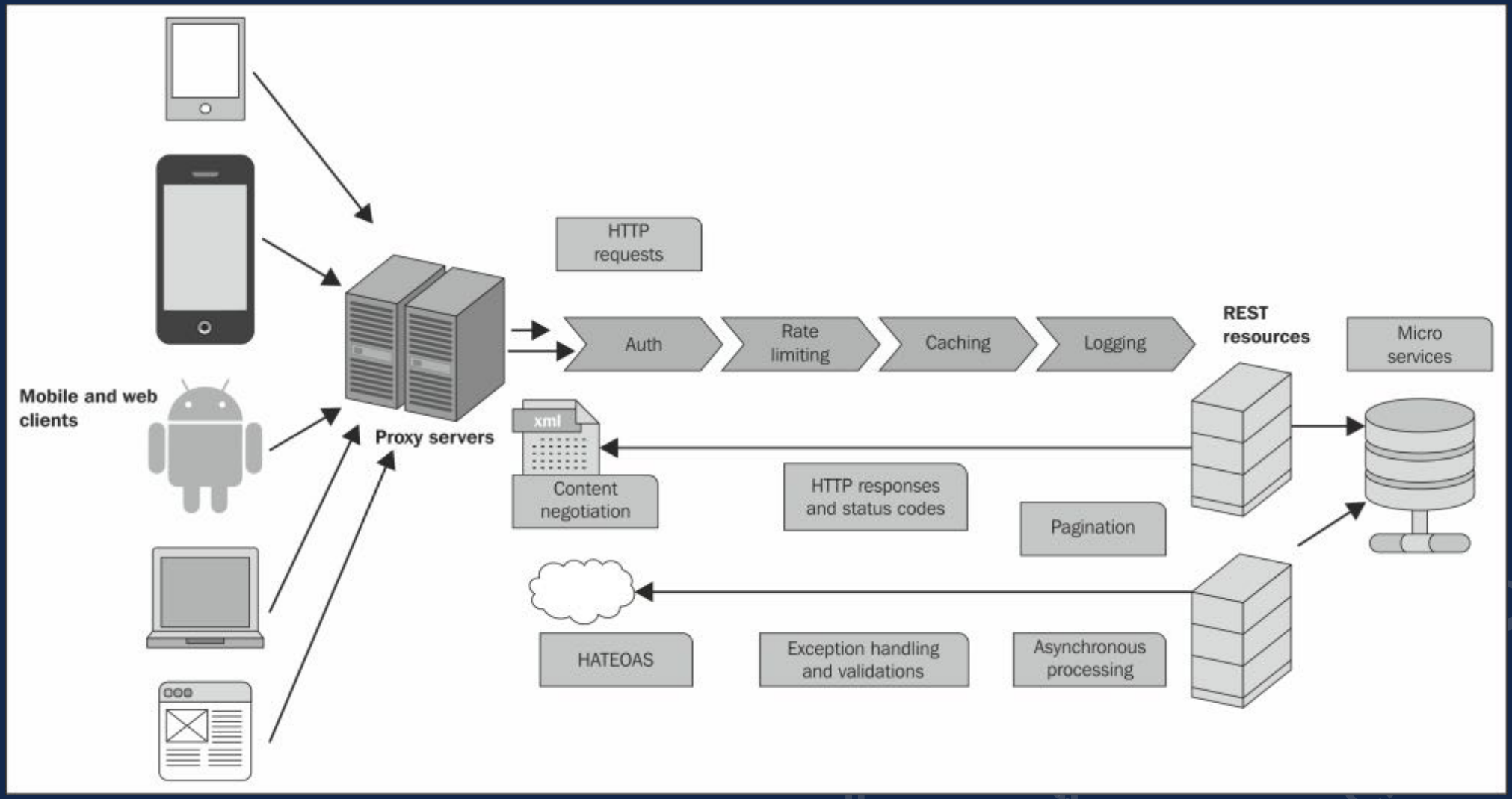
Server Folks

Client Folks

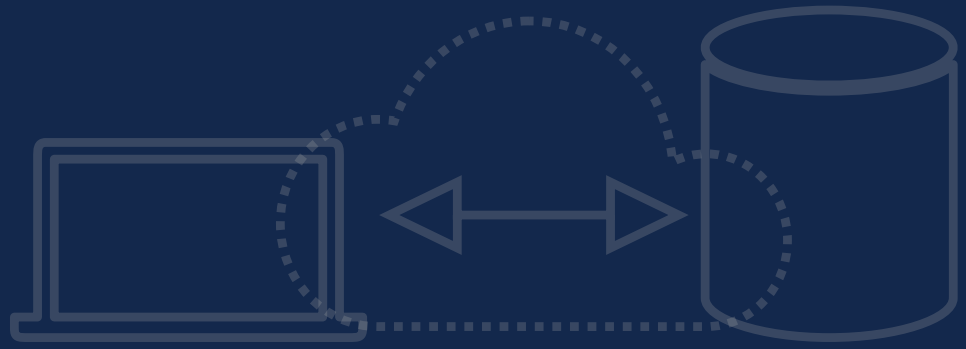


Do the Work or Pay the Price Someday

&*#@ Nothing is ever easy!

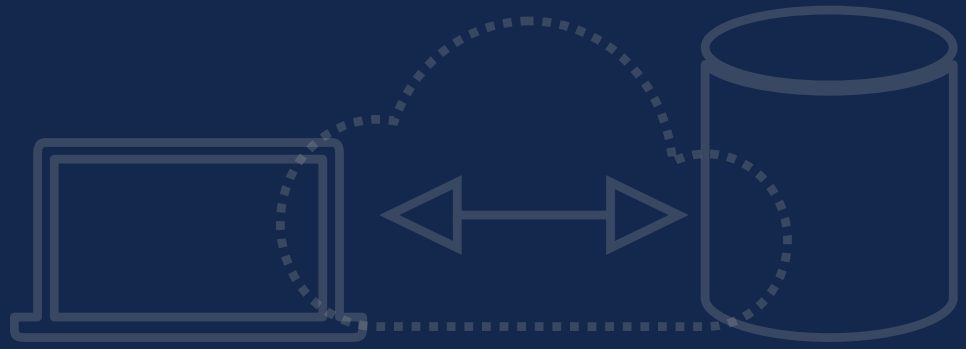


Building Our Analytics App



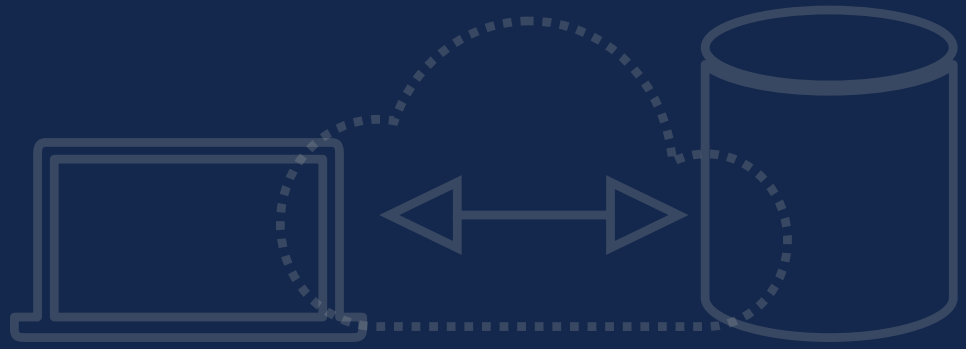
Let's Go Look at a Diagram

- This was already detailed
- First the questions / hypothesis
  - What are the technographics of our users - inform future and current tech choices
  - Is our site/app slow?
  - Does our site have errors?
  - What do people do on the site? (optional)



Some Philosophy

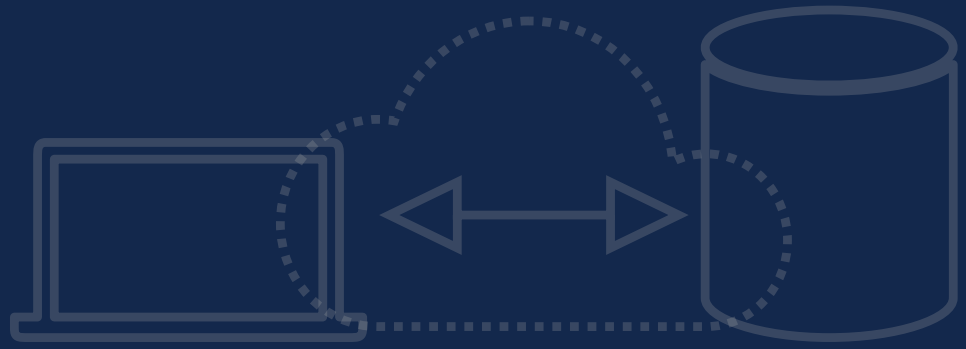
- Collect just what we need to answer the questions
- Collect more than what we need in case other questions come up



Data Hoarder!

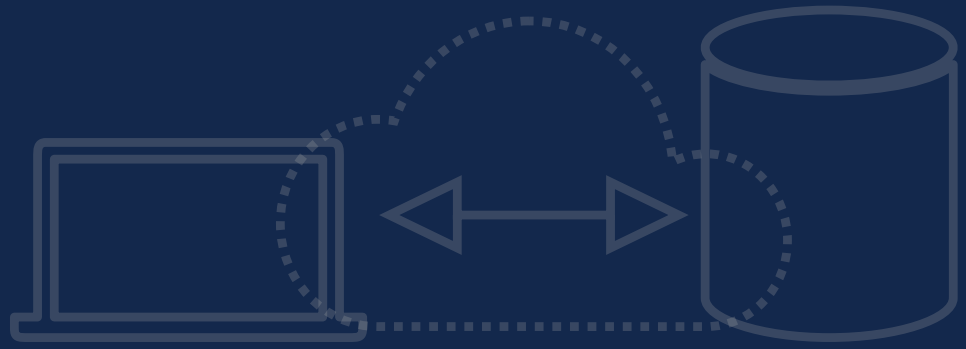
Tech Overview

- Collection of client data
  - Static Data - browser type (version, mobile/desktop) screen resolution, load time performance and connection rates
  - Run Time Data
    - Performance
    - JS errors - window.onerror (optional but encouraged)
    - Events - Passive vs Defined
      - Passive: arbitrary user events (click, scroll, etc.)
      - Defined: myTrack('eventname', 'value') - (optional but encouraged)



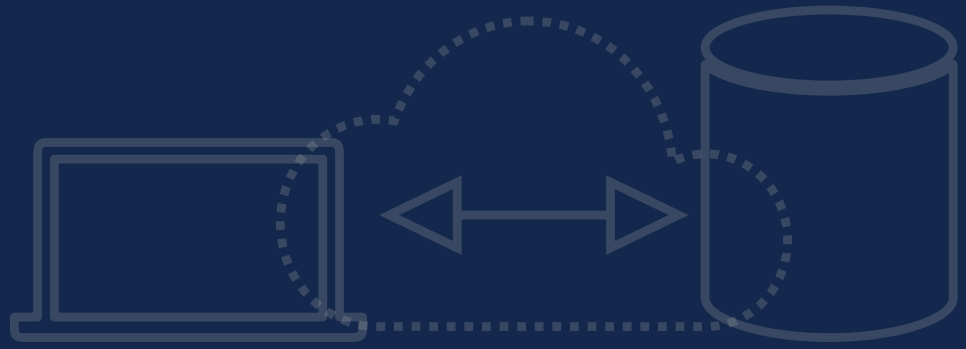
Overview Contd.

- Storage of collected data
  - Ingest and save immediately
  - Ingest, clean and then save
  - DB type - SQL vs NoSQL
- Query of collected data
- Reporting Backend



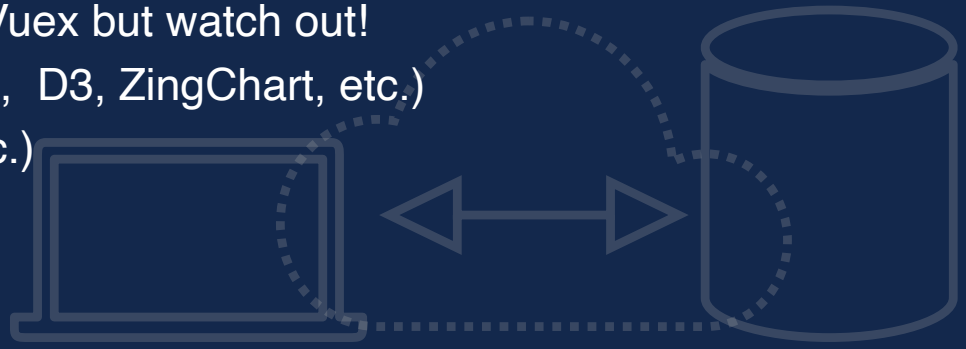
Overview Contd.

- Backend
  - Authentication for Reporting Users
    - Simple username/password vs OAuth style
    - Buy / Find / Build Trade-off
    - User Management
  - Basic Navigation to Reporting Pages
    - Tables / Grids
    - Charts
      - Static vs Dynamic (Static Images vs JS Based)
    - Distribution
      - Print, Email, or Share



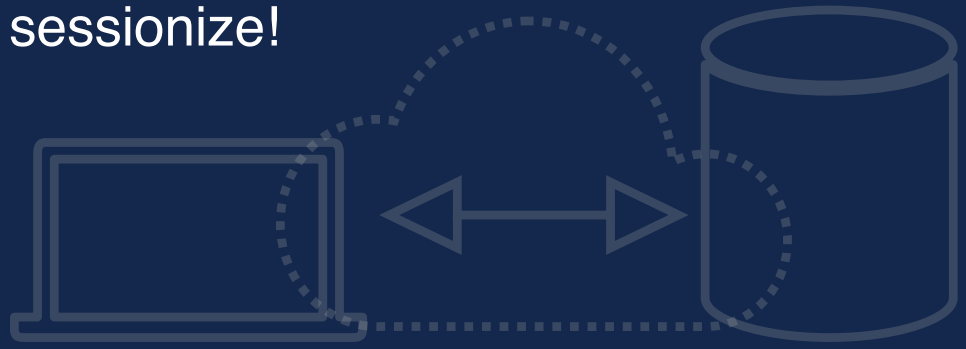
Overview Contd.

- Collector Script - Client Side JS
  - Vanilla JS, end user facing, don't want to rewrite or replace often
- Reporting Backend
  - This Year - Raw servers (Node or PHP)
  - Previous Year - Cloud functions, Firebase
  - Last Year: MySQL or MongoDB
  - Previous Years: Cloud NoSQL or SQL
  - Let's discuss all the problems (sessionid, caching, the front-end/backend issue)
- Reporting Frontend
  - Traditional Page Style vs SPA Style
    - 1 page/state = 1 URL vs Single URL + # routes
  - HTML + CSS + JS or React/Redux or Vue/Vuex but watch out!
  - Charting : Images vs JS (Highcharts, ChartJS, D3, ZingChart, etc.)
  - Gridding : Table + DOM (agGrid, ZingGrid, etc.)



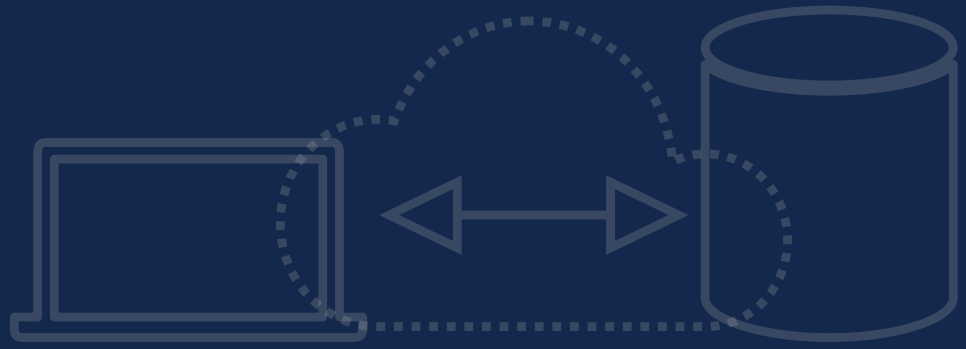
Collector Script

- Design Thoughts
  - Write this in Vanilla JS
    - No dependencies = no frameworks
    - Should be small
    - Avoid blocking page render
    - Keep your payload small
    - Think resiliency
  - Serve it from a set location
    - http://yourserver.com/client/collector
    - This should return the script and sessionize!



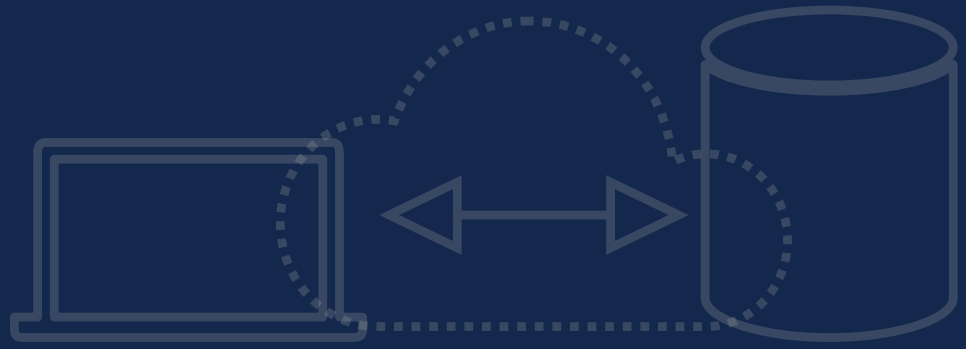
Collector Script

- You may decide to do the send
  - A lot: full objects (navigator, screen, etc.), full event stream of clicks, etc.
  - A little: just the values you are interested in
- You may encode as you like
  - JSON, query string, some compact format, …



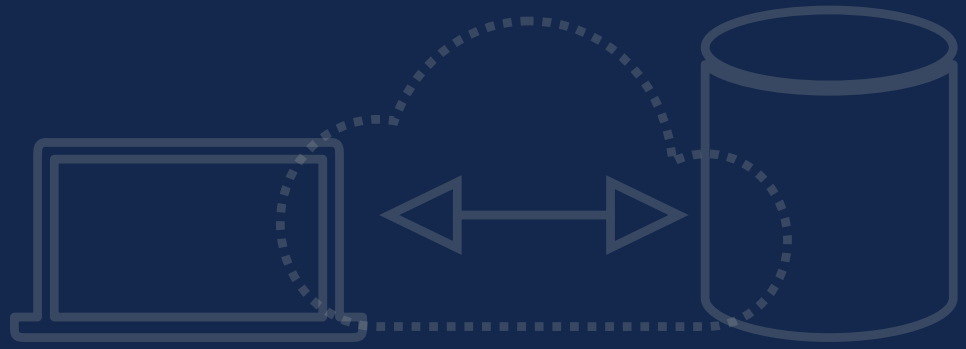
Basic Collection Testing

- When you do manual testing use your Chrome Dev Tools to change the performance profile of your browser to get different data. Use other browsers types (Firefox, Safari, Edge, Opera, etc.) as well to get different data.



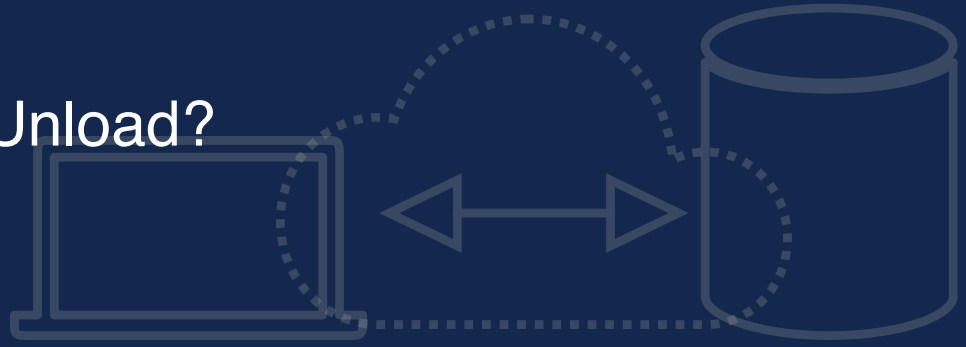
Simulated Collection

- Your basic testing will likely not build a meaningfully large enough data set
- To rectify data size write a generation script to make randomized packets to send to your end points.
- You may use Node, PHP or a Bash script in conjunction with an HTTP mechanism like Curl to easily send data to your endpoints.
- Make at least 1000 random records for testing purposes
- Do this to experiment with rate issues and to build a meaningful data set



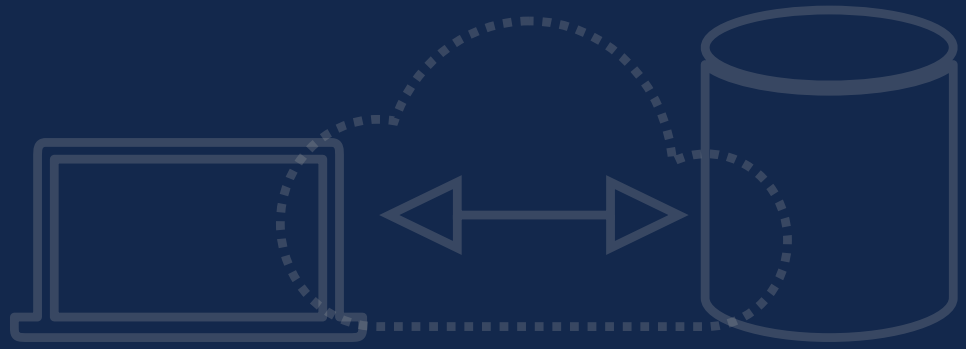
Client Comm APIs

- If you can make an HTTP request in HTML or CSS use with JS to send data
  - Traditional 1-way: <img>, <script>, <style>, <iframe>
    - Query string only, except iframe
  - Traditional 2-way: <img>+cookie, <iframe>
  - XMLHttpRequest (aka XHR, aka Ajax)
  - Fetch API
  - navigator.sendBeacon
  - Web Sockets
- Considerations: Script off? Fast Unload?



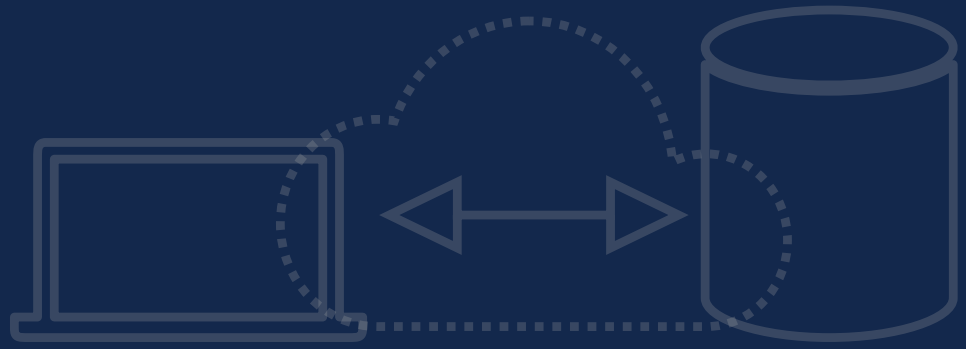
Data Storage

- Store your data either in a MySQL or MongoDB database hosted on your same server
- Design your data structure after you understand what you are collecting and what submitted packets look like
- You may directly insert your data and avoid an ingestion/clean step for simplification. In a real world scenario we likely would have such a layer.



Backend App

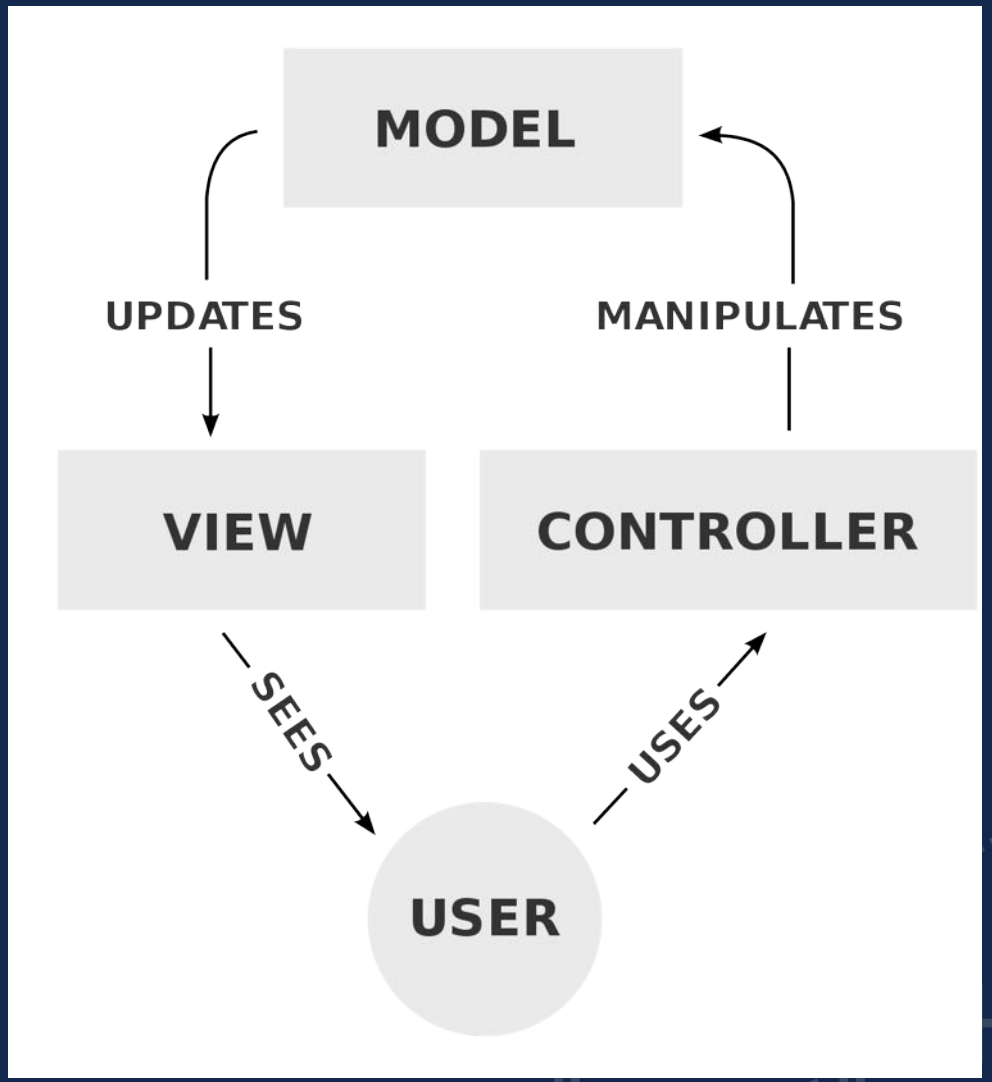
- Authentication Layer
  - Node: use Express + Some Auth Layer like passportjs.org
  - PHP: use Laravel + Its Auth or Larvel Passport
- Avoid rolling your own login
- Access at http://yourserver/reporting/login



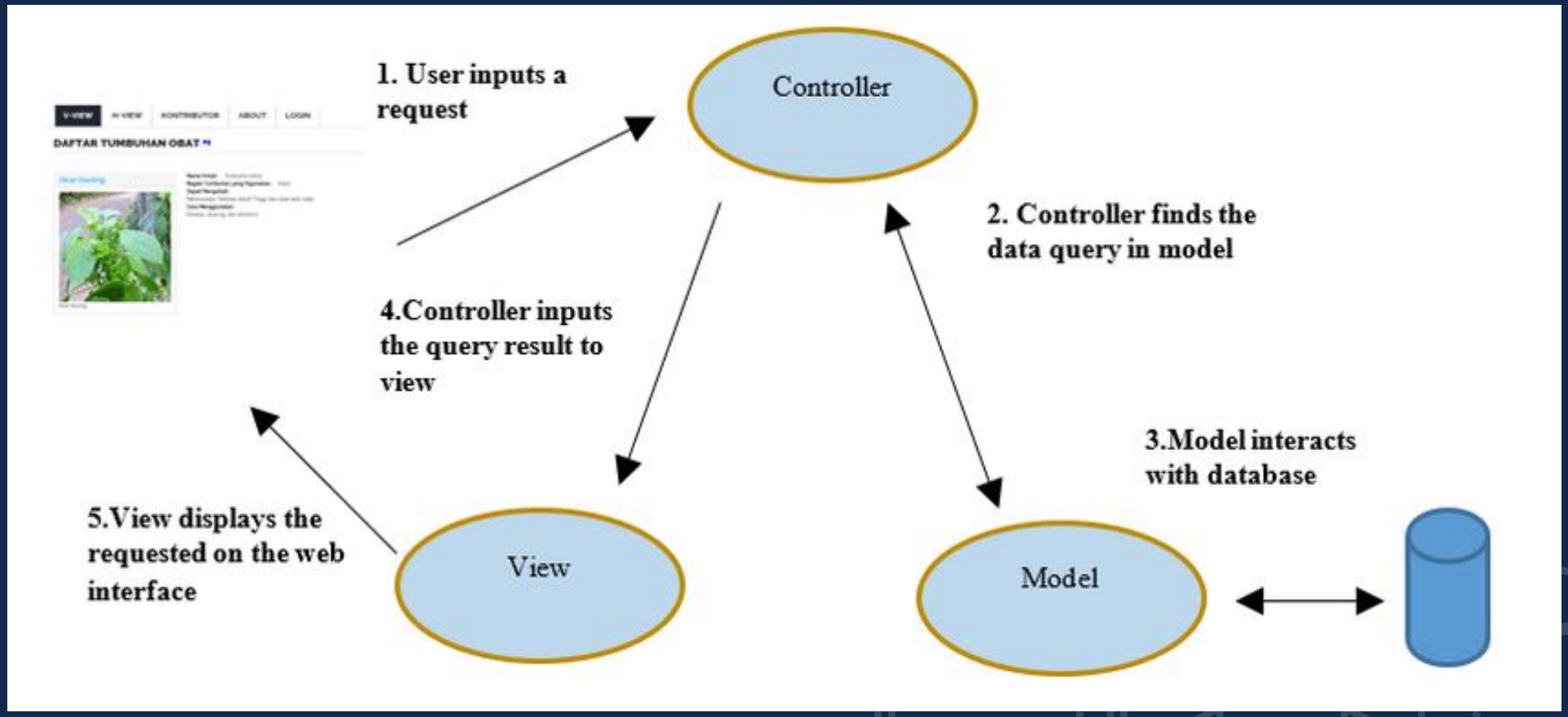
App Style

- Your reporting app may be either a server-side style app (traditional 1 page - 1 URL), SPA style, or a resilient hybrid style.
- You must employ an MVC style design
- Client Side app frameworks are limited to Vanilla JS, jQuery, VueJS or React.
- You are free to use a CSS framework of your choice for styling
- The following slides overview the MVC patterns you may use

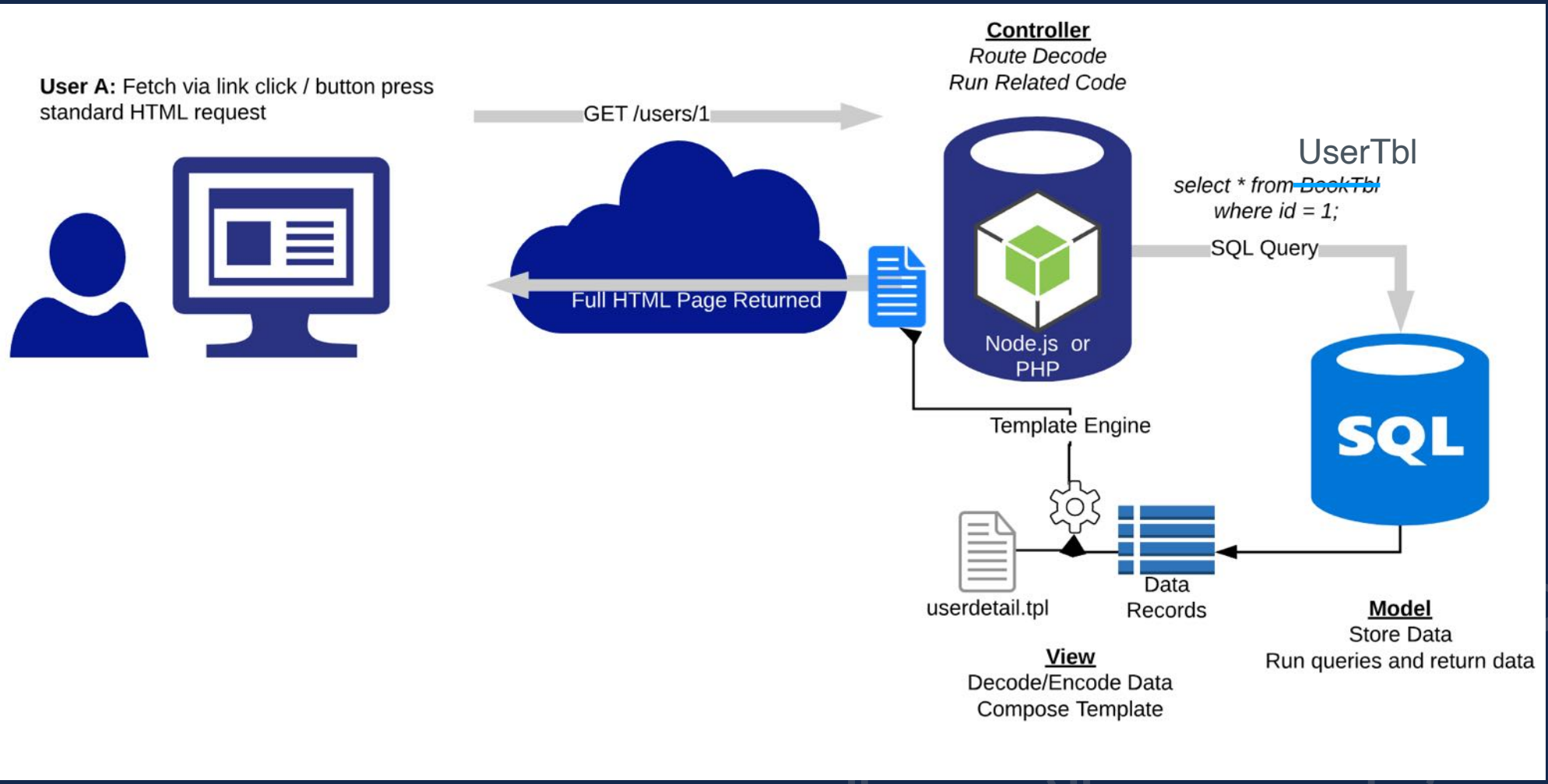
MVC



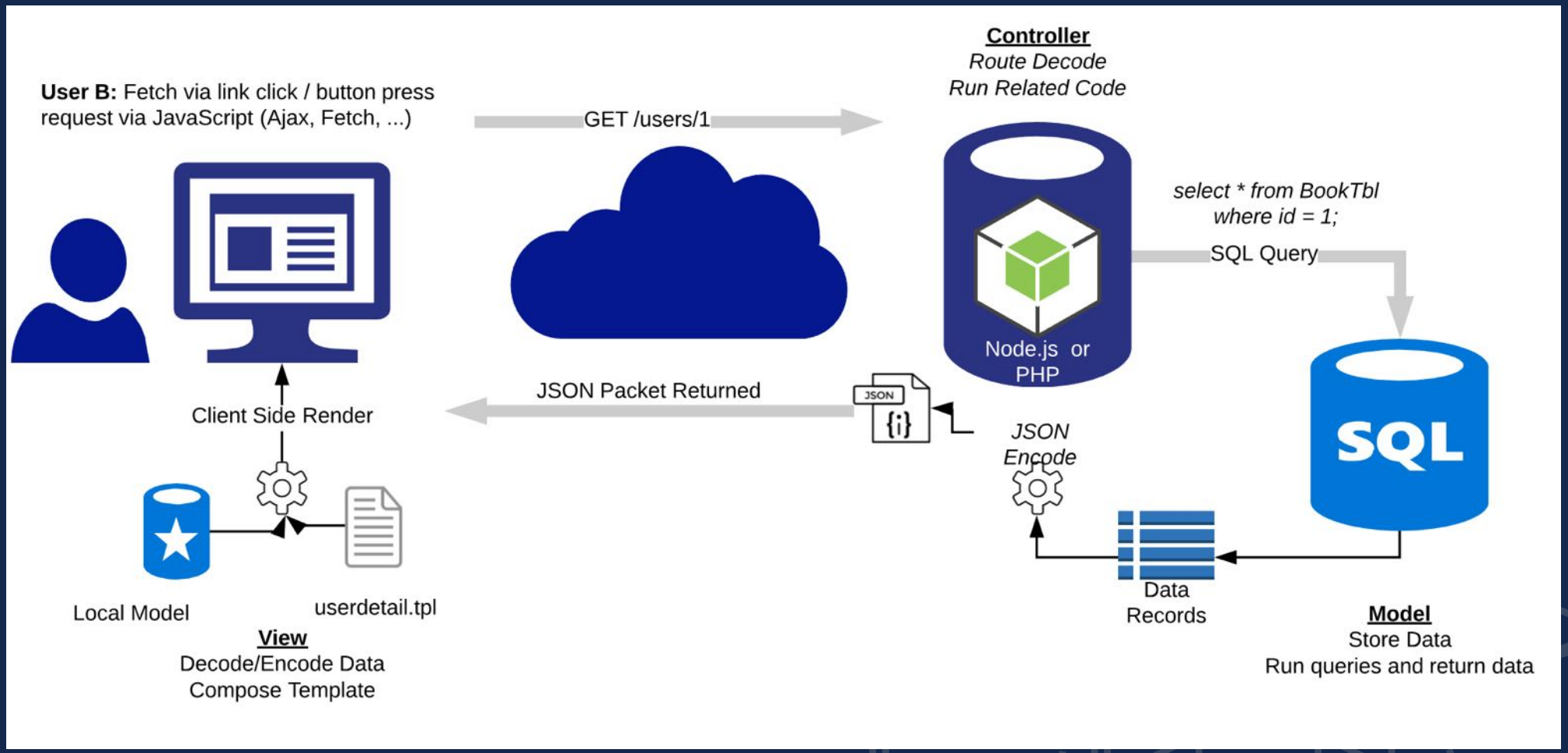
MVC Closer


1. User inputs a request
Controller
2. Controller finds the data query in model
4.Controller inputs the query result to view
3.Model interacts with database
5.View displays the requested on the web interface
View
Model

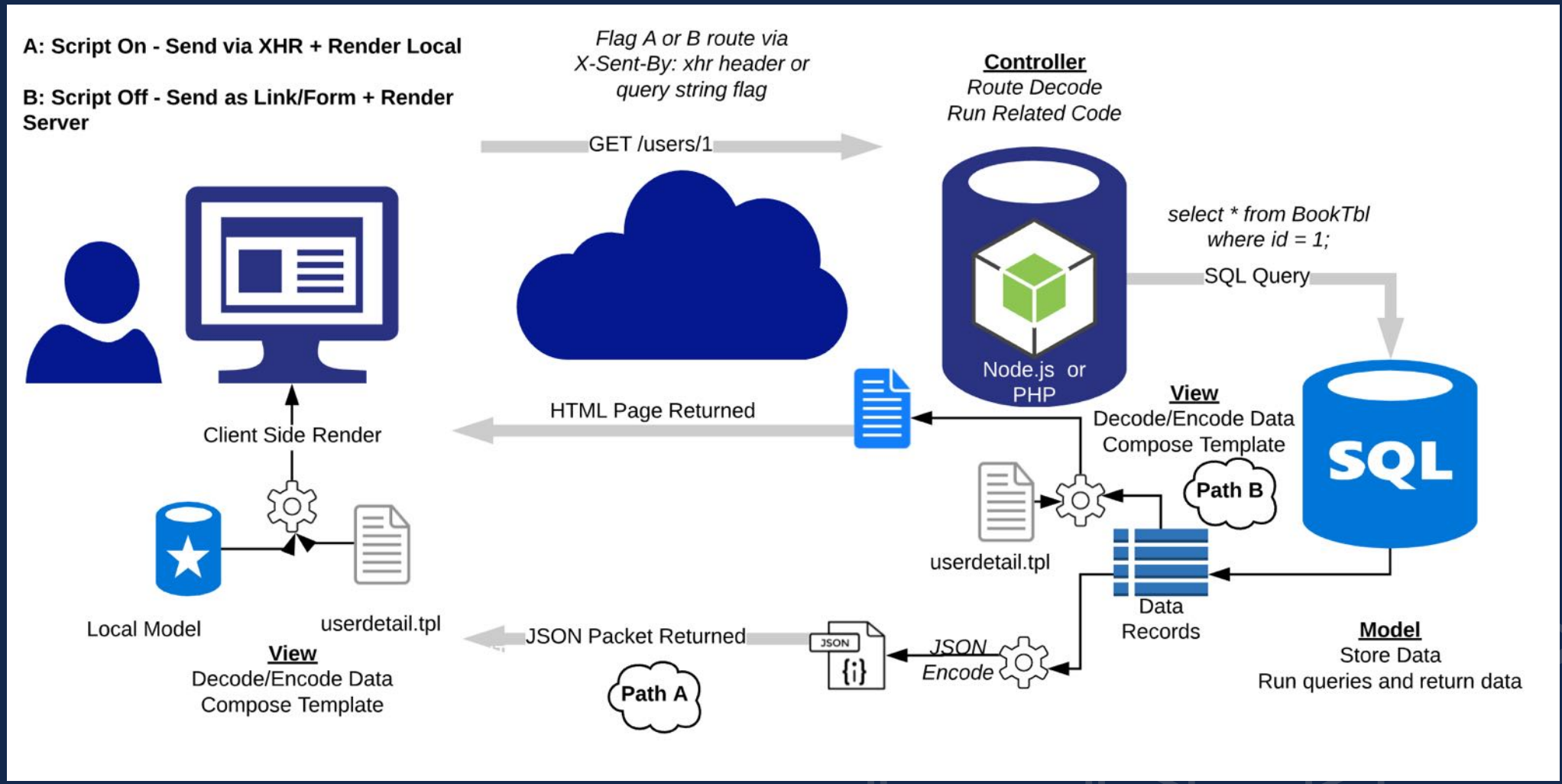
Ex: Server-Side MVC



Ex: Client Side MVC

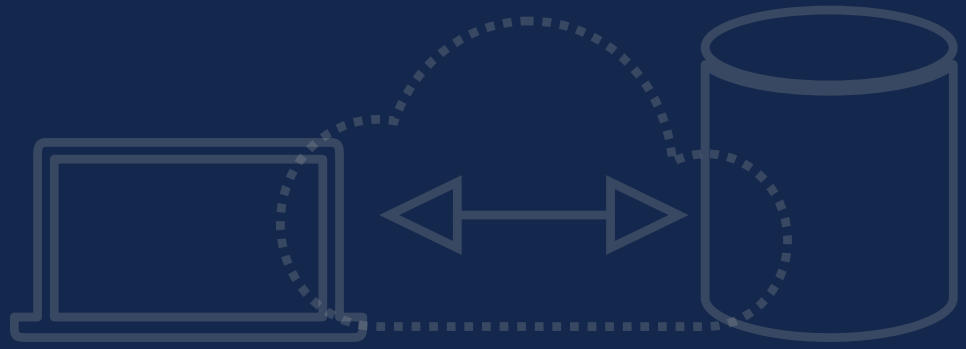


Ex: Adaptable MVC



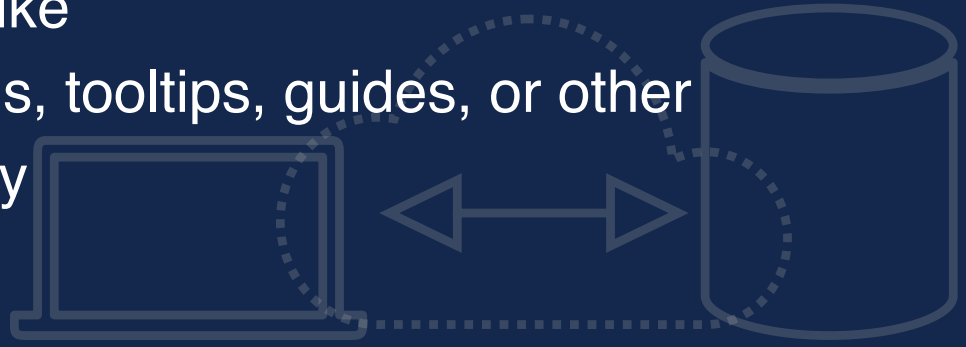
Reports

- Often basic CRUD systems have a reporting feature.
- Our online analytics has many options for reporting such as things like
  - Device, Browser and Display dimensions
  - Performance (Time Series)
  - Events, Errors, etc.



Visualization

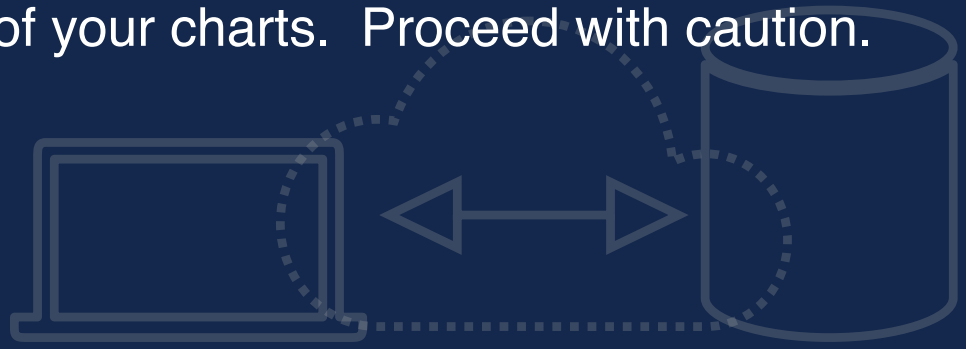
- Useful reports should provide charts to consume the data more easily
- Pie charts and Bar charts should be employed on static data
- Grids + line charts or bars on time series data plus appropriate pie charts if necessary
- Event visualization is at the discretion of the student as extra credit
- Visualization should use a JavaScript library
  - You may use something low level like D3 and roll your own visualizations or you may employ a charting specific library like HighCharts or ZingChart as you like
- You should consider doing drill downs, tooltips, guides, or other interactive features for data discovery



Data Sharing - The Taking Action Part!

- Given the importance of taking analytic data and summarizing it for use by others your system ought to provide a rudimentary way to share data
  - Given our short time this optional, but if you did you may either provide a PDF file or a generated public link of a shared report.
  - You could easily provide a way to email the shared data to some interested 3rd party by clicking a button and filling in a name, subject and emailing it.
  - You also could experiment with sending to Slack if you like for extra credit

- Note: Sharing may require static output of your charts. Proceed with caution.



Conclusions

- The CRUD, REST and MVC patterns are useful in the context of our project and common in server-side focused web development
- Be careful with confusing the basic ideas of the patterns which are pretty straight forward with mastery as it can explode in complexity quickly
- Approaching these ideas one piece at a time is a less risky way to proceed on your project as well as in future work efforts.