JSON Web Tokens

An Introduction

Topics

1. Security Tokens
2. JSON Web Tokens
3. Specifications
4. JWT Signatures
5. RSA Keys
6. HMAC Keys
7. Claims
8. Scopes
9. Bearer Tokens

This presentation assumes an understanding of the *Crypto Concepts* workshop.

- symmetric and asymmetric encryption
- digital signatures

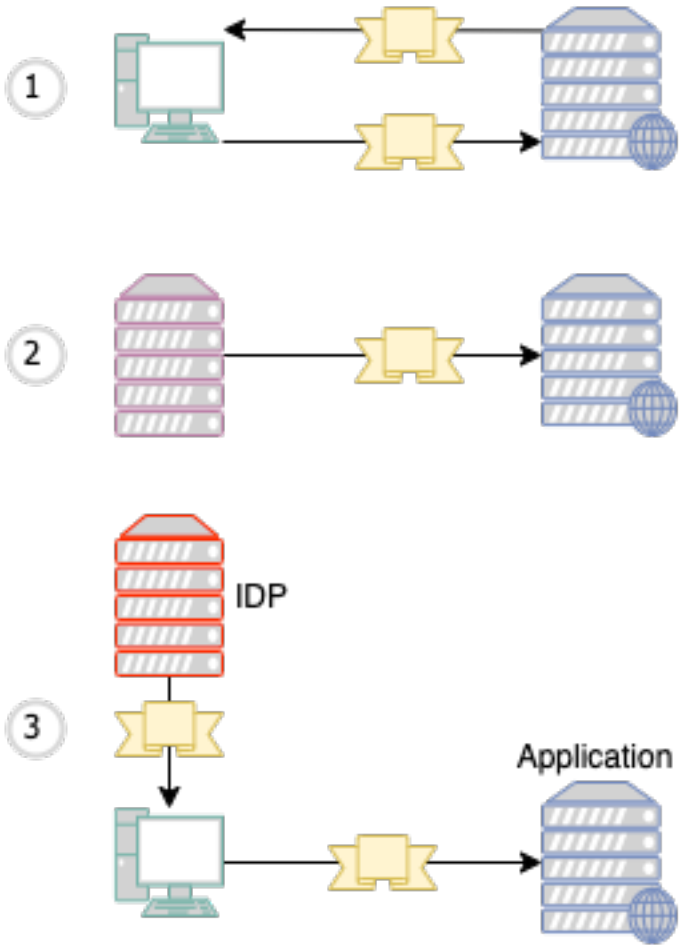
Also assumes basic knowledge of HTTP.

The **OpenSSL** command is helpful for some of the exercises; but is not required. For Windows downloads, see https://wiki.openssl.org/index.php/Binaries.

Security Tokens

- A token is a set of trusted claims about a user.
  - Claims must be trusted to be useful.
  - Source of trust is usually cryptographic.
- Token Sources
  1. Traditional web application stored tokens embedded in cookies into client browsers. The browsers send them back on subsequent requests. The trust comes from the application signing its own token.
  2. System-to-system tokens require the application to trust the signature of each application client. As the number of clients grows, the management of the trust relationships also grows.
  3. An Identity Provider (IDP) helps scale identity management. Only the IDP has to be trusted, regardless of the number of clients.

  Some applications support a combination of the three.
- Tokens assert the identity of the user, possibly along with characteristics such as roles. The application is responsible for turning token content into application privileges.

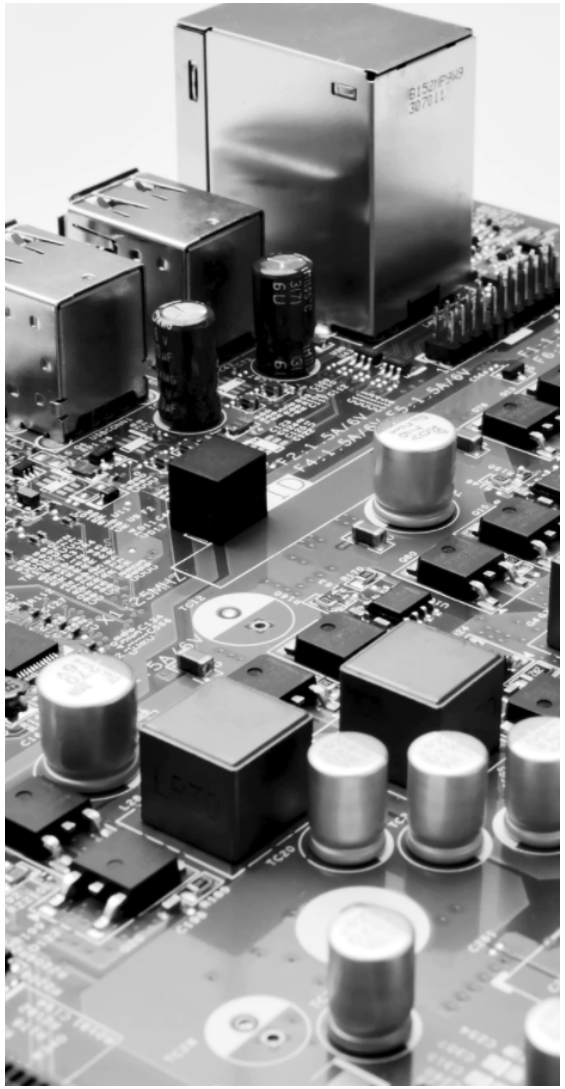


JSON Web Tokens

- JWTs have become the defacto token format for web applications.
- Its **decoded** format is JSON – easily manipulated by most programming languages.
- Its **encoded** format is a slight variant of Base64 encoding known as Base64url – ideal for inclusion in web URLs and HTTP headers.
- Figure on right shows decoded above, encoded below.
  - provided by `https://jwt.io`
  - header (in red)
  - payload (in purple)
  - Encoded JWT uses dots to separate components. The third component (blue) is a signature.

Specifications and Libraries

- Using a JWT as described by an application does not require a deep understanding of JWT. You only need to understand your JWT library.
- The JWT specifications are not difficult to read.
  - JWT (JSON Web Token) – https://tools.ietf.org/html/rfc7519
    - Provides explanation on the standard claims
  - JWS (JSON Web Signatures) – https://tools.ietf.org/html/rfc7515
    - Describes the different signature methods
    - Is the best source of examples
  - JWE (JSON Web Encryption) – this is much more involved than the other two. It is not often employed – encryption is usually left to the transport layer.
- Catalog of Libraries – https://jwt.io/#libraries-io
  - Over 100 libraries listed
  - Over 30 different programming languages



JWT Signatures

HMAC – Symmetric

- The shared secret can be any sequence of characters.
- Best to use a secret that is not easily recognized by casually viewing.
- Good source is https://www.grc.com/passwords.htm.
- Use header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

RSA – Asymmetric

- Only the public key is shared.
- Self-signed is fine; CN should identify the issuer, not FQDN of a server.
- Generate with openssl.
- Signature is larger than HMAC.
- Use header

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

Exercise 1 – Generate RSA Keys

The **openssl** command is required for this exercise.

1. Create a directory named `JwtWorkshop` and change to it using the command line. Verify you have access to the **openssl** command by checking the version. Your version may be different that the one shown.

```
JwtWorkshop$ openssl version
LibreSSL 3.2.2
```

2. Run the **openssl** command to create a new 2048-bit public/private RSA key pair.

```
JwtWorkshop$ openssl genrsa -out jwt-key.pem 2048
Generating RSA private key, 2048 bit long modulus
```

3. Generate an x509 certificate from your public key. Note that it is an interactive command.

```
JwtWorkshop$ openssl req -x509 -new -key jwt-key.pem
                -out jwt-cert.pem -days 1850
You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) []:US
State or Province Name (full name) []:California
Locality Name (eg, city) []:County of Los Angeles
Organization Name (eg, company) []:ISAB
Organizational Unit Name (eg, section) []:
Common Name (eg, fully qualified host name) []:My JWT Signer
Email Address []:
```

Notes:

- Not every option is required.
- Common Name is **not** a fully qualified host name.
- `-days 1850` means it's valid for 5 years.

Exercise 2 – Inspect x509 Certificate

The **openssl** command can also be used to list attributes of an x509 certificate. The term **x509** refers to the standard list of attributes. The beginning of the command is the same.

```
openssl x509 -in jwt-cert.pem -noout
```

- **x509** is the subcommand
- **-in** is the input certificate filename
- **-noout** means there is no certificate output

The final parameter depends on what you want to see. We'll investigate three of them in this exercise.

1. **-dates** – Placing this parameter at the end of the command lists the validity dates of the certificate.

```
JwtWorkshop$ openssl x509 -in jwt-cert.pem
                          -noout -dates
```

2. **-subject** – Remember the interactive parameters you had to provide to create the certificate? These combine to make the *subject*.

```
JwtWorkshop$ openssl x509 -in jwt-cert.pem
                          -noout -subject
```

3. **-text** – This parameter lists all the certificate details. It's usually more than you want.

```
JwtWorkshop$ openssl x509 -in jwt-cert.pem
                          -noout -text
```

HMAC Signatures

- HMAC secrets can be any string.
  - They can also be binary; but then you have to provide it base64-encoded to your library.
  - When doing simple tests, you can use easily remembered strings like "my hmac".
  - For broader use, use either a long phrase or randomly generated characters.
  - A good source of random HMAC passwords is https://www.grc.com/passwords.htm.

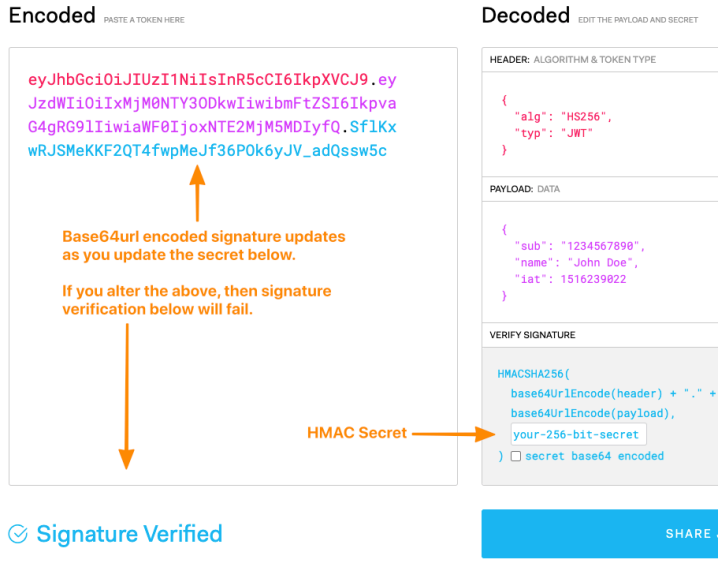
- Must be guarded closely for real applications.
  - Store them in restricted filesystem locations.
  - Choose so not easily retained when casually viewed.

Exercise 3 – HMAC

1. Open your Chrome browser to https://jwt.io and scroll to debugger.
2. Make sure **HS256** is selected. **H** = **H**MAC.
3. Replace the text

   `your-256-bit-secret`

   with your own secret.
4. Notice how the signature changes in the encoding section when you change the secret.
5. Delete a single character from the blue encoded token. Notice the signature is no longer valid.



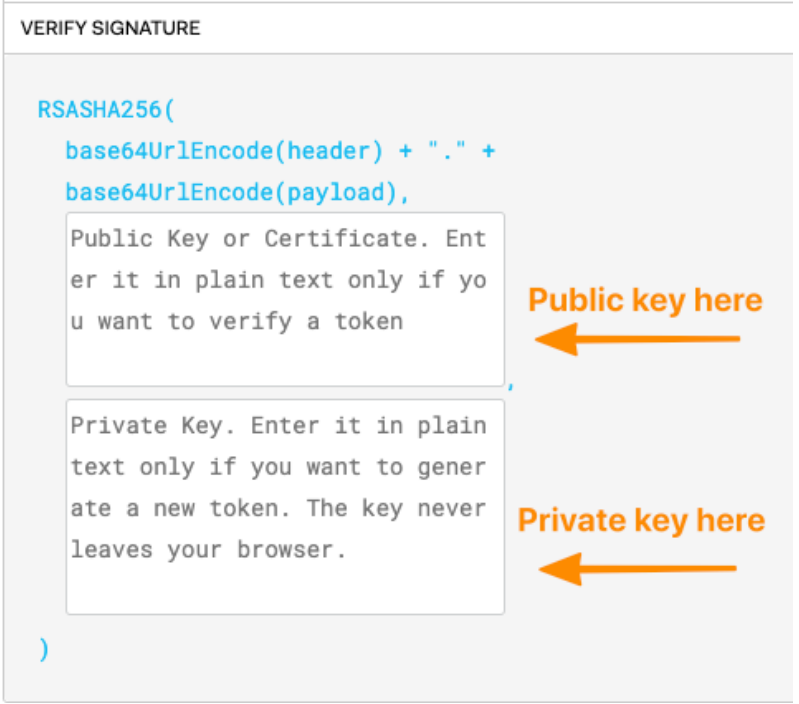
Exercise 4 – RSA Signatures

If you're coming here from the previous exercise, you can continue using the same web page.

1. Open your Chrome browser to https://jwt.io/ and scroll down to the debugger.
2. Make sure **RS256** is selected. **R** = RSA. This should cause the message Invalid Signature to appear in red at the bottom.

> Since we haven't provided a private key, there is no way to generate the signature.

3. List your private key content on the your command line; then paste it to Private Key box (the second box). This should allow the JWT to be **generated**. But notice that it is still **not verified**.
4. In a manner similar to the previous step, paste your public key into **Public Key or Certificate** box. You should see the Signature Verified message in blue.



5. Try tweaking a single character, either in the encoded box or the payload. What does this do to the validation?

Exercise 5 – Validating a JWT

1. Copy the encoded JWT on the right into the https://jwt.io **Encoded** box. Notice how it detects the HMAC signature.
2. Paste "`County of Orange`" in the entry field for the HMAC secret. It seems to validate, but that's only because it's altering the signature to match.
3. If you paste the encoded JWT from the right again, the signature won't match.

This shows us that to validate an HMAC secret, we must paste the secret first, then the encoded JWT.

4. Enter "`County of Los Angeles`" into the field for the secret.
5. Copy and paste the encode JWT again. It should be valid this time.

JWT with HMAC Signature

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ
9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZ
SI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWUs
ImlhdCI6MTUxNjIzOTAyMn0.151Nm8rnunx
acN0rzwZdtaFgdOC-cWeR-TpZB7SEy30
```

Validating a JWT signed with an RSA key is similar.

1. Paste the encoded JWT into the **Encoded** field.
2. Paste the issuer certificate into the **Public Key** field.

Claims

- The whole point of JWT is to reliably convey claims about a principal.
- Some claim names are standard.
  - `iss` – **iss**uer of the JWT
  - `sub` – **sub**ject, the asserted identity
  - `iat` – **i**ssued **at**, when the token was created
  - `exp` – **exp**iry, when the token expires
  - `aud` – **aud**ience, intended recipient of token
- The **`iss`** value is used by the token consumer to lookup the public key or secret associated with the signature. It must be settled on beforehand.
- The **`iat`** and **`exp`** values are *epoch timestamps*, i.e. number of seconds since midnight UTC, January 1, 1970.
  - Most programming languages provide utilities for generation and conversion of epoch timestamps.
  - Online: https://www.epochconverter.com/
- The **`aud`** claim can be used to restrict the consumers that attempt to validate the token. Not as common as the other four.
- The standard claim names are short in order to keep the size of the overall token small.
- There are many other standard claim names listed in the specification.
- Non-standard claim names may also be used. But they must comply with JSON rules.

The "name" and "admin" claims are non-standard, but perfectly reasonable.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

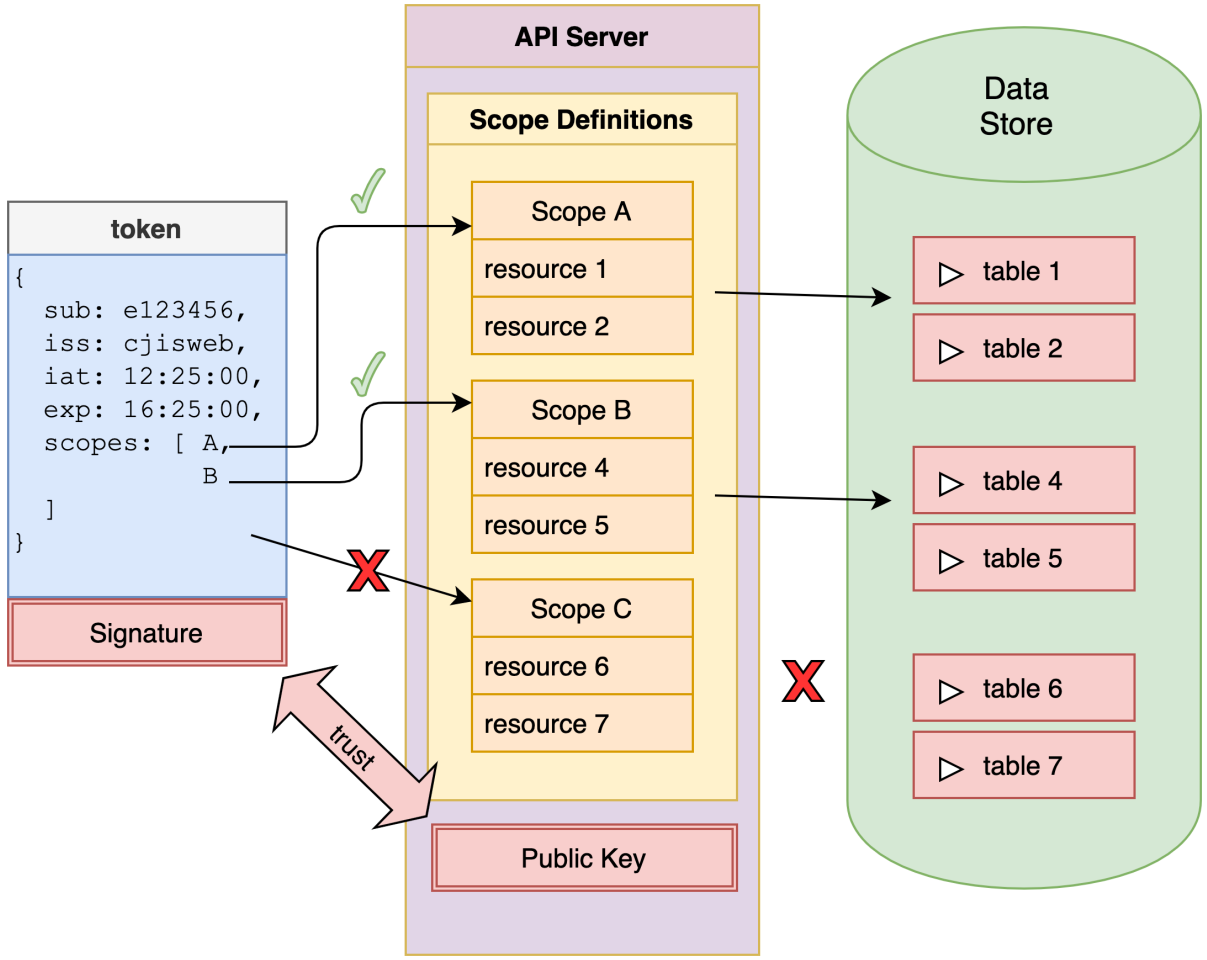
Scope

A **scope** represents a list of resources along with an access level for each resource. When a scope is associated with a user, the user assumes the access level of the scopes.

In the diagram, user e123456 has access to resources in scopes A and B, but not C.

The diagram does not illustrate access levels such as "update", "add", or "delete".

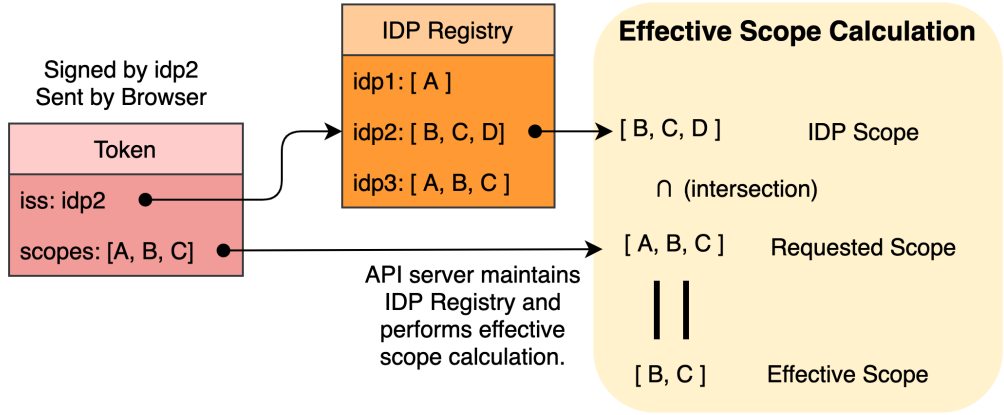
With HTTP, resource names and access levels are expressed as URLs and methods, respectively.



Effective Scope

Effective scope accounts for the fact that not all IDPs are equal. Some may be authoritative enough to assert any scope. Others may only be allowed to assert a limited set of scopes.

- Example – PIMS is a DA application that authenticates its users.
  - PIMS can invoke CJIS Tables on behalf of one of its authenticated users.
  - PIMS can associate scopes for the user in the JWT.
  - PIMS should only be allowed to include scopes admissisible for the DA. It should not be allowed to assert scopes for a public defender system.



The effective scope is the intersection of

- what the issuer asserts for the principal
- what the issuer is permitted to assert

Each application determines these limits for each issuer.

Bearer Tokens

A standard for submitting tokens in HTTP requests. A simplistic explanation is the following:

> A *bearer token* is the value of the `Authorization` HTTP request header in the format
> `Bearer` <token value>

That's like saying a REST service is JSON over HTTP. There is more to it; but at a high level, that's the gist.

Specification: https://tools.ietf.org/html/rfc6750

If your token value is "abcdef123456", then submitting it as a bearer token amount to adding an HTTP request header of

```
Authorization: Bearer abcdef123456
```

Many APIs that accept authentication tokens expect them as bearer tokens. Some implement the entire bearer token specification. Others simply expect the `Authorization` header in the format described.

JWT bearer tokens are now popular in OAuth 2.0 implementations. OAuth itself doesn't specify a token format (it's a message pattern specification). However, OpenID Connect (OIDC), the authentication protocol used by web login services such as Google, Facebook and GitHub specifies

- OAuth for the message pattern
- JWT for the token format
- Bearer token for the HTTP header

This has led to wide adoption of JWT bearer tokens even in the absence of OAuth.

Demo – Authenticate with JWT Bearer Token

The curl command to the left shows all the elements of an authenticated call. The space between "Bearer" and the token value appears as a new line. In practice, the entire command is a single line. Note:

- `Content-Type` header
- `Authorization` header
- Override default method to PATCH

```
$ curl -H "Content-Type: application/json"
       -X PATCH
       --data-binary '{"id": "826", "short_description": "UNLAWFUL LIQUOR SALES 1"}'
       -H "Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJ3b3Jrc2hvcCIsInN1YiI6ImU2NTQzMjEiLCJ
pYXQiOjE2MjAxNjc4MjEsImV4cCI6MTYyMDE3NTA2NywiZW5hYmxlZCI6dHJ1ZSwic2NvcGUiOlsic2hvcnR
kZXNjIl0sImF1ZCI6ImNqaXNhcGkifQ._6dRfbQUwBiGcQ0FQtRN2gzbVJpOHT29Y8v2ihe21Rg"
       http://localhost:6010/api/ChargeCode -i
HTTP/1.1 200 OK
X-Powered-By: Express
Access-Control-Allow-Origin: *
x-cjisapi-requestid: 1a18e780-ad2f-11eb-9a82-49ae19637dda
Content-Type: application/json; charset=utf-8
Content-Length: 17
Date: Tue, 04 May 2021 23:18:57 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{"changedRows":1}
```

The End

Roadmap

This figure summarizes available workshops that ISAB has delivered or has under consideration.

Contact Paul Glezen <pglezen@isab.lacounty.gov> for scheduling another delivery or development of a new workshop.