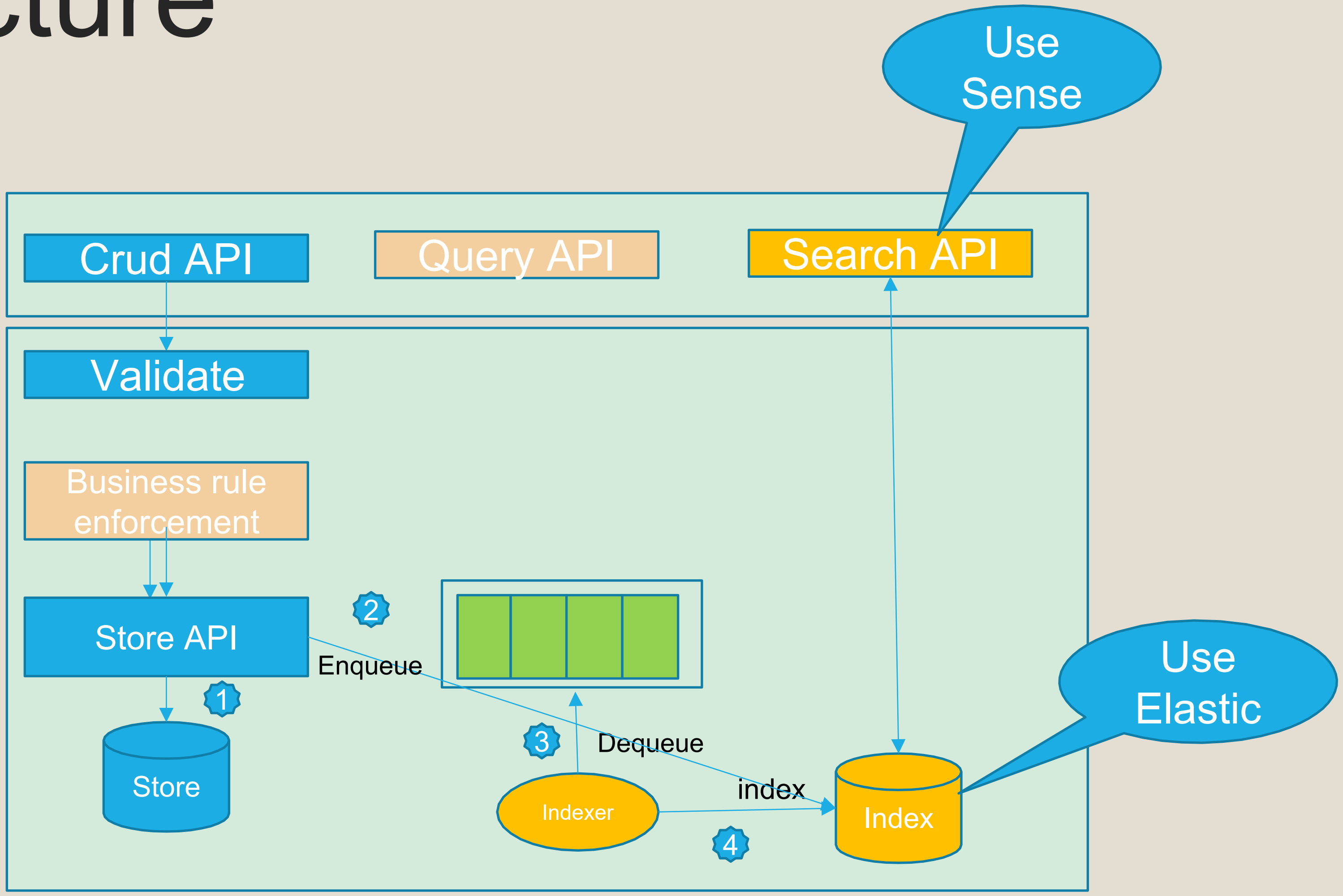
ADVANCED TOPIC IN BIG DATA

10/22/16

Architecture



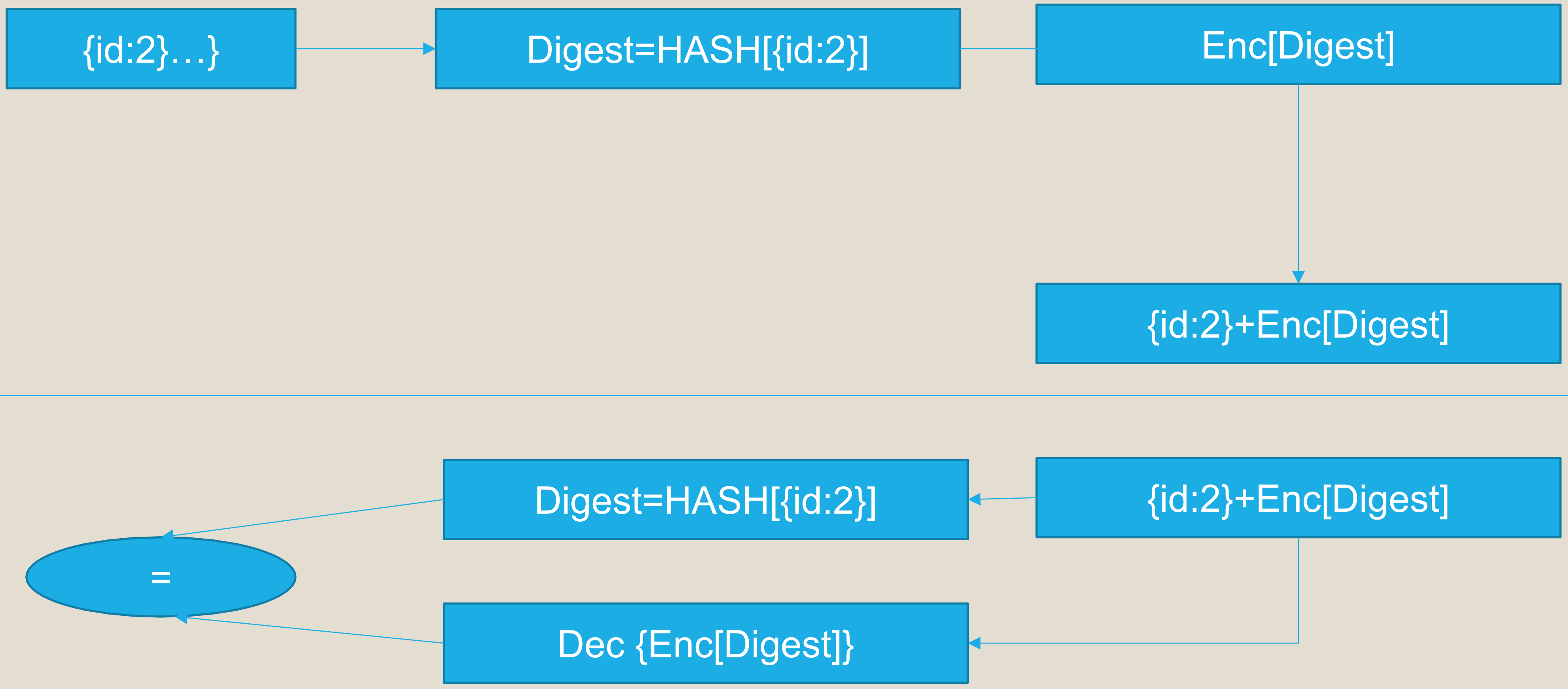
Prototype outline:

- Rest API that can handle any structured data in Json
- Rest API with support for crud operations, including merge support, cascaded delete
- Rest API with support for validation
- Json Schema describing the data model for the use case
- Advanced semantics with rest API operations such as update if not changed
- Storage of data in key/value store
- Search with join using Elastic
  - Parent-Child indexing
- Queueing
- Security

Catching up on old topics

- Implementing merge semantics
- Etag implementations
- Security implementation progress
  - Revisit security implementation using tokens

Asymmetric Crypto



Join and elastic search

- How to implement to join with elastic search?
  - Pros
  - Cons

- Demonstrate this approach using sense

http://fideloper.com/api-etag-conditional-get
They use MD5 having to calculate the Etag.
http://stackoverflow.com/questions/415953/how-can-i-generate-an-md5-hash

Fulltext search

- Basic concepts:
  - Indexing:
    - Is the process of creating an index.
    - An index is defined as a collection of fields. Each field can be either single value/multivalued, have a type, stored, indexed, required, can be associated with different tokenizer's/analyzers
    - Dynamic fields is a very useful feature
    - An index contains a collection of documents.
    - A document is a collection of property (field) / value pairs
  - Searching
    - Is the process of discovering a document in an index that meets certain criteria's
    - the criterias are specified using fields that are found in a document

Query samples

- find all documents containing name:jeff
- find all documents containing name:jeff and age:30 (Or any other logical relation, e.g. or, not, and)
- find all documents created after 9-16-2016
- find all documents of type plan
- find all documents of type pla*; E.g.; plans, planning, planner etc.
- Find all the Unique terms of the field "type" in the system
- Counts:
  - how many times a certain value occurs in the index
- Aggregates:
  - Max, Min, Average, Sum, percentiles, etc.
- How many cameras are on sale between 50 and $100?

Faceted queries

- Is the bucketing of search results into buckets based on terms in the index
- Useful for determining the unique terms for a field and returns a count for each of those terms.
- Makes it easy to explore search results
- Faceting example is found here:
  - https://lucidworks.com/post/faceted-search-with-solr/

Faceting..

- Field faceting – retrieve the counts for all terms, or just the top terms in any given field. The field must be indexed.
- Query faceting – return the number of documents in the current search results that also match the given query.
- Date faceting – return the number of documents that fall within certain date ranges.

Filter queries

- Used to filter the results of the previous query
  - Often used to implement drill down into search results
- When filter query is added to the previous query, its effect is to exclude results that do not match the filter

- Example:
  - Return <u>all cameras </u>by manufacturer and their count
    - /query?q=camera facet.field=manu
  - Return <u>all cameras in this price range </u>by manufacturer and their count
    - http://localhost:8983/solr/query?q=camera &facet.field=manu &**fq=price:[400 to 500] (fq is filter query)**

Elastic Search

- Getting started:
  - https://www.elastic.co/guide/en/elasticsearch/guide/current/getting-started.html

Homework

- Demonstrate an example of join queries using elastic search. This is due 10/22

References

- https://cwiki.apache.org/confluence/display/solr/About+This+Guide
- https://lucidworks.com/post/faceted-search-with-solr/https://www.elastic.co/guide/en/elasticsearch/guide/current/denormalization.html
- Getting started:
  - https://www.elastic.co/guide/en/elasticsearch/guide/current/getting-started.html

Oauth

INFO 7255

Use cases for security

- Washington Post/Boston Globe: paywall, tiered-based subscription
- Flash sales
  - Can I prevent bots from sweeping up all inventory?
  - Can my application hold up against excessive demand?
    - Digital Waiting Room
- Authenticated access
  - Quota and throttling
- Anonymous access
  - throttling
- Bots Access
  - Good bots versus bad bots

Security requirements

- Authorized access against API
  - Only users authorized to access resources are allowed
  - Users able to see/edit their own plans
  - Users may read other plans, but no change them
    - Users may have certain access to this endpoint but not to the other one
- Anonymous browsing may be allowed
  - This is prior to user authentication
- App may not exceed certain requests per day/month: quota
- Apps that are making excessive number of requests need to be throttled
  - Digital Waiting Room

High-level Approach

- Client includes an authorization header
  - The value of the header is a token
- API uses the authorization header value (token) for authorization and authentication
  - Client signs token
  - API verifies token

Key design questions

- What is the overall approach for securing APIs?
  - Bearer Tokens
- What is the token structure?
  - JWT
- How are token generated?
  - How are they signed?
    - By an Idp
- How are tokens verified?
  - Authenticate the signer of the token
- Security crypto: Asymmetric? RS256
- Security guarantees: Authentication, non-tampering

First approach: API keys for securing access by apps

- High level flow:
  - Each app is granted a key at build time by the server
  - app includes key in every request that goes to server
  - Implications on quota and throttling?

OAUTH 1.0

- Username & Password

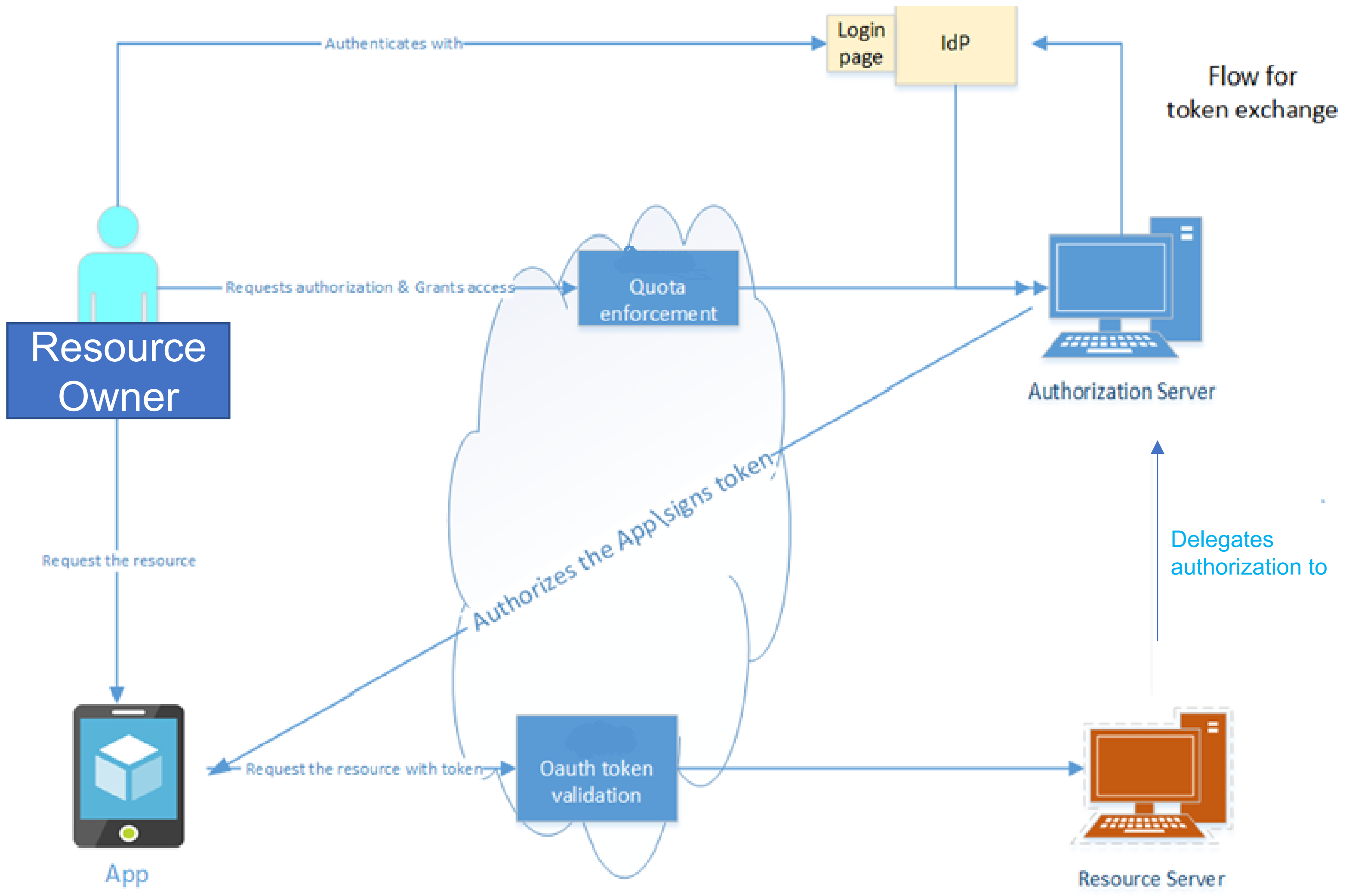
Industry accepted approach: OAUTH 2.0

- User downloads an app
- User authenticates with an IDP/Auth server
- User consents to give app access to user’s data
- IDP generates token
- App includes the token in the API calls

Public versus private app

- Public apps are those that cannot secure their credentials: single page application, mobile apps
- Private apps are those that can secure their credentials: any app running behind a firewall

oAUTH 2.0 Overview and Actors



Token Validation by Resource Server

- 1. Validate the structure of a JWT
- 2. Create an “allow list” that contains valid values for iss claim
- 3. Base64decode JWT header, payload
- 4. Retrieve alg and kid from Header
- 5. Retrieve iss from payload
- 6. Compare the value of iss to that stored in the “allow list”
    5. If iss value in allow list, use JWKS_URI to retrieve public key. Otherwise, signature invalid

    6. Verify signature

    7. Validate any other claims such as scope, aud, exp, etc.

Overview

- RFC OAUTH 2.0:https://tools.ietf.org/html/rfc6749
- JWT https://tools.ietf.org/html/rfc7519
- Example: https://dev.fitbit.com/docs/oauth2/

Oauth provider (Authorization Server)

- /register
- /Authorize
    - unsecure
    - Authorization code grant flow
        - Returns both access token and refresh token
        - Use for secure clients
    - Authorization code grant flow with PKCE
        - Use for unsecured client
    - implicit grant flow
        - Returns only access token
        - Use for unsecured client
- /Token
    - Secure
    - Exchange authorization code for a token
    - generate a new token from a refresh token

/register

- Input:
  - Client_type = confidential (private) or public
  - redirect_URI: https://
- Output:
  - client_ID, client_secret if client is confidential
  - Client_id for public

/Authorize

- The authorization endpoint must support "get"

- The supported query parameters are:
  - response_type
    REQUIRED. Value MUST be either "code“ or "token“
  - client_id
    REQUIRED. The client identifier obtained from the registration
  - redirect_uri
    Required. As described in Section 3.1.2.
    https
  - scope
    Required.
  - state
    Required

Authorization grant code flow example:

GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb &scope=read
HTTP/1.1

Host: server.example.com
•The response should have:
•HTTP status code should be set to 302:
•the redirect URI as the value of the location header.
•the query parameter code and its value, state and its value appended to the redirect URI.
•The code is required at all times. The state is required only if it has been present in the request.
•example:
•HTTP/1.1 302 Found
•Location: and that we should specify the location and the location it should contain it&state=xyz

error

- HTTP/1.1 302 Found
- Location: https://client.example.com/cb?error=access_denied&state=xyz
- REQUIRED. A single ASCII [USASCII] error code from the following:
- invalid_request
- The request is missing a required parameter, includes an
- invalid parameter value, includes a parameter more than
- once, or is otherwise malformed.
- unauthorized_
- The client is not authorized to request an authorization
- code using this method.
- access_denied
- The resource owner or authorization server denied therequest.
- unsupported_response_type
- The authorization server does not support obtaining an
- authorization code using this method.
- invalid_scope
- The requested scope is invalid, unknown, or malformed.
- server_error
- The authorization server encountered an unexpected condition that prevented it from fulfilling the reques

- temporarily_unavailable
- The authorization server is currently unable to handle the request due to a temporary overloading or maintenanceof the server. (This error code is needed because a 503
- Service Unavailable HTTP status code cannot be returned to the client via an HTTP redirect.)

error_description

- OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in

understanding the error that occurred. Values for the "error_description" parameter MUST NOT include

characters outside the set %x20-21 / %x23-5B / %x5D-7E.

- error_uri
- OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error. Values for the "error_uri" parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.
- state
- REQUIRED if a "state" parameter was present in the clientauthorization request. The exact value received from the client.

Implicit grant flow

- GET /authorize?response_type=token&client_id=s6BhdRkqt3&state=xyz&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb&state=xyz&scope=read
- Host: server.example.com
- the authorization server issues an access token and delivers it to the client by adding
- the following parameters to the fragment component of the redirectionURI:
- HTTP/1.1 302 Found
- Location: https://example.com/cb#access_token=2YotnFZFEjr1zCsicMWpAA&state=xyz&token_type=Bearer&expires_in=3600
- access_token
- REQUIRED. The access token issued by the authorization server.
- token_type
- REQUIRED. The value should be set to bearer
- expires_in
- RECOMMENDED. The lifetime in seconds of the access token
- scope
- REQUIRED, if identical to the scope requested by the client;
- otherwise, REQUIRED. The scope of the access token as described by Section 3.3.
- state
- REQUIRED if the "state" parameter was present in the client

authorization request. The exact value received from the client.

HTTP/1.1 302 Found Location: https://client.example.com/cb#error=access_denied&state=xy

- error
- REQUIRED. A single ASCII [USASCII] error code from the following:
- invalid_request
- The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than

once, or is otherwise malformed.

- unauthorized_client
- The client is not authorized to request an access token using this method.
- access_denied
- The resource owner or authorization server denied the request.
- unsupported_response_type
- The authorization server does not support obtaining an access token using this method.
- invalid_scope
- The requested scope is invalid, unknown, or malformed.
- server_error
- The authorization server encountered an unexpected condition that prevented it from fulfilling the request.
- (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client
- via an HTTP redirect.)
- temporarily_unavailable
- The authorization server is currently unable to handle the request due to a temporary overloading or maintenance
- of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned
- to the client via an HTTP redirect.)
- Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.
- error_description
- OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in
- understanding the error that occurred. Values for the "error_description" parameter MUST NOT include
- characters outside the set %x20-21 / %x23-5B / %x5D-7E.
- error_uri
- OPTIONAL. A URI identifying a human-readable web page with
- information about the error, used to provide the client developer with additional information about the error.
- Values for the "error_uri" parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters
- outside the set %x21 / %x23-5B / %x5D-7E.
- state
- REQUIRED if a "state" parameter was present in the client authorization request. The exact value received from the
- client.

/token

- The token request endpoint must support post with Content-Type: application/x-www-form-urlencoded
- the token request endpoint must authenticate the client making the request
- the token request end point must support basic authentication
- the token endpoint must ensure that the authorization code was issued to this client_ID
- the token endpoint must ensure that the authorization code is valid.
- the token endpoint must ensure that the authorization code is used ONLY once.
- authorization code must expire in 10s of seconds
- The token endpoint must set Cache-Control: no-store, Pragma: no-cache headers
- The token request endpoint supports the following parameters:
  - grant_type with value set to authorization_code, client_credentials, password, or refresh_token
  - code with its value set to the authorization code
  - redirect_URI with its value set to the redirect URI that was provided in the request for the authorization code
  - client_id; this value is required if the client is not authenticating with the authorization server
- The return payload must include the following:
  access_token, token_type, expires_in refresh_token, and any other key value pairs.

Exchange an authorization code for a token

- POST /token HTTP/1.1
  Host: server.example.com
  Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  Content-Type: application/x-www-form-urlencoded

- grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb

- HTTP/1.1 200 OK
  Content-Type: application/json;charset=UTF-8
  Cache-Control: no-store
  Pragma: no-cache

{
"access_token":"2YotnFZFEjr1zCsicMWpAA", "token_type":“Bearer", "expires_in":3600,
"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
"example_parameter":"example_value"
}

Refreshing the access token

- POST /token HTTP/1.1
  Host: server.example.com
  Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  Content-Type: application/x-www-form-urlencoded

- grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2TlKWIA

Methodology for securing rest API

- Client app registers with Oauth/Authorization Server
- Client app request a token
- oAuth provider generates an access token to client APP
- Client app includes access token in every HTTP request using Authorization header
- Client app sets the Authorization header to Bearer {access token}
- The rest API validates the access token
  - What does it need to validate the token?

Token Validation by Resource Server

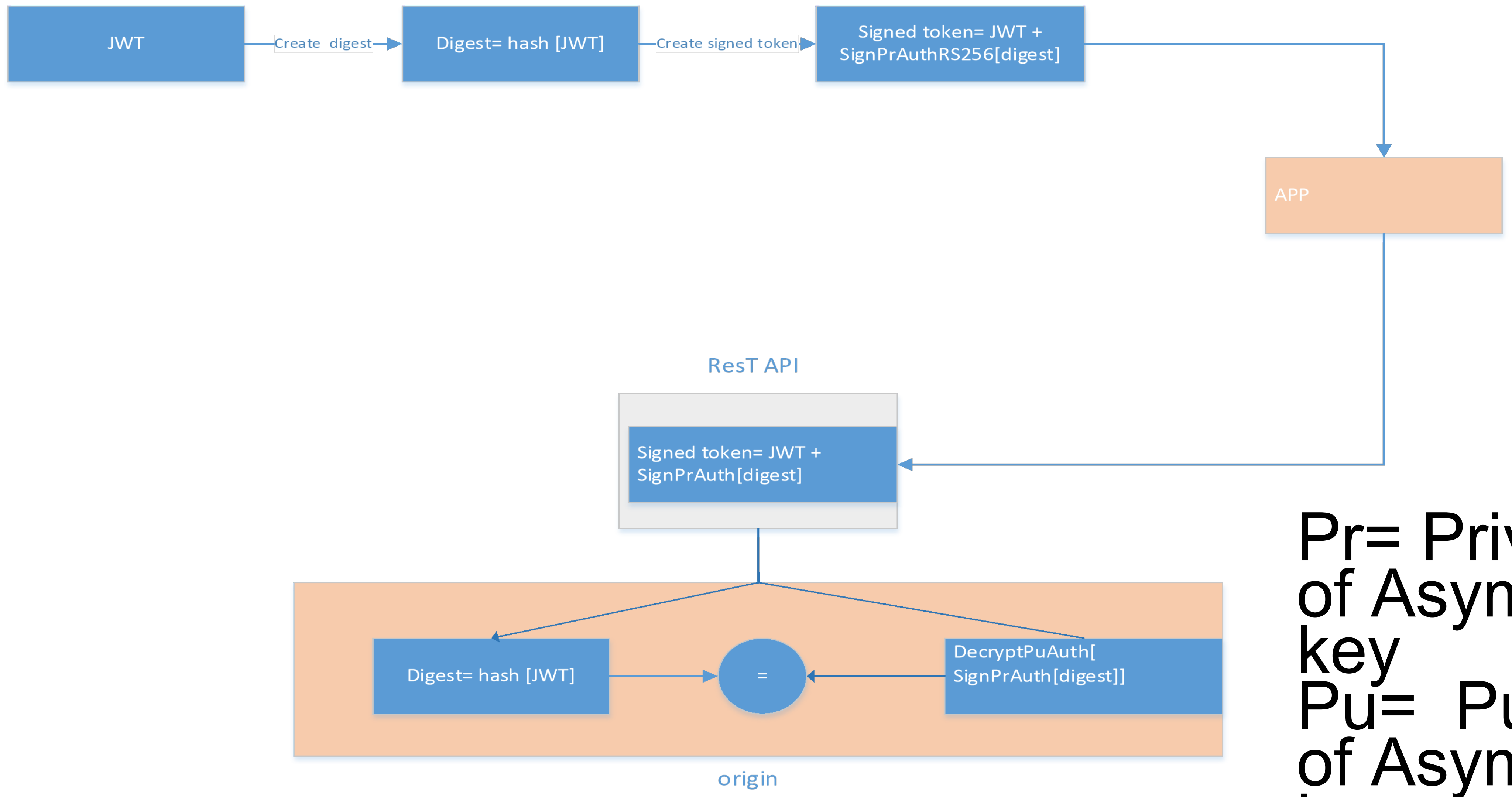
- 1. Validate the structure of a JWT
- 2. Create an “allow list” that contains valid values for iss claims
- 3. Base64decode JWT header, payload
- 4. Retrieve alg and kid from Header
- 5. Retrieve iss from payload
- 6. Compare the value of iss to that stored in the “allowed list”
  5. If iss value in allow list, use JWKS to retrieve public key. Otherwise, signature invalid
  6. Verify signature
  7. Validate any other claims such as scope, aud, exp, etc.

JWT example

```
{
  "typ": "JWT"
}


{
  "app": "TEST",
  "acc": "7888-a9a0-4de2-be72-57775575",
  "iss": "yyy",
“scope”:[”read,write”]
  "exp": 1561939073,
  "jti": "jhhhjhg-6cab-lkjjll-8512-kjkkjk“,
“aud”:”/plan/{id}”
}
RSASHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload)
```

Signature Verification using RS 256


JWT
Create digest
Digest= hash [JWT]
Create signed token
Signed token= JWT + SignPrAuthRS256[digest]
APP
ResT API
Signed token= JWT + SignPrAuth[digest]
Digest= hash [JWT]
=
DecryptPuAuth[SignPrAuth[digest]]
origin

Pr= Private key of Asymmetric key
Pu= Public key of Asymmetric key

Key Distribution

- When using RS 256:
  - Generate a public/private key pair
  - Signer uses the private key to sign the token
  - Rest API uses the public key to verify the signature
  - Rest API must have access to the public key
    - JWK : https://tools.ietf.org/html/rfc7517

Key Rotation (Private) and Distribution (Public)

- Using Kid
- {
- "alg": “RS256",
- "typ": "JWT",
- "kid":"2",
- ~~"jku":https://myjwks ;;; Not recommended to include this~~
- }

References for token signing

- https://connect2id.com/products/nimbus-jose-jwt/examples/jwt-with-rsa-signature
- https://en.wikipedia.org/wiki/JSON_Web_Token
- https://tools.ietf.org/html/rfc7519
- https://developers.google.com/oauthplayground/
- https://developers.google.com/identity/protocols/oauth2/openid-connect
- https://console.developers.google.com/apis/credentials?project=vital-invention-306022
- https://accounts.google.com/.well-known/openid-configuration
- JWT.io
- https://developers.google.com/identity/protocols/oauth2/openid-connect

Oauth

INFO 7255

Use cases for security

- Washington Post/Boston Globe: paywall, tiered-based subscription
- Flash sales
  - Can I prevent bots from sweeping up all inventory?
  - Can my application hold up against excessive demand?
    - Digital Waiting Room
- Authenticated access
  - Quota and throttling
- Anonymous access
  - throttling
- Bots Access
  - Good bots versus bad bots

Security requirements

- Authorized access against API
  - Only users authorized to access resources are allowed
  - Users able to see/edit their own plans
  - Users may read other plans, but no change them
    - Users may have certain access to this endpoint but not to the other one
- Anonymous browsing may be allowed
  - This is prior to user authentication
- App may not exceed certain requests per day/month: quota
- Apps that are making excessive number of requests need to be throttled
  - Digital Waiting Room

High-level Approach

- Client includes an authorization header
  - The value of the header is a token
- API uses the authorization header value (token) for authorization and authentication
  - Client signs token
  - API verifies token

Key design questions

- What is the overall approach for securing APIs?
  - Bearer Tokens
- What is the token structure?
  - JWT
- How are token generated?
  - How are they signed?
    - By an Idp
- How are tokens verified?
  - Authenticate the signer of the token
- Security crypto: Asymmetric? RS256
- Security guarantees: Authentication, non-tampering

First approach: API keys for securing access by apps

- High level flow:
  - Each app is granted a key at build time by the server
  - app includes key in every request that goes to server
  - Implications on quota and throttling?

OAUTH 1.0

- Username & Password

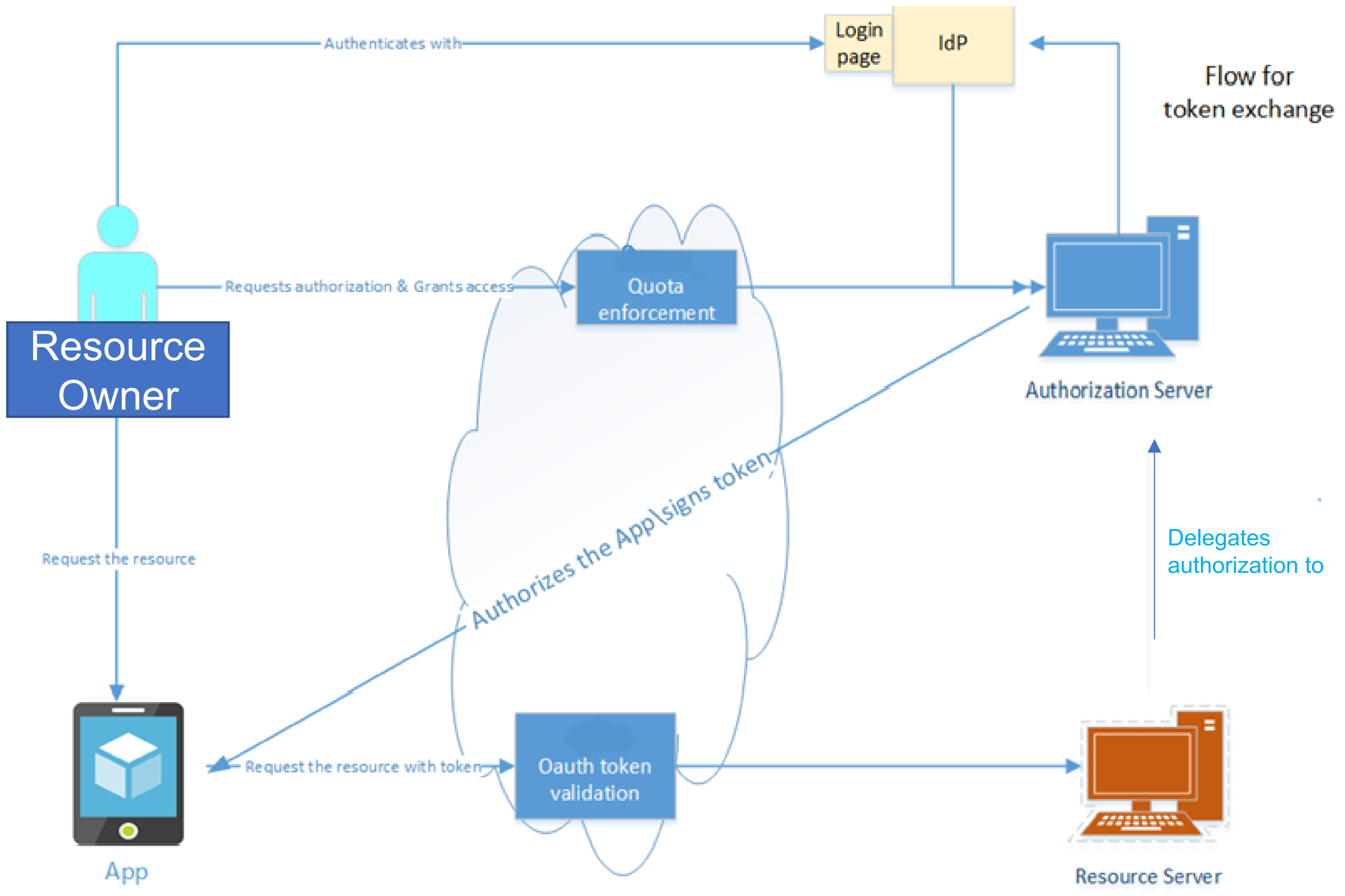
Industry accepted approach: OAUTH 2.0

- User downloads an app
- User authenticates with an IDP/Auth server
- User consents to give app access to user’s data
- IDP generates token
- App includes the token in the API calls

Public versus private app

- Public apps are those that cannot secure their credentials: single page application, mobile apps
- Private apps are those that can secure their credentials: any app running behind a firewall

oAUTH 2.0 Overview and Actors



Token Validation by Resource Server

- 1. Validate the structure of a JWT
- 2. Create an “allow list” that contains valid values for iss claim
- 3. Base64decode JWT header, payload
- 4. Retrieve alg and kid from Header
- 5. Retrieve iss from payload
- 6. Compare the value of iss to that stored in the “allowed list”
  5. If iss value in allow list, use JWKS_URI to retrieve public key. Otherwise, signature invalid
  6. Verify signature
  7. Validate any other claims such as scope, aud, exp, etc.

Overview

- RFC OAUTH 2.0:https://tools.ietf.org/html/rfc6749
- JWT https://tools.ietf.org/html/rfc7519
- Example: https://dev.fitbit.com/docs/oauth2/

Oauth provider (Authorization Server)

- /register
- /Authorize
  - unsecure
  - Authorization code grant flow
    - Returns both access token and refresh token
    - Use for secure clients
  - Authorization code grant flow with PKCE
    - Use for unsecured client
  - implicit grant flow
    - Returns only access token
    - Use for unsecured client
- /Token
  - Secure
  - Exchange authorization code for a token
  - generate a new token from a refresh token

/register

- Input:
  - Client_type = confidential (private) or public
  - redirect_URI: https://
- Output:
  - client_ID, client_secret if client is confidential
  - Client_id for public

/Authorize

- The authorization endpoint must support "get"

- The supported query parameters are:
  - response_type
    REQUIRED. Value MUST be either "code“ or "token“
  - client_id
    REQUIRED. The client identifier obtained from the registration
  - redirect_uri
    Required. As described in Section 3.1.2.
    https
  - scope
    Required.
  - state
    Required

Authorization grant code flow example:

GET /authorize?response_type=code&client_id=s6BhdRkqt3&state=xyz
&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb &scope=read
HTTP/1.1

Host: server.example.com
•The response should have:
•HTTP status code should be set to 302:
•the redirect URI as the value of the location header.
•the query parameter code and its value, state and its value appended to the redirect URI.
•The code is required at all times. The state is required only if it has been present in the request.
•example:
•HTTP/1.1 302 Found
•Location: and that we should specify the location and the location it should contain it&state=xyz

error

- HTTP/1.1 302 Found
- Location: https://client.example.com/cb?error=access_denied&state=xyz
- REQUIRED. A single ASCII [USASCII] error code from the following:
- invalid_request
- The request is missing a required parameter, includes an
- invalid parameter value, includes a parameter more than
- once, or is otherwise malformed.
- unauthorized_
- The client is not authorized to request an authorization
- code using this method.
- access_denied
- The resource owner or authorization server denied therequest.
- unsupported_response_type
- The authorization server does not support obtaining an
- authorization code using this method.
- invalid_scope
- The requested scope is invalid, unknown, or malformed.
- server_error
- The authorization server encountered an unexpected condition that prevented it from fulfilling the reques

- temporarily_unavailable
- The authorization server is currently unable to handle the request due to a temporary overloading or maintenanceof the server. (This error code is needed because a 503
- Service Unavailable HTTP status code cannot be returned to the client via an HTTP redirect.)

error_description

- OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in

understanding the error that occurred. Values for the "error_description" parameter MUST NOT include

characters outside the set %x20-21 / %x23-5B / %x5D-7E.

- error_uri
- OPTIONAL. A URI identifying a human-readable web page with information about the error, used to provide the client developer with additional information about the error. Values for the "error_uri" parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters outside the set %x21 / %x23-5B / %x5D-7E.
- state
- REQUIRED if a "state" parameter was present in the clientauthorization request. The exact value received from the client.

Implicit grant flow

- GET /authorize?response_type=token&client_id=s6BhdRkqt3&state=xyz&redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb&state=xyz&scope=read
- Host: server.example.com
- the authorization server issues an access token and delivers it to the client by adding
- the following parameters to the fragment component of the redirectionURI:
- HTTP/1.1 302 Found
- Location: https://example.com/cb#access_token=2YotnFZFEjr1zCsicMWpAA&state=xyz&token_type=Bearer&expires_in=3600
- access_token
- REQUIRED. The access token issued by the authorization server.
- token_type
- REQUIRED. The value should be set to bearer
- expires_in
- RECOMMENDED. The lifetime in seconds of the access token
- scope
- REQUIRED, if identical to the scope requested by the client;
- otherwise, REQUIRED. The scope of the access token as described by Section 3.3.
- state
- REQUIRED if the "state" parameter was present in the client

authorization request. The exact value received from the client.

HTTP/1.1 302 Found Location: https://client.example.com/cb#error=access_denied&state=xy

- error
- REQUIRED. A single ASCII [USASCII] error code from the following:
- invalid_request
- The request is missing a required parameter, includes an invalid parameter value, includes a parameter more than

once, or is otherwise malformed.

- unauthorized_client
- The client is not authorized to request an access token using this method.
- access_denied
- The resource owner or authorization server denied the request.
- unsupported_response_type
- The authorization server does not support obtaining an access token using this method.
- invalid_scope
- The requested scope is invalid, unknown, or malformed.
- server_error
- The authorization server encountered an unexpected condition that prevented it from fulfilling the request.
- (This error code is needed because a 500 Internal Server Error HTTP status code cannot be returned to the client
- via an HTTP redirect.)
- temporarily_unavailable
- The authorization server is currently unable to handle the request due to a temporary overloading or maintenance
- of the server. (This error code is needed because a 503 Service Unavailable HTTP status code cannot be returned
- to the client via an HTTP redirect.)
- Values for the "error" parameter MUST NOT include characters outside the set %x20-21 / %x23-5B / %x5D-7E.
- error_description
- OPTIONAL. Human-readable ASCII [USASCII] text providing additional information, used to assist the client developer in
- understanding the error that occurred. Values for the "error_description" parameter MUST NOT include
- characters outside the set %x20-21 / %x23-5B / %x5D-7E.
- error_uri
- OPTIONAL. A URI identifying a human-readable web page with
- information about the error, used to provide the client developer with additional information about the error.
- Values for the "error_uri" parameter MUST conform to the URI-reference syntax and thus MUST NOT include characters
- outside the set %x21 / %x23-5B / %x5D-7E.
- state
- REQUIRED if a "state" parameter was present in the client authorization request. The exact value received from the
- client.

/token

- The token request endpoint must support post with Content-Type: application/x-www-form-urlencoded
- the token request endpoint must authenticate the client making the request
- the token request end point must support basic authentication
- the token endpoint must ensure that the authorization code was issued to this client_ID
- the token endpoint must ensure that the authorization code is valid.
- the token endpoint must ensure that the authorization code is used ONLY once.
- authorization code must expire in a few seconds
- The token endpoint must set Cache-Control: no-store, Pragma: no-cache headers
- The token request endpoint supports the following parameters:
    - grant_type with value set to authorization_code, client_credentials, password, or refresh_token
    - code with its value set to the authorization code
    - redirect_URI with its value set to the redirect URI that was provided in the request for the authorization code
    - client_id; this value is required if the client is not authenticating with the authorization server
- The return payload must include the following:
  access_token, token_type, expires_in refresh_token, and any other key value pairs.

Exchange an authorization code for a token

- POST /token HTTP/1.1
  Host: server.example.com
  Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  Content-Type: application/x-www-form-urlencoded

- grant_type=authorization_code&code=SplxlOBeZQQYbYS6WxSbIA
  &redirect_uri=https%3A%2F%2Fclient%2Eexample%2Ecom%2Fcb

- HTTP/1.1 200 OK
  Content-Type: application/json;charset=UTF-8
  Cache-Control: no-store
  Pragma: no-cache

{
"access_token":"2YotnFZFEjr1zCsicMWpAA", "token_type":“Bearer", "expires_in":3600,
"refresh_token":"tGzv3JOkF0XG5Qx2TlKWIA",
"example_parameter":"example_value"
}

Refreshing the access token

- POST /token HTTP/1.1
  Host: server.example.com
  Authorization: Basic czZCaGRSa3F0MzpnWDFmQmF0M2JW
  Content-Type: application/x-www-form-urlencoded

- grant_type=refresh_token&refresh_token=tGzv3JOkF0XG5Qx2
  TlKWIA

Methodology for securing rest API

- Client app registers with Oauth/Authorization Server
- Client app request a token
- oAuth provider generates an access token to client APP
- Client app includes access token in every HTTP request using Authorization header
- Client app sets the Authorization header to Bearer {access token}
- The rest API validates the access token
  - What does it need to validate the token?

Token Validation by Resource Server

- 1. Validate the structure of a JWT
- 2. Create an “allow list” that contains valid values for iss claims
- 3. Base64decode JWT header, payload
- 4. Retrieve alg and kid from Header
- 5. Retrieve iss from payload
- 6. Compare the value of iss to that stored in the “allowed list”
    5. If iss value in allow list, use JWKS to retrieve public key. Otherwise, signature invalid

    6. Verify signature

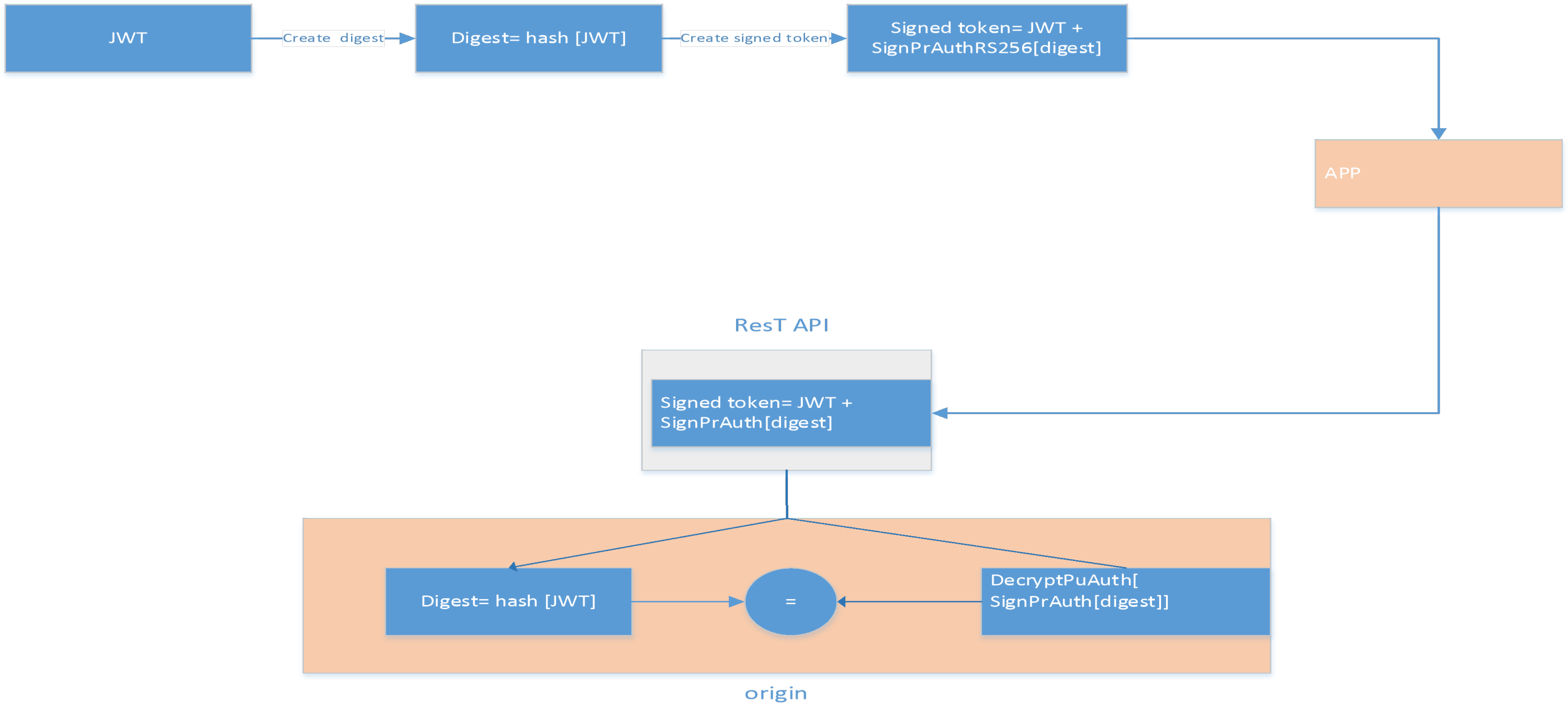
    7. Validate any other claims such as scope, aud, exp, etc.

JWT example

```
{
  "typ": "JWT"
}



{
  "app": "TEST",
  "acc": "7888-a9a0-4de2-be72-57775575",
  "iss": "yyy",
“scope”:[”read,write”]
  "exp": 1561939073,
  "jti": "jhhhjhg-6cab-lkjjll-8512-kjkkjk“,
“aud”:”/plan/{id}”
}
RSASHA256( base64UrlEncode(header) + "." + base64UrlEncode(payload)
```


JWT
Create digest
Digest= hash [JWT]
Create signed token
Signed token= JWT + SignPrAuthRS256[digest]
APP
ResT API
Signed token= JWT + SignPrAuth[digest]
Digest= hash [JWT]
=
DecryptPuAuth[SignPrAuth[digest]]
origin

Key Distribution

- When using RS 256:
  - Generate a public/private key pair
  - Signer uses the private key to sign the token
  - Rest API uses the public key to verify the signature
  - Rest API must have access to the public key
    - JWK : https://tools.ietf.org/html/rfc7517

Key Rotation (Private) and Distribution (Public)

- Using Kid
- {
- "alg": “RS256",
- "typ": "JWT",
- "kid":"2",
- ~~"jku":https://myjwks ;;; Not recommended to include this~~
- }

References for token signing

- https://connect2id.com/products/nimbus-jose-jwt/examples/jwt-with-rsa-signature
- https://en.wikipedia.org/wiki/JSON_Web_Token
- https://tools.ietf.org/html/rfc7519
- https://developers.google.com/oauthplayground/
- https://developers.google.com/identity/protocols/oauth2/openid-connect
- https://console.developers.google.com/apis/credentials?project=vital-invention-306022
- https://accounts.google.com/.well-known/openid-configuration
- JWT.io
- https://developers.google.com/identity/protocols/oauth2/openid-connect

Query Service for REST APIs

Marwan Sabbouh
Independent Researcher
24 PUBLICATIONS 128 CITATIONS
SEE PROFILE

Query Service for REST APIs

Marwan Sabbouh

1 Introduction

Big Data and Micro Services architectures make heavy use of ReST APIs that are backed by NoSQL databases and full text search engines. Notably missing from these architectures are relational databases due to the requirements of high throughput, high availability, and low latency on CRUD operations. To provide querying capabilities, full text search engines (e.g. Elastic Search) are starting to provide limited join queries capabilities. Additionally, distributed graph query engines (Titan) and graph query languages (e.g. GraphQL), are gaining in popularity. While these technologies are filling the functional gaps, they are also increasing the complexities of big data architectures. In this paper, I describe an approach for querying data that only leverages full text search and NoSQL databases. The approach builds on the conventions specified by data interchange protocols (e.g. DIP, gData, OData) to define a fast graph based indexing technique and an addressing scheme for all entities consumed by the ReST APIs. The graph based indexing algorithm runs in real-time as the data is ingested through the ReST API. The indexing algorithms store the generated indices in the NoSQL database, which are used by the query service for query resolution. In this short paper, I will describe the indexing algorithm and query resolution. For addressing, we will assume an addressing scheme like JSON Path.

2 Background

In this section, I define terminology that helps in explaining the indexing algorithm.

1. A graph is comprised of nodes and relationships. Each graph node is uniquely identified through its type and its unique identifier. A source node is connected to a target node through a relation. Consequently, we define a relation as having a source node and a target node.
2. The domain of the relation is the type of the source node.
3. The range of the relation is the type of the target node.
4. A JSON Object is a collection of property/value pairs. There are two types of properties in a JSON Object: simple properties, and object properties. A simple property is a property whose value is not a JSON Object. An object property is a property whose value is a JSON Object.
5. A JSON Object consisting only of simple properties is a node in the graph.
6. An object property is a relation in the graph.
7. The value of an object property consisting only of simple properties is a target node in the graph.

Figure 1 illustrates a JSON document describing comments on a blog entry. The JSON document consists of three JSON Objects. The root object with simple properties _type, _id, title and with the object property comments. The value of the comments property is another JSON Object with simple properties _type, _id, created and with the object property author. The value of the author property is another JSON Object with simple properties _type, _id, and name.

```
{“_type”: “blog”,
“_id”: “123456”,
“title”: “novel indexing techniques”,
“comments”:[ {
        “_type”: “comment”,
        “_id”: “78910”,
        “created”: “05-12-2017”,
        “author”: {
            “name”: “Michael Smith”, // NOTE: Duplicated
            field
            “_id”: “121314”,
            “_type”: “person”
            “_inferred”:[” blog-person”]}
    }]
}
```

Figure 1: Sample JSON Document

Figure 2 shows the graph model of the JSON document.

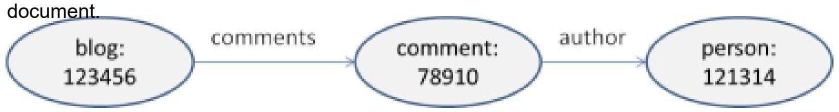


Figure 2: Graph Representation of Figure 1

3 Indexing Algorithm

1. First, the indexing algorithm interprets the nested document shown in figure 1 as a directed graph. This is shown in figure 2.
2. Second, the indexing algorithm makes use of inverseOf inference and transitive inference. The resulting graph is shown in figure 3.

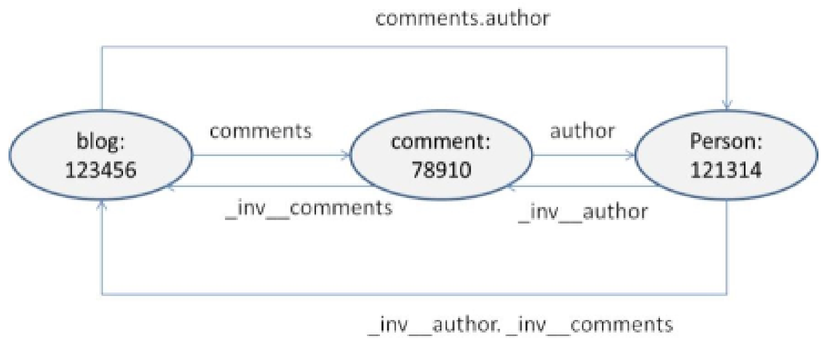


Figure 3: Inferred graph of figure 2

Figure 3 shows *_inv__comments* as the inverse relationship of *comment*, *_inv__author* as the inverse relationship of author, and *_inv__author. _inv__comments* as the inverse relationship of *comments.author*. These inverse relationships are generated automatically by the indexing algorithm. Figure 3 also shows the transitive relationship that was inferred by the algorithm. In this example, the transitive relationship is shown to be *comments.author*. Note that, we left the algorithm generate the transitive relationship name and the best it could do is something like "*comments.author*". The algorithm also infers the relationship between nodes blog:123456 and person:121314 . This eliminates the need of the data modeler defining inverseOf and transitive relationships.

3. Third, the indexing algorithm infers the range of all transitive relationships. In this example, this transitive relationship "comments.author" has "blog-person" as *inferred* object type for its target object.

4. Fourth, the indexing algorithm annotates the target object of all transitive relationships with their inferred type. For the example shown in figure 1, the target object of the transitive relationship "comments.author" is person_121314. It contains the property _inferred with value "blog-person".

Internally to our system, the indexing algorithm represents the inferred graph as shown in figure 4.

1. comment_78910_ _inv__comments: [blog_123456]
2. comment_78910_author: [person_121314]
3. person_121314_ _inv__author: [comment_78910] *
4. person_121314_ _inv__author._inv__comments: [blog_123456]*
5. blog_123456_comments: [comment_78910]
6. blog_123456_comments.author: [person_121314, blog-person_121314]*
7. relation_ _inv__author._inv__comments_inverseOf: [comments.author]*

8. relation_ _inv__author_inverseOf: [author]*

9. relation_comments.author_inverseOf: [_inv__author._inv__comments]*

10. relation_comments_inverseOf: [_inv__author]*

11. author_inverseOf_inferred: [blog-person_121314]

Figure 4: Semantic index

Note that in figure 4 the lines ending with an asterisk indicates inferred knowledge.

4 Querying Algorithm

To illustrate how the indexing algorithm can be used in search, I present the following example. Suppose a user would like to find all persons whose first name Michael and who have commented on blogs. Assuming the availability of an inverted index search engine, e.g. Elastic Search, the query syntax may look something like this:

/search/indexname/blog?q = comment. author. name: "Michael*".

To understand the search query, the URI path is of the following form: /search/indexname/{object type} followed by a typical Lucene query parameters. The fact that the object type is blog, tells the search engine to return instances of object type blog that matches property name with the value starting with "Michael". The JSON path of the property name in the nested document is "comment.author.name". In this case, the JSON document shown in figure 1 will be returned. Please observe that for this query to work the property "name" of object type person must be duplicated in the nested object of object type person.

To avoid data duplication, we can use the semantic index to do the join query. In this case, the query remains the same as shown above: /search/indexname/blog?q = comment.author.name: "Michael*".

The query parser does the following:

1. Splits the JSON path "comment.author.name" into relationships comments.author, and property name
2. Find the range of the relation comments.author. This returns type person, and the inferred type blog-author.
3. Form the query to find all instances of person with the property name starts with Michael: /search/indexname/person?q = name: "Michael*" & field: "_id".
4. Append to the query above the inferred type as a query parameter. The above query becomes: /search/indexname/person?q = name: "Michael*" & field: "_id" & _inferred: blog-person.
5. Executes the query. It returns the JSON instances of type person with _inferred property set to blog-person, and the name property starting with Michael. However, due to the query parameter field: "_id" the search engine only returns a list of all the JSON instance identifiers that matches the query. For this simple example, the returned list will contain 121314 which is the value of the property "_id" of the JSON instance of type person.
6. Find the inverse of relationship comments.author (found in the first step) by looking up in the semantic index the value of relation_comments.author_inverseOf . This returns relation _inv__author._inv__comments.
7. Find in semantic index the value of the key person_121314_ _inv__author._inv__comments. This returns blog_123456

The above algorithm demonstrates how to accomplish join query without data duplication. That is, we could remove the name property from the JSON document of figure 1, and the search query will still return the correct result. This is because the query in step 3 is to search for instances of object type person.

5 Experimental Results

I implemented the indexing algorithm as part of the ReST API and the query service rest API. I used Redis in conjunction with Elastic Search to implement the system. The ReST API was implemented in Java using Spring Boot. The early results are quite encouraging. We can index a JSON document consisting of 5000 objects in less than 30 ms. A sophisticated join query consisting of three conditions on different nodes in the graph returned in less than 30 ms. Furthermore, the indexing algorithm also proved useful in implementing merge/patch functionality for ReST API, in addition to playing a key role in business rule specification and enforcement. I will describe the latter technique in another document.

6 Conclusion

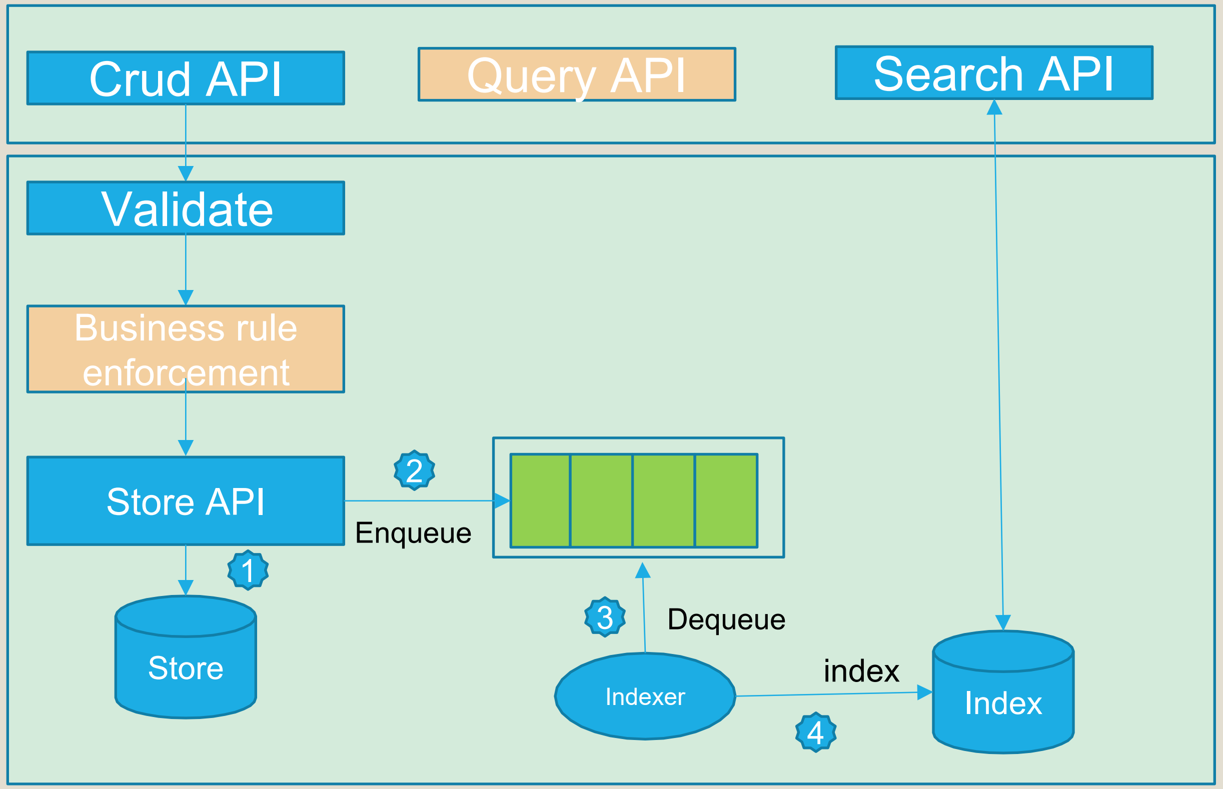
As we require more functionality from big data technologies, we are faced with increased technological complexities that are preventing many small to medium-size companies from adopting these technologies. For these architectures to become pervasive, techniques that simplify the big data technological stack are critically needed. I believe the approach described here is a step in that direction.

ADVANCED TOPIC IN BIG DATA

Quick Review

- By now, you should be familiar with strongly typed data protocols
- You should have reviewed gData, oData, Protocol Buffers
- You should have fair understanding of the overall architecture
- You should have some code working on your laptop

Architecture



Prototype Requirements:

Rest API that can handle <u>any structured data in Json</u>

- URIs, status codes, headers, data model, version

- Rest API with support for crd operations
    - Post, Get, Delete
- Rest API with support for validation
    - Json Schema describing the data model for the use case
    - Controller validates incoming payloads against json schema
- The semantics with ReST API operations such as update if not changed/read if changed
    - Update not required
    - Conditional read is required
- Storage of data in key/value store
- Must implement use case provided

Rest API Specifications

- Data Models
  - Payload structure and serialization
- URI conventions
  - /{type}/{id}
  - /plan/12xvxc345ssdsds
- Status Code
  - 200,201
  - 302,304
  - 401, 404, 403, 412, 429
  - 500
- Headers
  - Students should review the HTTP standard headers
  - Various uses of Etag, If-Match, If-None-Match, Authorization in Rest APIs
- Version
  - Accept
  - URL
- Security

- Example: https://www.hl7.org/fhir/http.html

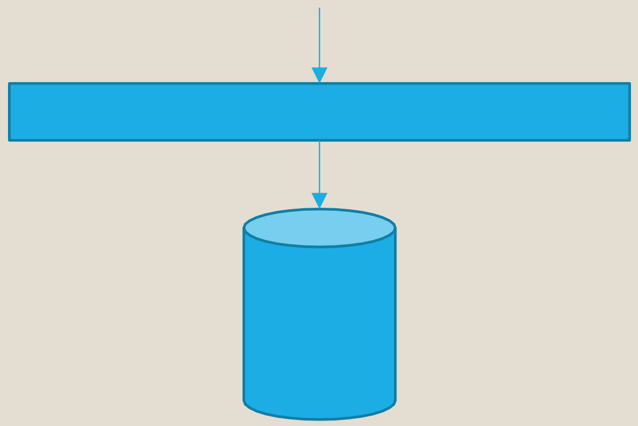
Tooling

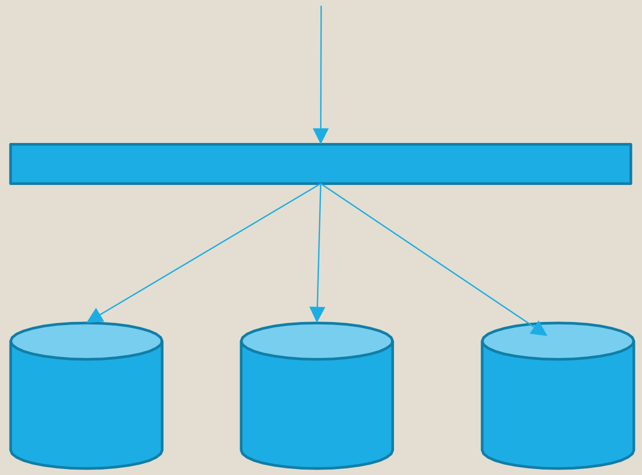
- Json simple for Json parsing
- Spring Boot for rest API development
- Elastic Search for search and retrieval capabilities
- Redis for Cache solutions
- Json Schema for schema validation
- Zuul for API Gateway pattern

But how do I distribute the data?

- single point of failure
- Limited space/storage
- Strongly consistent

- Highly available distributed system
- Seemingly unlimited storage
- What about consistency?





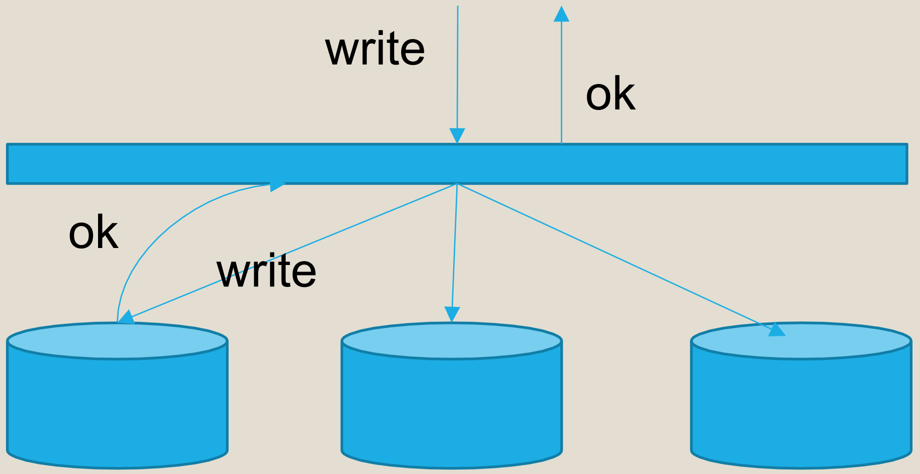
Key/value stores

- Key readings:
  - Dynamo: Amazon’s Highly Available Key-value Store :
    - http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
  - Bigtable: A Distributed Storage System for Structured Data: http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable - osdi06.pdf
  - CAP Theorem
    - *Consistency*
      - Eventual consistency, Read your own write, Strongly consistent
    - *Availability*
    - *Partition tolerance*
    - In the presence of network failure, you have to choose between consistency and high-availability

Problems

- In the presence of many servers, how do I determine the server that stores the object?
  - Consistent hashing to the rescue
- But what if one of the servers fails or the network connection to the server fails?
  - Replication techniques:
    - Primary/backup
    - Active replication
- If I have multiple servers and if an object is stored on more than one server
  - How do I keep the objects consistent?
    - Eventual consistency, strong consistency, weak consistency

Weak consistency



Quorum consistency

R=read replica count

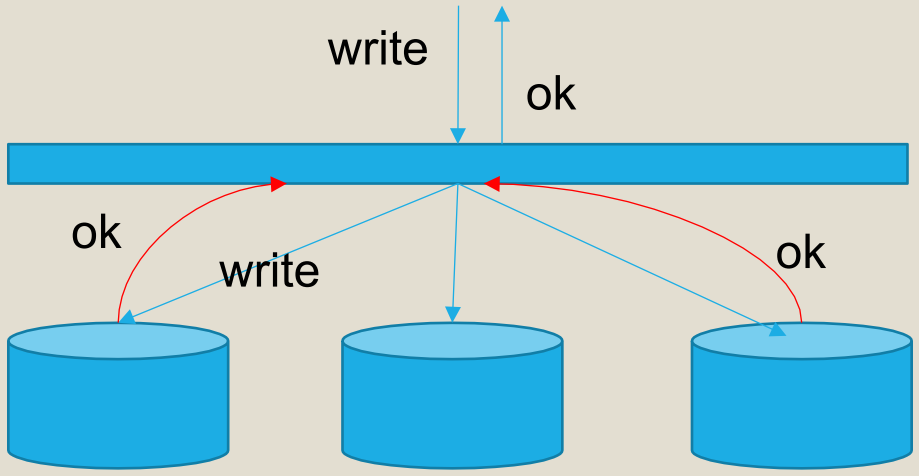
W=write replica count

N=replication factor

Q=**QUORUM** (Q = N / 2 + 1)

- If W + R > N, you will have consistency

- On read, two of the replica must respond
- On write, two of the replica must make the data durable before acknowledging the right



Data Modelling

- K1□v1
- K2□v2
- K1?, K2?

Consistent hashing

- http://theory.stanford.edu/~tim/s17/l/l1.pdf

- How do you map a large number of objects into few servers?
  - h(x) mod n
- What if the number of servers changes, what would that do to the objects that have already been assigned?
- How do ensure a universal distribution of objects across servers?
- The key idea is:
  - hashing the names of all objects
  - hash the names of all the cache servers s
  - The object and cache names need to be hashed to the same range, such as 32-bit values.

Key design issues

- Partitioning algorithm
  - Uniform load distribution
- Schema less
- Replication strategy
- Recovering from partial failure
  - Joining a group
    - Load partitioning amongst replicas
- Load rebalancing
- Range query support
- Data versioning
- Support for structured data or simply Blobs
- Marshaling/Unmarshaling
  - How do you store int and floats in redis?

Mapping of meta-model into key/value store

- JSON payloads can be modeled as a graph.
- https://www.researchgate.net/publication/315679274_Query_Service_for_REST_APIs
- https://www.researchgate.net/publication/315679444_Business_Rules_for_REST_APIs

  How do we map a JSONObject into the key value store?
- What is the key signature?
- Do we store the data as a blob?

- Do we store the data as structured?

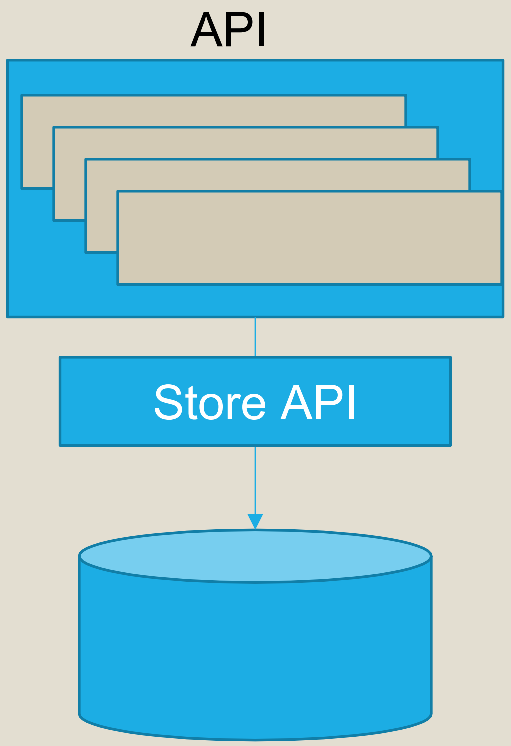
Trade-offs between storing the data as a blob versus structured storage

- Storing data as a blob is fast, atomic, reliable
  - But, how do you update the data?

- Storing the data as structured data requires more work on initial creation, but update are much quicker

A typical design pattern

A compound document
with nested objects



Should the compound document be decomposed into its constituent objects for storage, and/or indexing, etc…?

ADVANCED TOPIC IN BIG DATA

Quick Review

- Syllabus
- JSON
- JSON Schema
- Creating strongly types system with JSON
- The need for validation
- Addressing (briefly)

Creating strongly typed data with Json

- Every object is an instance of a type
- System exposes aspects, e.g. _id, _type, etc. that are used in any object.
- Define the type version in the system, and associate with it the property test.
- The property test has datatype array of integers

```
◦ {
◦         "_name": "version",
◦         "_type": "Entry",
◦         "_org": "logoilabs.com",
◦         "_id": "version",

◦                 "properties": [{
◦                 "_name": "test",
◦                 "dataType": ["int"],
◦                 "_type": "Field",
◦                 "_org": "logoilabs.com",
◦                 "_id": "version----test",
◦                 "isFieldOf": "version"}]
◦         }
```

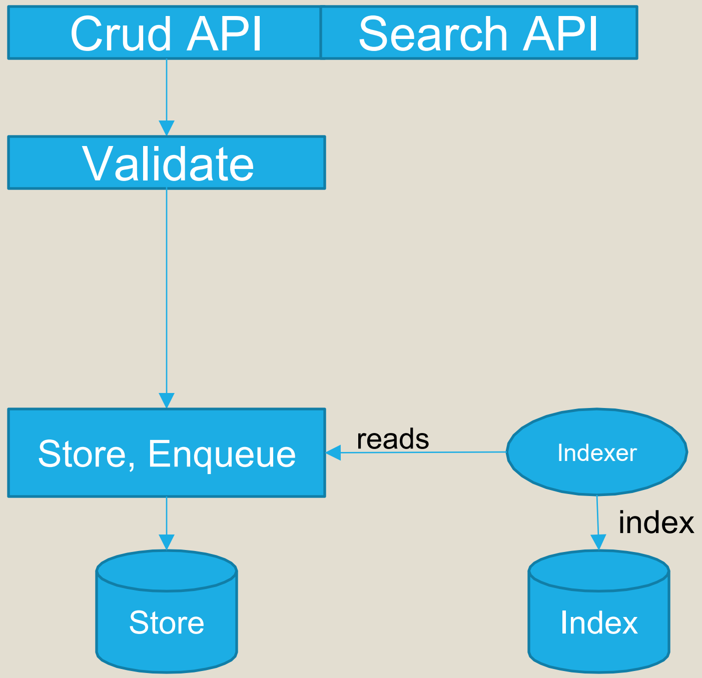
Variety of strongly typed system

- define types and properties
- define references to objects
- support for inheritance?
- extending the definition of types with additional properties
- system defined types and properties
- aspects
- support for versioning?
- advanced data modeling primitives: intersection, one of, cardinality support, union
- Examples:
  - GDATA, https://developers.google.com/gdata/
  - Protobuf, https://developers.google.com/protocol-buffers/
  - Microsoft Odata
  - Facebook:GraphQL

Class exercise

- Come up with a convention to depict a reference to an object in a json payload
- See if you can come up with two different ways of doing it, and then compare the two to choose the better one
- 15 minutes

Architecture



Rest API Specifications

- URI conventions
  - /type/id
- Headers
  - Students should review the HTTP standard headers
  - Various uses of Etag, Not Modified Since, Authorization in Rest APIs
- Payload structure and serialization
- Security
- Status Code
- Example: https://www.hl7.org/fhir/http.html

- marwansabbouh@gmail.com

Data Sharing for Cloud Computing Platf orms

Marwan Sabbouh, Kenneth McCracken, Geoff Cooney
Social Platform
Here, a Nokia Business
Cambridge, MA, USA
Marwan.Sabbouh@here.com, Ken.McCracken@here.com, Geoff.Cooney@here.com

***Abstract*— Cloud computing platforms consist of a set of reliable services that are run in the cloud. Typically, consumer applications use software development kits (SDKs) provided by the computing platform services to store, update, and retrieve instances of data in the cloud. Services provided by the cloud computing platform, expose different data modeling paradigms that consumer applications use to interact with the cloud. The service-specific data modeling paradigms and SDKs increase the complexity of data sharing between consumer applications that interact with the different services of the cloud computing platform. To make matters more complicated, it's not uncommon in an enterprise to find different groups using different cloud computing platforms. In this paper, we will describe a set of abstractions that can be used to abstract different computing platforms. The abstractions not only abstract the computing platform, but also enable the data discovery and sharing between applications. We will further show that these abstractions do not add substantial latency on the performance of the computing platform.**

Keywords- web protocols; data modeling language; NoSQL; Rest; API; JSON

I. Introduction

Most cloud computing providers offer a distributed datastore/database [1] [2] [3], in the form of a NoSQL engine or a NewSQL [4] [5] engine. These distributed databases expose a data modeling paradigm that their consumers can use to interact with the cloud system. For example, Amazon Web Services offers DynamoDB [6]; a fully managed NoSQL database service that makes it easy to store and retrieve data, for applications requiring varying levels of throughput. DynamoDB data modeling concepts include Tables, Items, and Attributes. Applications wishing to store their data in the cloud, can then define their tables, items and attributes as required by the distributed databases. However, what if multiple applications need to store and share their data. Certainly, the sharing of the data is enabled through the

The need for cloud data models requires the availability of the data modeling language as part of the protocol. A cloud system data model not only contains the type definition, the listing of properties and their constraints, but also contains object-metadata specifying processing directives to the cloud system, property annotations, index definitions (if search queries are desirable for example), and access control policies.

In addition to the above requirements, there is also the requirement that the data model must be processed by the same components of the cloud system that process instance data. Hence, there is a need for the data modeling language to share a common object model with the rest of the instance data.

We believe that the above requirements are enabled through the use of a data interchange protocol providing:

1) A common object model

The object model stipulates that all data are instances of some type. The object model specifies a unique way to identify and type an object. The object model also specifies how to reference a remote object. In the context of a cloud system, the object model may also specify primary keys

2) A common serialization format

The serialization format, which in our case is JavaScript Object Notation (JSON) [8], eliminates the potential of having syntactic mismatches in the system. JSON was chosen for many reasons, particularly its simplicity, fast serialization/deserialization [9] as compared to Extensible Markup Language (XML), and the familiarity of programmers with it.

3) A common exchange

The ... exchange ... applications to know a priori the expected ... format

4) A data modeling language

This in turn, eliminates the potential of any structural mismatches in the system.

The data modeling language provides the necessary vocabulary for writing data models. Data models can serve a few purposes in the context of sharing data. First, they validate the instances. Second, they can be used by both consumers and providers of data to build a common understanding for a domain. Third, they contain access control policies and indexing directives.

5) Universal Resource Identifier (URI) conventions

URI conventions describe the address of the data in the cloud system. URI conventions make it possible to navigate the exchange document to extract objects or instances, and a (property, value)-pair.

6) Representational state transfer (REST) APIs for storing and modifying the data

With URI conventions already in place, storing, creating, and retrieving data can easily be accomplished using REST [10] APIs. Once more, this API applies to both instance data and data models.

II. BACKGROUND

There are several solutions available, with each providing its own advantages and disadvantages. Google's Data Protocol (gData) [11] is a web protocol for reading, writing, and modifying data on the web. gData supports JSON serialization. The basic idea is that Google's internal services publish their data using gData, enabling consumers of the services to consume the data in a uniform way. The gData object model is that of an RSS/Atom feed. gData defines common definitions of certain objects, called Kinds in gData, but stops short of defining and using a data modeling language. Therefore, gData is an appropriate paradigm for publishing applications data, but may not be suited as a protocol for abstracting a NoSQL/NewSQL distributed database.

Open Data Protocol (OData) [12] is a web protocol that is backed by Microsoft. This protocol is used for creating and updating data using web technologies such as Atom feeds, JSON, and Hypertext Transfer Protocol (HTTP). OData provides an Entity Data Model (EDM) using Common Schema Definition Language (CSDL). CSDL is XML-based representation of the entity model exposed by the OData service. Additionally, OData provides a service model [14] that allows the discovery of the capabilities of services. While OData claims that it "provides a uniform way to describe both the data and the data model" [15], [illegible] However, the representation of the entity data model, which is in XML, and the representation of the instance data, which is RSS/Atom feed. That is, the entity data model is

Protocol Buffers [13] is an approach for encoding, serializing, de-serializing structured data in an efficient way. Protocol Buffers describe data structures in a textual format. Protocol Buffers tools are then able to compile an description of data structures and generate a proxy or stub that handles the encoding, serializing, and de-serializing of structured data. While being very efficient in many levels, the use of Protocol Buffers [illegible] type, this makes it very difficult to introduce new data model into production live systems in a seamless way. Furthermore, the data structure descriptions are

quickly discovered that JSON Schema was not expressive enough to describe the indexing requirements and some of the access control policies that need to be specified. That is, while we were able to validate instances of data models using JSON Schema, we were unable to validate the data models themselves using JSON Schema, as the data models contained index definitions for search, and access control policies, whose support required disjoint property/object, among other features missing from JSON Schema. Since, JSON Schema and our data representation in JSON did not share a common object model; we were unable to process the data models using the same software components we had in place for the instance data. This meant that in an earlier implementation, the processing of data models was done manually.

Yet, another option that was considered but ultimately rejected was the use of a different modeling language that is expressive enough, e.g. Resource Description Framework (RDF) [16] and Web Ontology Language [17] (OWL). RDF/OWL is a powerful data modeling language designed for use on the web. It offers a powerful object model, as well as vocabularies for defining types, properties, annotations, restrictions, and many other useful data modeling primitives. However, the issues with using RDF/OWL are the lack of JSON serialization for RDF/OWL, and the open world assumption of RDF/OWL. The open world assumption states that any statement that is not known to be true is not necessarily false. These issues present significant hurdles in the adoption of RDF/OWL in cloud-based systems that require validation of data. Further, that choice would have presented latency challenges for processing the instance data fast enough.

Hence, we have arrived at the conclusion that the Data Interchange Protocol is needed.

III. PROTOCOL SPECIFICATION

Data Interchange Protocol (DIP) defines an object type system using JSON. DIP uses special attributes, e.g. _type and _name, for ALL data, making all DIP Objects "typed" and "named". DIP uses a special attribute _id to specify the key of the object in the distributed database. DIP provides standard DIP object types, e.g. Entry and Field for the creation of types, and for associating properties with a type. DIP defines URI conventions for addressing DIP objects and properties. DIP data modeling language defines an inheritance model allowing Is-A relationships between types. Additionally, DIP supports data modeling features such as disjoint properties, property restrictions, property intersections (implications), property cardinality constraints, disjoint objects, and intersection objects, making the DIP modeling language expressive enough to validate both DIP access controls settings are applied on the type being defined.

A. DIP Object Model

DIP object model treats every object as instance of some type. DIP object model defines a uniform way for 1) creating types that are unique in the cloud system; 2) instantiating instances of those types; 3) identifying those instances in the cloud; 4) referencing remote instances; 4) addressing the instances and their properties. DIP objects are exchanged using DIP documents. A DIP document is comprised of a JSON list with each member of the list is a JSON object. Fig. 1, shows an example.

```
[{
"_type": "Entry",
"_id": "Field",
"_name": "Field",
"_comment": "base class",
“extends”:null
…},{…}]
```

Fig. 1. DIP document

DIP defines its object model using system properties. DIP object model reserves the use of all property names beginning with _. We refer to property names beginning with _ as System Properties. System Properties have special meanings, and they can occur in any object. They are integral to the system and they are not defined in any data model. DIP has the following System Properties that relate to the object model: _type, _name, _id, _comment, _ref, _uri.

_type specifies the type of a DIP object, meaning that all property (name, value)-pairs in the object that are non-System Properties are associated with the type specified as the value of _type. The value of _type is a single value string. Fig. 1 shows that the value of _type is "Entry". The value of _type is interpreted as a relative URI (see _uri description). In DIP, Entry is used to define new types. DIP objects are required to specify _type.

_name specifies the name of a DIP object. Since _name is Field, Fig. 1 shows that the type being defined is Field. The value of _name is a single value string, and is unique in the context of a DIP document. Hence, Fig. 2 shows a DIP document containing a single logical object comprised of two objects with the same name. Objects are required to specify _name.

```
[{
"_type": "Entry", "_id": "Field", "_name": "Field",
"_comment": "base class", …
}, {"_type": "Entry", "_name": "Field",
"extends": null}]
```

Fig. 2. Single logical object in DIP

In the DIP object model, Fig. 2 and Fig. 1 are semantically equivalent as they define a single logical object named Field with its "extends" property set to null.

_id when combined with _name is typically used to specify the key in a NoSQL database that holds the DIP document as its value. The value of _id is a single value string, which represents a unique identifier in the cloud system. _id must be present in the first object of a DIP document. When "_id" is not present in DIP objects, its value is assigned to be that of the first object.

_uri is used to combine _id, _name, _type in a single property (name, value)-pair. _uri is used as an alternative notation for _id, _name, _type. Therefore, Fig. 1 could be stated as shown in Fig. 3.

```
[{
"_uri": "/Field/Field:Entry",
"_comment": "base class",
"extends": null
…
},
{…}]
```

Fig. 3. DIP document using _uri

Fig. 3 shows the same logical object as Fig. 1 and Fig. 2. The value of "_uri" is a single value string and has the form /_id/_name:_type. Hence, Fig. 3 shows that _id is Field, _name is Field, and _type is Entry. Fig. 3 also shows the value of "_id" to be a relative URI which can be expanded to be an absolute URI by the system.

_comment provides textual description of the object being defined.

_ref provides a mechanism for referring to remote objects. The value of _ref is either a string or a list of string. This string has the same form as the value of_uri. However, the difference between _ref and _uri is that _ref points to an object that was defined elsewhere, while _uri defines the object. Fig. 4 shows a typical use of _ref.

```
[{
"_uri": "/C544C14C-51F0-0001-5FB6-2620140061F9F/Marvin:Person",
"_comment": "instance of Person named Marvin",
"employer": {"_ref": "/C544d14d-51F0-0001-5FB6-2620134344F9F/Nokia:Company"}
…
}]
```

Fig. 4. Typical use of_ref

Fig. 4 shows an instance Marwan of type Person. This instance has a property employer. Its value is a reference to an instance Nokia of type Company. It is defined in DIP document C5...F9F0-0001-5FB6-2620134344F9F.

B. Modeling of DIP data

DIP Data can be modeled as a property graph [18]. A DIP object is comprised of nodes and edges. A node is characterized by either the value of _uri, or the combined value of _name, and _id. Typically, such node has outgoing edges from it. The properties that are part of the DIP object are either attributes of a node, or outgoing edges from a node. When the property value is another DIP object containing System Property _ref, the value of _ref characterizes a node with incoming edge. That is, when the value of the property is interpreted as a URI, it is a node in the graph. Fig. 5, shows the representation of Fig. 4 as a property graph.

/E5..4C/Marwan
employer
/C5..9F/Nokia

Fig. 5. Graph representation of Fig. 4

C. DIP Type System

The DIP type system supports the basic types and structures defined by JSON, e.g. array, object, number, string, true, false, null. That is, a DIP object is a valid JSON object. Entry is the type that all other types use. Entry is used to define new types. The type Field is used to define properties and to associate them with a type. To specify that the DIP object is an instance of type X, it suffices to set the _type property to X in the DIP object.

1) Type Entry

Entry is the base type in DIP type system. Fig. 6 shows the formal description of Entry. The first object in the DIP document specifies Entry to be a type. The second object in the DIP document is an instance of type Field. This instance contains the usual System Properties, indicating that the instance name is extends. In addition to the System Properties, it contains other properties: dataType, isFieldOf, and sequence. In summary, Fig. 6 says that Entry is a type, it has a property called extends. The value of extends is a list of string. The sequence number of extends is 1. Furthermore, extends is a field of Entry. DIP adopts the convention of grouping all properties belonging to type in one DIP document.

Note that, in order for the Entry document to specify an instance of Field, Field must have been defined as a type in another document. The definition of Field is described next.

```
[{
    "_id": "Entry", "_type": "Entry",
    "_name": "Entry",
    "_comment": "base class"
 },{
    "_type": "Field",   "_name": "extends",
    "dataType": [ "string"] ,
    "isFieldOf": "Entry",  "sequence": 1
 }]
```

Fig. 6. Entry definition

2) Type Field

Fig. 7 shows excerpts from the definition of Field.

```
[{
     "_id": "Field", "_type": "Entry",
     "_name": "Field",
  },{
     "_type": "Field","_name": "dataType",
     "dataType": "any", "minCardinality": 1,
     "isFieldOf": "Field",  "sequence": 1
  },{
     "_type": "Field","_name": "isFieldOf",
     "dataType": "string",
     "isFieldOf": "Field","sequence": 2
  },… {
     "_type": "Field","_name": "sequence",
     "dataType": "int",
"isFieldOf": "Field","sequence": 13
  },…]
```

Fig. 7 Field definition

The first object in the Field document states that Field is a type. The remaining three objects in the Field document define isFieldOf, dataType, and sequence as instances of Field. That is, these instances can occur in any object of type Field. There are other properties defined for type Field that are not shown here. Notable among them are minCardinality, maxCardinality, intersectWith, and disjointWith. We offer a brief description of each of these properties.

When a user creates a type T in DIP, and defines property p for that type: if p has its dataType attribute set to "string" or "int" "Boolean", or "number", this means every instance of T must have the property's value a scalar corresponding with the data type. Otherwise, the instance is not valid. if p has its dataType attribute set to a (relative) URI, e.g. "/Address", this means the value of p is an instance of the type specified in the URI, e.g. Address. Fig. 6 shows an example of how to specify a property, i.e. extends, whose value is a List of string.

- if p has its minCardinality attribute set to a number v, this means every instance of T must contain at least v occurrences of that property. Otherwise, the instance is not valid.
- if p has its maxCardinality attribute set to a number v, this means every instance of T must contain at most v occurrences of that property. Otherwise, the instance is not valid.
- if p has its intersectWith set to another property p2, this means every instance of T with p set to a value must also have p2 set to a value. Otherwise, the instance is not valid.
- if p has its disjointWith set to another property p2, this means every instance of T with p set to a value must not have p2. Otherwise, the instance is not valid.

3) Multi-typing an Instance

DIP has the capacity to express that an object is of type A, and of type B. At first look, one might think that DIP does not allow multi-typing of instances due to the fact that the value of _type is a single value string. However, DIP accomplishes this feat by enclosing two different objects having the same value for _name, and different values for _type in the same DIP document. This is shown in Fig. 8.

```
[{
    "_id": "C5254000-….BD20F34E142C",
    "_type": "Person",
    "_name": "Marwan",
     "gender": "male"
  },{
    "_type": "Worker",
    "_name": "Marwan",
     "employer": "Nokia"
  }]
```

Fig. 8. Instance multi-typing

Fig. 8 shows an instance named Marwan having two types: Person, and Worker. This approach to multi-typing an instance keeps the properties of each type separated from each other. For example, Fig. 8 shows that employer is a property belonging to Worker, by the virtue of including this property in the object with _type set to Worker. Other approaches to achieve multi-typing are almost guaranteed to be more complicated, as they would need to introduce prefixes for each type that help distinguish the properties belonging to one type from the properties belonging to another type.

4) Type Object

In addition to specifying constraints on properties, DIP permits the specifying of intersection, and disjoint constraints on objects. Therefore, the DIP type Object defines the properties intersectWith , and disjointWith whose values must be interpreted as URIs. I will elaborate briefly on those properties without showing the definitions for the sake of brevity.

- If T sets intersectWith to type T2, then, any instance I of T, must also be an instance of T2. Otherwise, the DIP document is not valid.
- If T sets disjointWith to type T2, then, any instance I of must not be an instance of T2. Otherwise, the DIP document is not valid.

When a user defines a type T to be an instance of Entry and of Object;

Fig. 9 shows a typical use of type Object. Fig. 9 shows that the DIP document, if it contains an index, i.e. an instance of SCBEIndex, the instance of index must also be an instance of scbeAccessControl. That is, if the document defines an index, it must also define access control on the index.

```
[{
    "_type": " Object",
    "_name": " SCBEIndex"
    "intersectWith": ["scbeAccessControl"]
  },{
    "_type": " Entry",
    "_name": " SCBEIndex"
  }]
```

Fig. 9. Example use of type Object

5) Extension Mechanism

DIP supports a mechanism to express is-a relationships between objects. The processing of object A extends object B is as follows:

- Any instance of type A must also be an instance of type B. This is accomplished using the multi-typing serialization described in section C.3.
- Therefore, any document that contains an instance of type A, that same document must also contain the serialization of that instance of type B.
- All property and object constraints defined on type B are applied on the instance of type B.
- All property and object constraints defined on type A are applied to the instance of type A.

Note that the definition of extends is used in Fig. 6.

6) Serializing Properties with Cardinality Greater Than One

DIP serialization offers a straightforward approach to expressing multiple occurrence of a property. Fig. 10 shows an example.

```
[{
      "_id": "Person",
      "_type": "Entry",
      "_name": "Person"
   }, {
      "_type": "Field",
      "_name": "address",
      "dataType": "string",
      "isFieldOf": "Person",
      "sequence": 1
   },   {
      "_type": "Field",
      "_name": "address ",
      "isFieldOf": "Person",
      "dataType": "./:Address"
}]
```

Fig. 10. Definition of property address

Fig. 10 shows that the property dataType occurs twice to state that the property address can have a value either of type string or of type Address.

7) *Property Restrictions*

Earlier in the paper, we have seen examples on how to specify an object property, or property whose value is an instance of a certain type. Sometimes, it is useful to further restrict the value of an object property to not only state that its values are instances of a certain type, but also to specify a restriction on the property belonging to that type. DIP makes that possible by manipulating the format used for _uri_. Recall that the format of _uri_ is of the form "/_id/_name:_type". To specify a restriction on the value of _type, we simply manipulate the value of _uri. These are few examples:

- To specify any instance of type Field, we write the following: "./:Field"
- To specify any instance of type Field in document c54…789, we write the following: "c54…789/:Field"
- To specify an instance of type Field by name, we write the following: "./name: Field"
- To specify any instance of type Field, but with added restriction of having dataType to be string, we write the following: "./:Field; dataType = string"
- To specify any instance of type Field, but with added restriction of having dataType to be string *and* isFieldOf to be systemProperties, we write the following: "./:Field; _op= and; isFieldOf = systemProperties; dataType = string"
- To specify any instance of type Field, but with added the restriction of having dataType to be string *or* isFieldOf to be systemProperties, we write the following: "./:Field; _op= or; isFieldOf = systemProperties; dataType = string"

The above notations for property restrictions have several applications particularly in data modeling languages. For example, DIP data modeling language wishes to provide subclassing functionalities based on restriction of properties. Fig. 11 shows how this can be done using DIP.

```
[{
      "_id": "stringField",
      "_type": "Entry",
      "_name": "string Field",
      "_comment": "Field description",
      "extends": ["./:Field;dataType=string"],
   }]
```

Fig.11. Example use of property restriction

Fig. 11 shows the definition of type stringField which is the class of all instances of Field, with dataType = string. Suppose the existence of an instance of Field, but with datatype = number, that instance would not be a member of the type stringField.

8) *Indexing of DIP Documents*

It is common for cloud systems to offer indexing and search solutions. In this case, applications' data stored in the cloud are indexed in order to provide search functionality. To implement this functionality, cloud systems often allow applications to define their own index definition file. In this approach, each application would have its own index definition. DIP object model makes it possible to index all applications' data using a single index definition. The table header below shows a typical index definition for any DIP document. Other variation of this header which includes other columns is also possible provided that it maintains the columns shown in Table I.

Table I Index definitions for DIP documents

_id (key, string)	_type (key, string)	_name (key, string)	property (key, string)	value (multivalue ,string)	_ref (string)	simpleType

Table II shows the necessary indexed fields that comprise an index definition. The fields tagged as key form a compound key indicating a unique row in the table. The value field is tagged multivalued indicating that field's value is a list. The column labeled simpleType contains the type of the value when it is a simple type, e.g string, int, etc. The column labeled _ref is an indicator that the value is a pointer to another object. Table II partially shows the index data for Fig. 10.

Table II Index data of Fig. 10

Person	Entry	address	dataType	["string"]		str
Person	Entry	address	dataType	[" Address"]		str

D. *REST API for Create, Read, Update, and Delete Operations*

Web data protocols offer their consumers a REST API to create, read, update, and delete (CRUD) their data. DIP offers a fully functional REST API. In this paper, we provide a short summary of these operations in Table III.

From a conceptual perspective, DIP permits clients to create/get/delete a DIP document, and to update/insert/delete any object in the DIP document. In addition to working at the document and object granularities, DIP permits clients to get/update/insert/delete any properties in the DIP document. A few additional observations to make on the API:

- The HTTP request body for all HTTP Put/Post operations is a DIP document. Also, the HTTP response body for all HTTP Get, is also a DIP document. Therefore, whether retrieving a single object or the entire document, the structure of the reply is always the same.
- A HTTP Get request on URI /_id/_name:T would also resolve to URI /_id/_name:T$_1$ if T$_1$ extends T and so on. In that case, the returned DIP document contains the union of the operations on the two URIs.
- Upon receiving a HTTP Post/Put request on URI /_id/_name:T$_1$, API enforces that the instance is also of type T if T$_1$ extends T.
- Upon receiving a HTTP delete request on URI /_id/_name:T$_1$, API also executes delete operation on URI /_id/_name:T if T$_1$ extends T.

It is an error to create the same DIP document more than once. That is, once the document is created it can only be modified or deleted in subsequent operations.

Table III DIP operations using Rest

URL convention	HTTP verb	description
/_id	Post/Delete a Get	Create/Remove/Get DIP document
/_id/_name	Get/Post/ Delete	Retrieve/update/Delete the named instance
/_id/_name:type	Get/Post/ Delete	Retrieve/update/Delete the named/typed instance

IV. Performance Evaluation

For DIP to be successful, it must not add significantly to the latency of the NoSQL database. The majority of the latency introduced by DIP comes from the validation of DIP instances. The instance validation is comprised of 1) validate property disjointness ; 2) validate property intersection; 3) check property data types; 4) check for required properties; and 5) check property values and enumeration constraints. It is important to know that this validation time is a function of the number of properties in the object, and not the size of the instance. The tests were run on a personal computer running in Windows 7, with CPU I7-2640M, and 8GB of memory. Most of the overhead imposed by DIP takes place on update operations and not on get. Of that overhead, most of it is spent on validation. We observed that the validation time represents about 6% of our average update operation latency for payloads with less than 200 properties. For large sized payloads, e.g. the number of properties approaching 1000, the validation time becomes more significant. Therefore, these numbers clearly demonstrates that validation can happen synchronously, e.g. as part of the request workflow, for smaller payload size. However, for larger payload size, validation should take place asynchronously after a request has been acknowledged as successful by the server, e.g. by having the server responds to the sender with HTTP status code 202, and then continue with the validation. Furthermore, we acknowledge that these numbers can further be improved by utilizing more efficient algorithms for validation. This can be the subject of another paper.

V. Discussion

In this section, we describe our experience using the protocol. There are two internal projects that successfully used this protocol. In one of the project, we used DIP to express constraints related to cloud-based social platform. We found DIP to be very expressive, allowing us to state things like "all indices must define access control", "this mode of access control on this index excludes this other mode of access control on instance data", one can only define an index on this DIP property if it has been defined in the data model". Particularly useful was the use of meta-modeling in DIP enabled by the multi-typing of instances. Also in the same endeavor, disjoint property, disjoint object and intersection property and intersection object proved very useful.

Related to the stability of DIP as an abstraction layer for distributed databases, DIP once more showed its worth as we were able to migrate from our own instantiation of Voldemort (Sumbaly, et al., 2012) to DynamoDB on AWS without any impact on client applications.

Regarding the use of the URI convention, we received feedback that it would be best if the type of the object is represented as a matrix parameter. Representing the type in the URI is the result of identifying each object in the system by id and type. As we improved our system to identify object solely by ID, we kept the type in the path so as not to break backward compatibility.

Since in cloud system, the same data tend to live in distributed databases and on other clusters for further analysis by recommendation engines, it is therefore important that we are able to treat the DIP data as a graph. Note that as the data is first stored in the distributed database, instance validation plays a key role to ensure the quality of the data. However, as data is analyzed by recommendation engines, open world assumptions take precedence as we are interested in deriving knowledge from existing one. To this end, we translated DIP to Resource Description Framework (RDF) fairly easily. This is an important point as an earlier attempt for us to use RDF as the entry point into the cloud was met by strong resistance from developers who are unaware of RDF's inference rules.

Related to the use of the document structure as a list of objects, we also found that to be very convenient as it enabled an application to embed two different objects in the same document, therefore guaranteeing atomic transactions for a single key, even in the presence of a distributed database that does not offer this functionality. However, a limitation to this approach is the payload size at about 60 KB.

VI. Conclusion

In this paper, we presented the data interchange protocol for sharing data. As part of that, we presented the data interchange protocol object model, typing system, uniform indexing of data, and graph modeling of the DIP data. We also reported our experience with the protocol. Our next steps is to submit this as a draft standard to one of the standard bodies.

Acknowledgment

This work has benefited from the many attempts of design such a system internal to Nokia, starting with the work on the Unified API and culminating in the development of the Social [illegible]

References

[1] Chang, Fay, Dean Jeffrey et. al., "Bigtable: a distributed storage system for structured data," in 7th symposium on Operating systems design and implementation, Seattle, November, 2006.

[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value store," in Twenty-first ACM SIGOPS symposium on Operating systems principles, 2007.

[3] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, A. L. Y. Li and V. Yushprakh, "Megastore: Providing Scalable, Highly Available Storage for Interactive Services," in Conference on Innovative Data system Research (CIDR), pp. 223-234, 2011.

[4] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem and P. Helland, "The end of an architectural era: (it's time for a complete rewrite.," in 33rd international conference on Very large databases, p. 1150-1160, 2007.

[5] J. Starkey, "Database management system". USA Patent 8224860, 17 07 2012.

[6] "Amazon DynamoDB," Amazon, [Online]. Available: http://aws.amazon.com/dynamodb/. [Accessed 23rd July 2013].

[7] D. Gagne, M. Sabbouh, S. R. Bennett and S. Powers, "Using Data Semantics to Enable Automatic Composition of Web Services," in IEEE International Conference on Services Computing (SCC 06) Pg. 438-444, Chicago USA, 2006.

[8] "The application/json Media Type for JavaScript Object Notation (JSON)," IETF RFC 4627, 2006.

[9] "jvm-serializers Wiki," GitHub, [Online]. Available: https://github.com/eishay/jvm-serializers/wiki. [Accessed 23rd July 2013].

[10] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California Irvine-PhD dissertation, Irvine, 2000.

[11] "Google Data Protocol," Google, [Online]. Available: https://developers.google.com/gdata/. [Accessed 23 July 2013].

[12] "Open Data Protocol," Microsoft, [Online]. Available: http://www.odata.org/introduction/. [Accessed 23 July 2013].

[13] "Protocol Buffers," Google, [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed 23 July 2013].

[14] M. Shapiro, "Structure and Encapsulation in Distributed Systems: the Proxy Principle," in 6th International conference on Distributed Computing Systems, Cambridge, Ma, 1986.

[15] "A JSON Media Type for Describing the Structure and Meaning of JSON," IETF Internet Draft , 2011.

[16] "Resource Description Framework," W3C Recommendation, Cambridge, 2004.

[17] "Web Ontology Language," W3C Recommendation, Cambridge, 2004.

[18] M. Rodriguez, "Knowledge Representation and Reasoning with Graph Databases," [Online]. Available: http://markorodriguez.com/2011/02/23/knowledge-representation-and-reasoning-with-graph-databases/. [Accessed 23 July 2013].

[19] "Project Voldemort," [Online]. Available: http://www.project-voldemort.com/voldemort/. [Accessed 24 July 2013]..

Introduction to big data architecture

Why big data

- Volume
- Variety
- Velocity
- Extensibility (Schemaless)

Use case

- A company wishes to provide its employees medical coverage. So, they create medical plans tailored to the employees needs. Each plan consists of large number of covered services, e.g. acupuncture, physical, well-baby visits, emergency room visits, and so on. Additionally, each plan specifies the cost associated with that plan. For example, the co-pay for the various visits, and any deductible that should be met before the patient is reimbursed.
- the company has created a website for its employees where they can view each medical plan and the covered services associated with the plans. Additionally, the website is also used by the plan administrators to create new plans and modify existing plans.
- Is this a use case for big data?

How can we tell?

- Start by asking a few questions:
  what is the data size of a medical plan?
  What does a medical plan look like? That is,
  how can we model a medical plan?

  Other factors?
  How many people are viewing the website?
  E.g. throughput rates, latency requirements?

  Is there a need to batch import/export plans
  from the system?

Use case continued

- The analyst responsible for plan creation wants to quickly modify any plans that he created with additional attributes. For example, the analyst may want to remove services and add services to the plan. The analyst may also wish to extend any plan with additional attributes that may not have been foreseen during the design of the system.
- The question is: how can we extend the definition of a plan?

Technical requirements so far

- Need for data modeling
- Need for CRUD APIs
- Need for batch APIs
- Need for data extensibility
- Need for data validation

Use case continued

- While an analyst editing a plan, this plan must not be visible to employees. Furthermore, other analysts may view, but not edit, this plan
- Hence, the need to secure the system with authentication, and authorization support

Use case continued

- An employee using the system may find the medical plan that best fit his/her needs by using the search box. A user may search on any attribute

- Technical requirement:
  - need for search

Assignments

- Json schema: http://json-schema.org/
- Json: HTTP://json.org
- JSON Parser; JSON Simple
- Jsonpath
- Springboot
- marwansabbouh@gmail.com