JavaScript による End-to-End セキュリティ

入門編

栗原 淳

September 3, 2019

はじめに

はじめに

この講義では

- End-to-End (E2E) セキュリティの原則と必要性
- Web サイトでの E2E セキュリティ実践のため、JavaScript での暗号の利用
  - ブラウザ側
  - サーバ側 (Node.js)

のさわりを学ぶ。

この講義の対象と事前準備

対象:

- 暗号・セキュリティ技術に興味がある初学者
- Web に暗号技術を導入したい Web 系のエンジニア

必須ではないが触って楽しむのには必要な事前準備:

- Git が使えるようになっていること
- Node.js が使えるようになっていること
- Google Chrome 系ブラウザ and/or Firefox が利用可能なこと

モダン Web サイトと End-to-End セキュリティ

Web サイトにおける昨今の情勢

- EU における General Data Protection Regulation (GDPR) の施行 (2018 年)
- GDPR に続いて、カリフォルニア、南米、オセアニアで類似の法律の制定の動き
- 日本においても、2020 年に個人情報保護法の改正法案提出の見通し

⇓

企業にとって、「正しく」「強固」に
ユーザデータ、ユーザプライバシを保護することは必須の事項

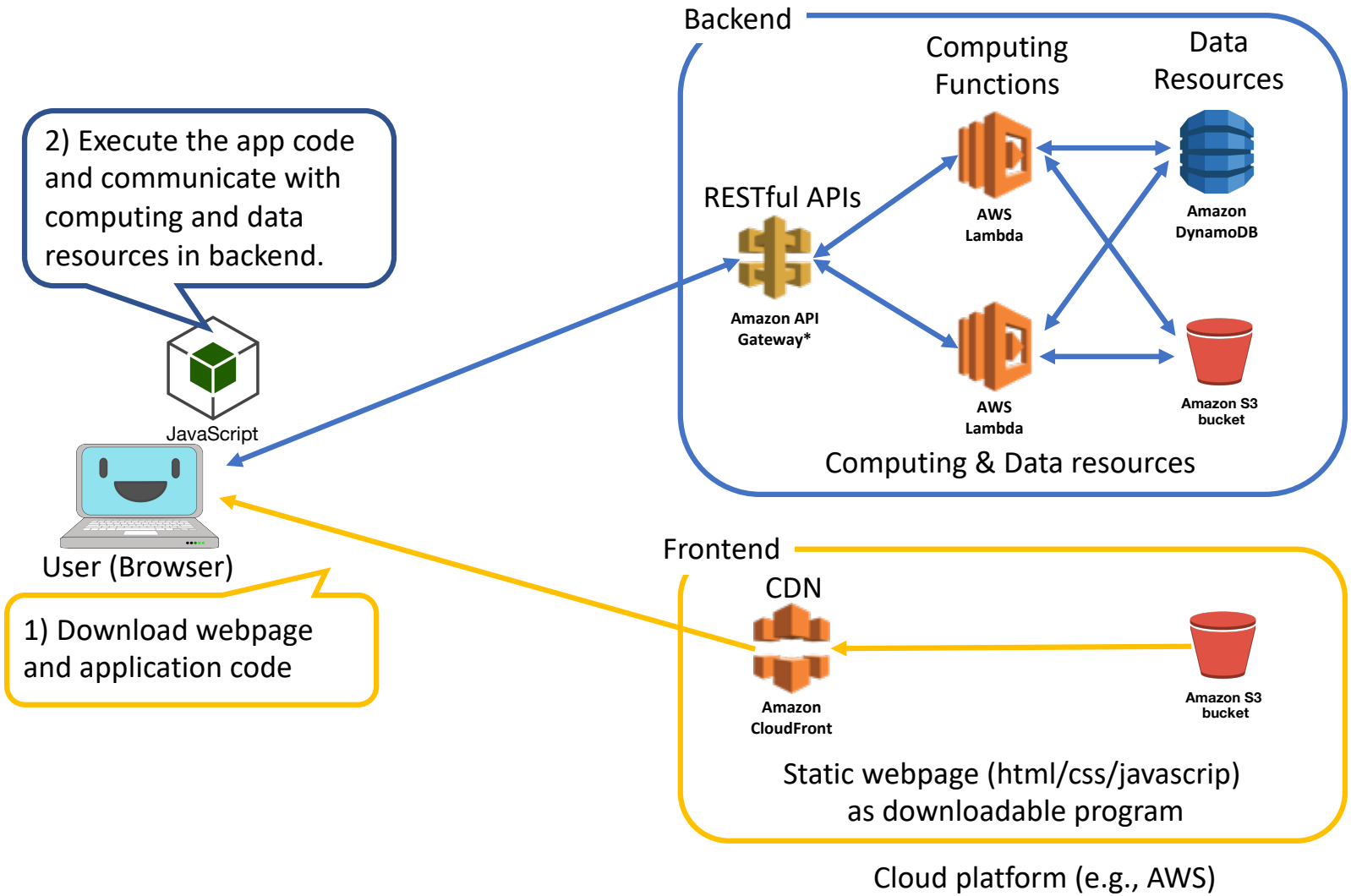
最近流行りの **Web** システム

- クラウドプラットフォーム上で構築
- 「サーバ」のない (サーバレス) 構成
- JavaScript (ReactJS など) を多用した、Single Page Application 構成

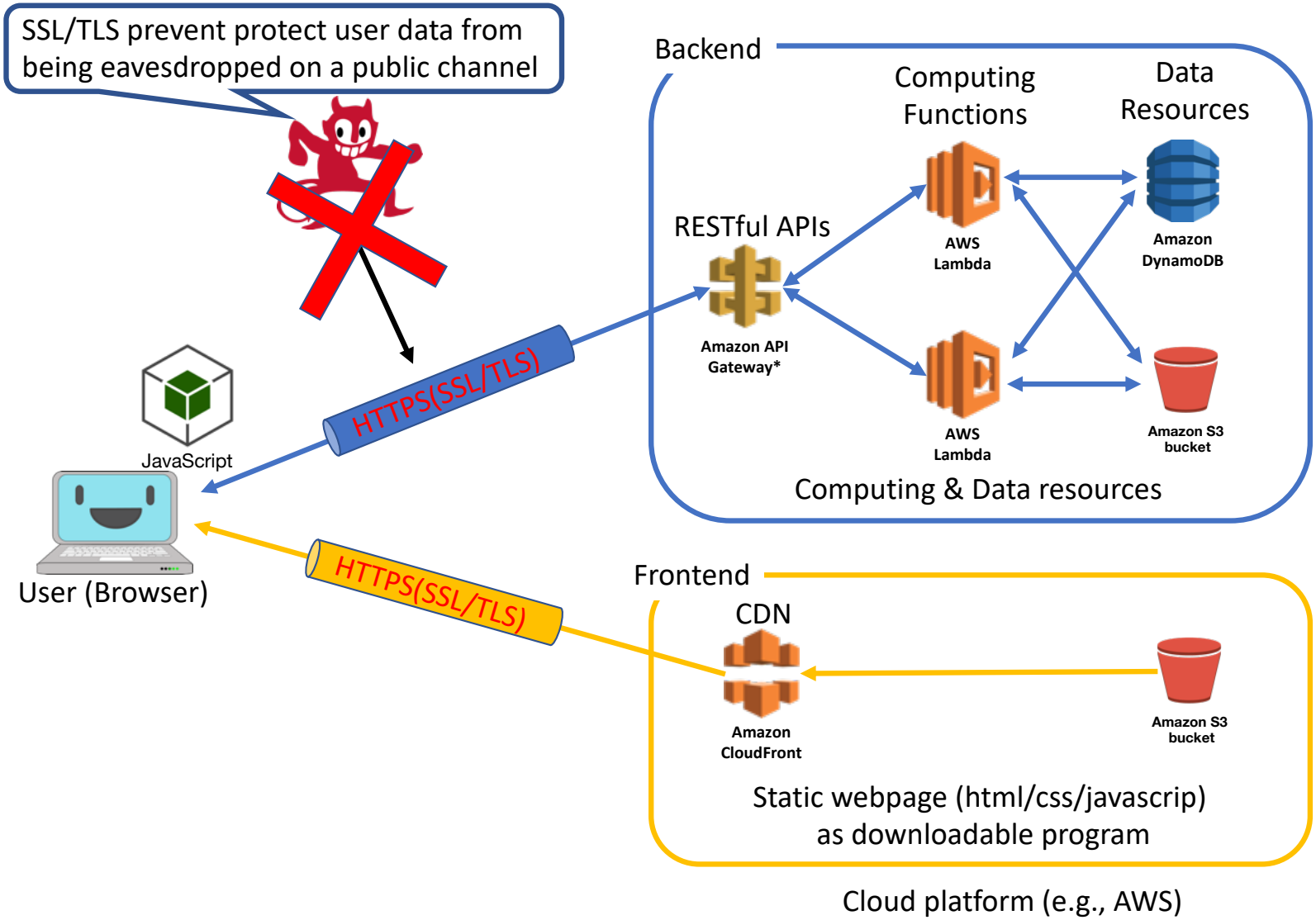
⇓

ユーザの手元で計算を実行する機会の増加

AWS を例にした典型的な構成:



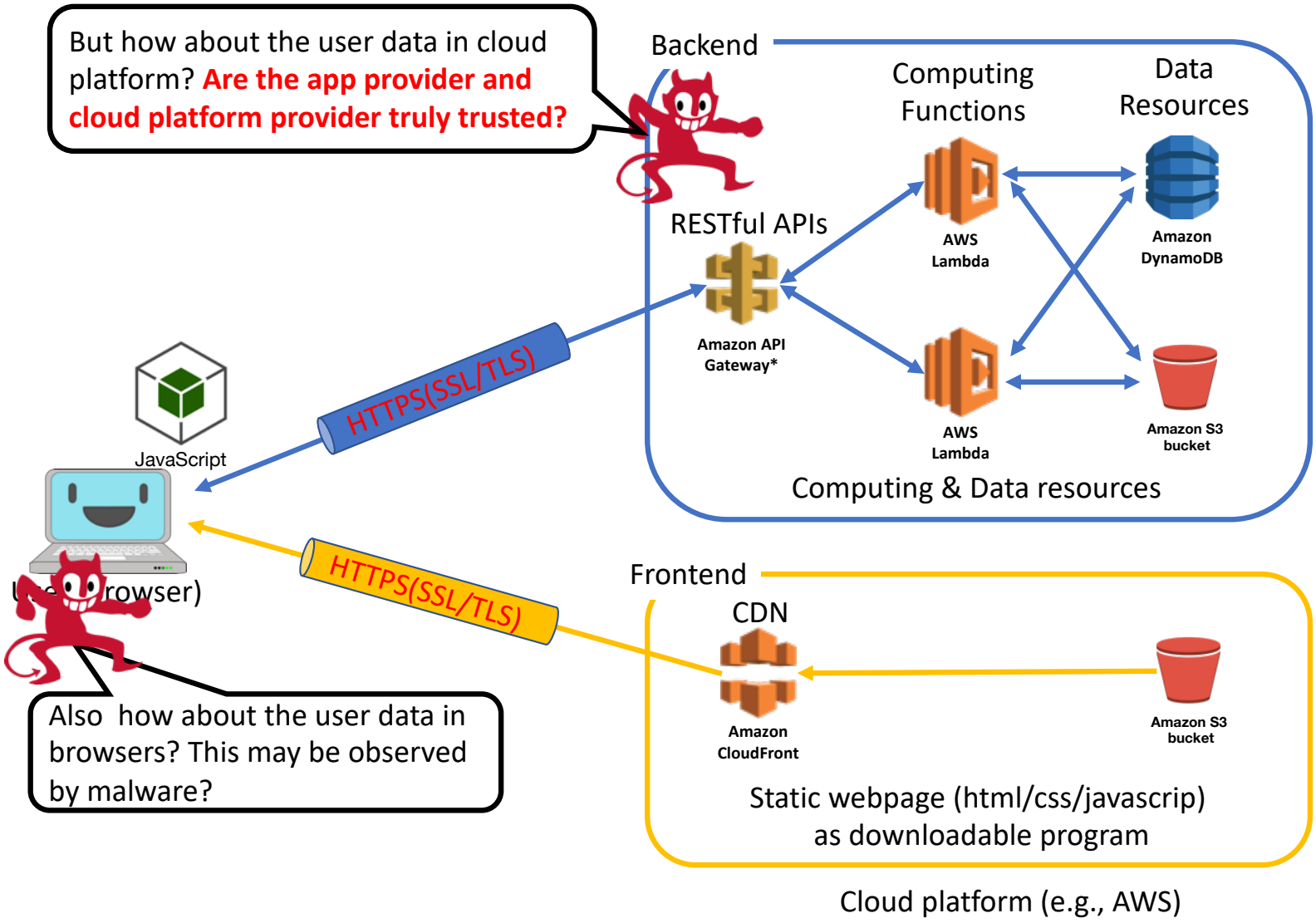
AWS を例にした典型的な構成:



通常、ユーザ・クラウド間の HTTP 通信路は SSL/TLS で保護

⇒ HTTP 通信路から外部の盗聴者へのユーザデータ漏洩を防止

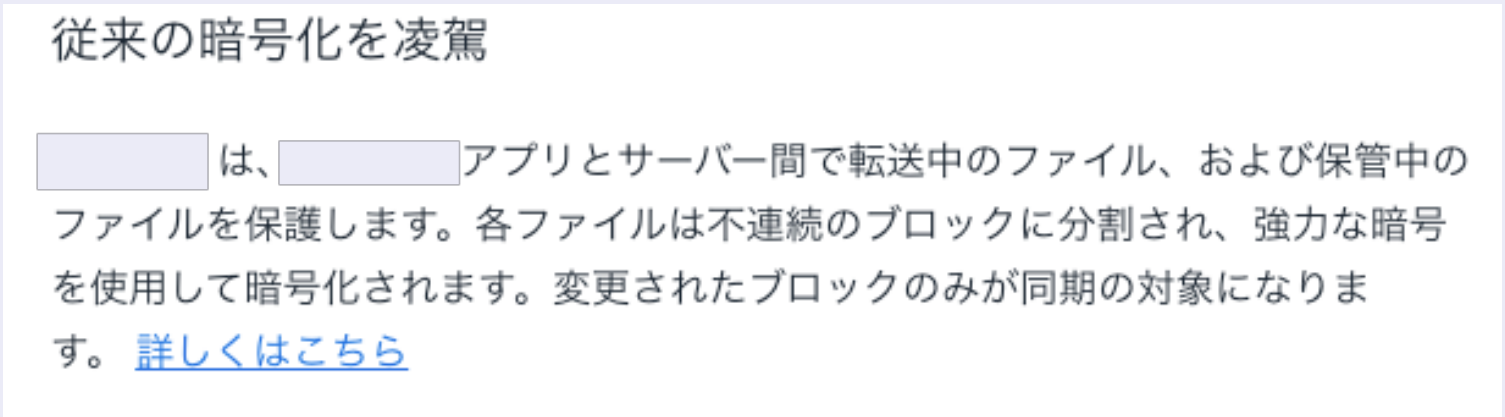
AWS を例にした典型的な構成:



しかし、クラウド PF 内・ブラウザ内のデータ保護は……？

「暗号化しているから安全です」という叙述トリック

(Web とはちょっと違いますが…) 某クラウドストレージ事業者の例



クラウドベースの (Web) サービスでよくある文言：

- (SSL/TLS で) 転送中のデータを暗号化して保護
- ストレージに保存されるデータは暗号化して保護

- (SSL/TLS で) 転送中のデータを暗号化して保護
  ⇒ 公開通信路の盗聴からデータを保護
- ストレージに保存されるデータを暗号化して保護
  ⇒ ストレージ自体が盗まれた時や、第三者のストレージを使っている場合のデータ漏洩を防止

いずれも事業者に対しての秘匿性を担保しているわけではない[1]

⇓

(望む・望まないにしろ) 事業者はユーザデータを不必要に取得

[1] 事業者はデータを見放題ということ。

このようなクラウドサービス・Web App を作ることは：

- ユーザにとって：共有不要な相手とデータを共有している
- 事業者にとって：昨今のプライバシ・セキュリティ要求の高まりから、無用なリスクを背負いこむ可能性が大

今後、Web App を作っていくにあたって

「必要な相手とだけ」確実に・正しく、データを共有できるように、適切なデータ秘匿が必要 (不必要にデータ取得しない)

データの秘匿性・プライバシを謳うサービス

- Tresorit[2]:
  事業者・サーバに情報が漏れないことを謳ったクラウドストレージサービス。Dropbox に近い。
- KeyBase[3]:
  事業者・サーバに情報を漏らさず、メッセージ・ファイル共有（クラウドストレージ）が可能な SNS。
- Signal[4]:
  事業者・サーバに情報を漏らさないメッセージング・通話アプリケーション。「最も安全な」チャットサービスと呼ばれており、各類似サービス (WhatsApp など) にプロトコルを提供。

北米・EU 共に、スノーデンの事件以降、事業者にも情報を与えない End-to-End 暗号化を謳ったサービスが強く注目を浴びている。

[2] https://tresorit.com/
[3] https://keybase.io/
[4] https://signal.org/

End-to-End セキュリティとは

End-to-End (E2E) Principle[5]

The end-to-end principle is a network design method in which application-specific features are kept at communication end points.

(アプリケーションの機能はネットワークシステムの終端で実装されるべきという原則)

[5] J. H. Saltzer et al., “End-to-End Arguments in System Design”, in Proc. ICDCS 1981, pp. 509–512, Apr. 1981.

アプリケーションのセキュリティについての E2E Principle:

End-to-End (E2E) セキュリティ

情報を共有する主体同士 (End-to-End) について、セキュリティ3原則を担保する。

- 情報の秘匿性 (主体のみで情報を共有可能) ⇒ E2E 暗号化
- 主体・情報の真正性 (主体が生成した情報であることを証明)
- 情報の可用性 (主体同士が正しく情報を利用可能)

ポイント：アプリケーションにおいて「情報を共有する主体」は一体何か。

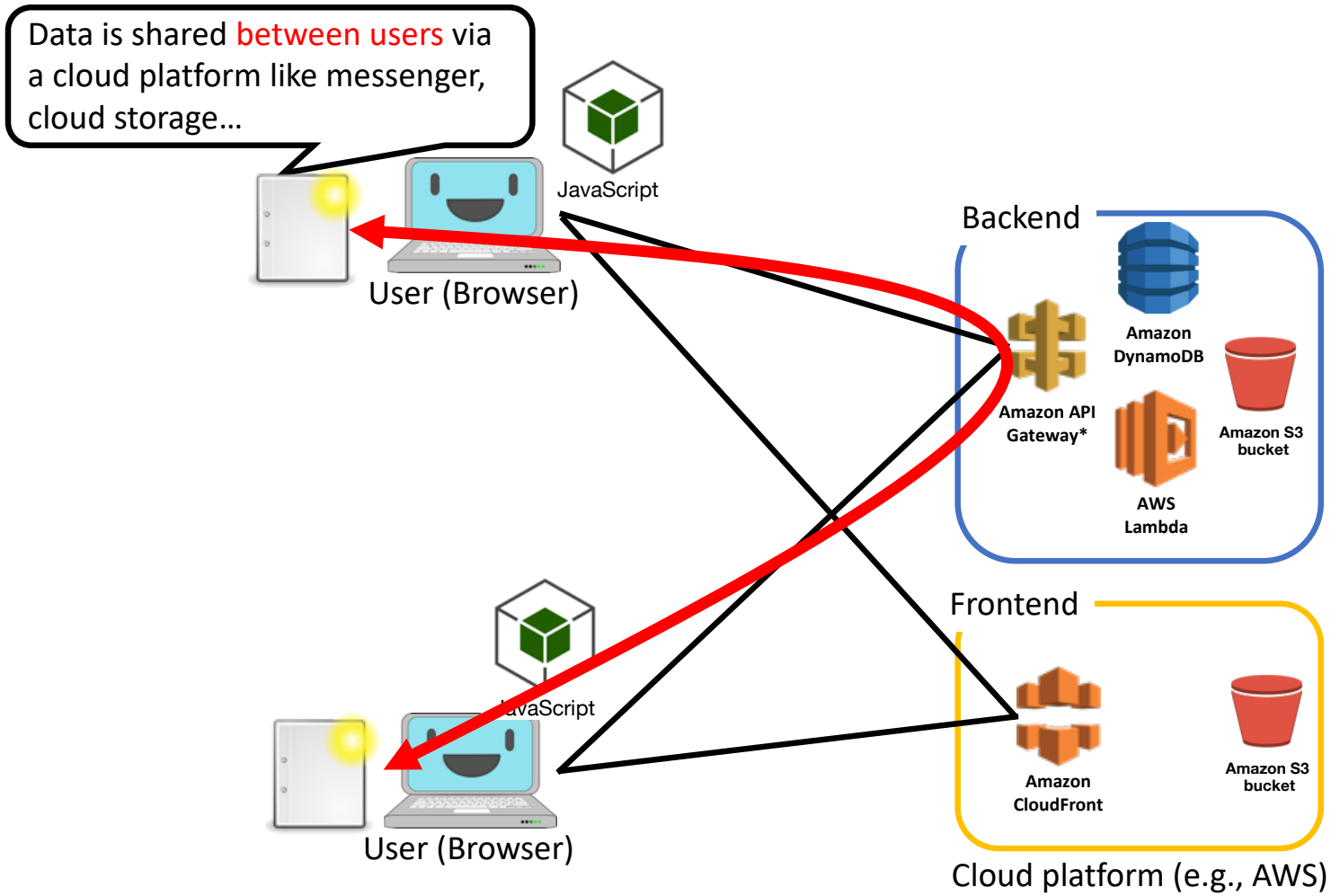
- アプリケーションを利用するユーザ同士？
- サーバ・クライアントアプリ同士？
- 他？

では、Web アプリケーションにおけるエンドポイントとは？

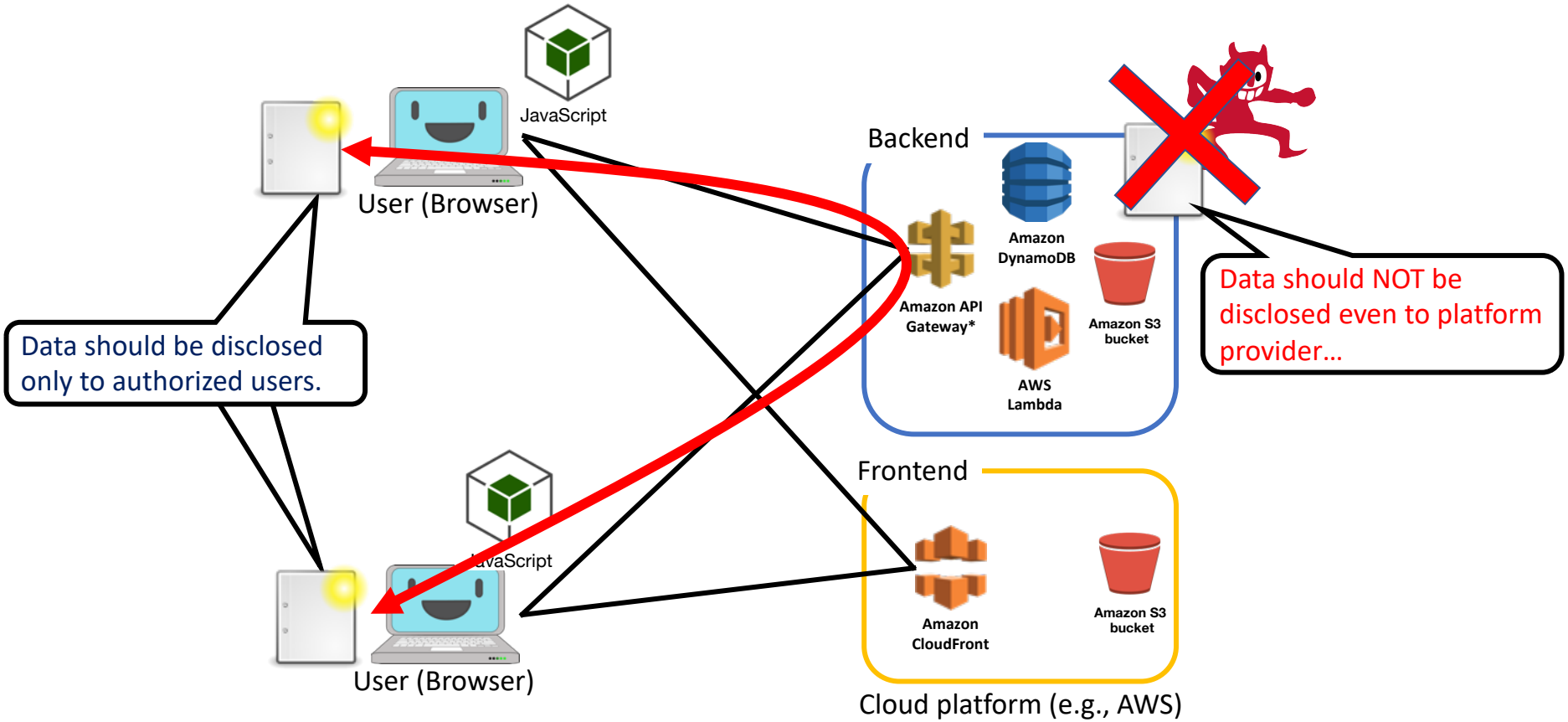
- ユーザ側エンド:
  ⇒ JavaScript/HTML/CSS など「ブラウザ内で実行・レンダリングされる」Web フロントエンド要素[6]
- もう一方のエンド:
  ⇒ Web アプリケーション次第で変化

[6] 実際にユーザが触れる要素がエンドポイントであって、Web サービスにおいては、端末やブラウザはエンドポイントにならない

例: クラウドプラットフォームを介し、ユーザ同士が情報を共有する主体の場合 = ユーザのブラウザ同士が両エンド

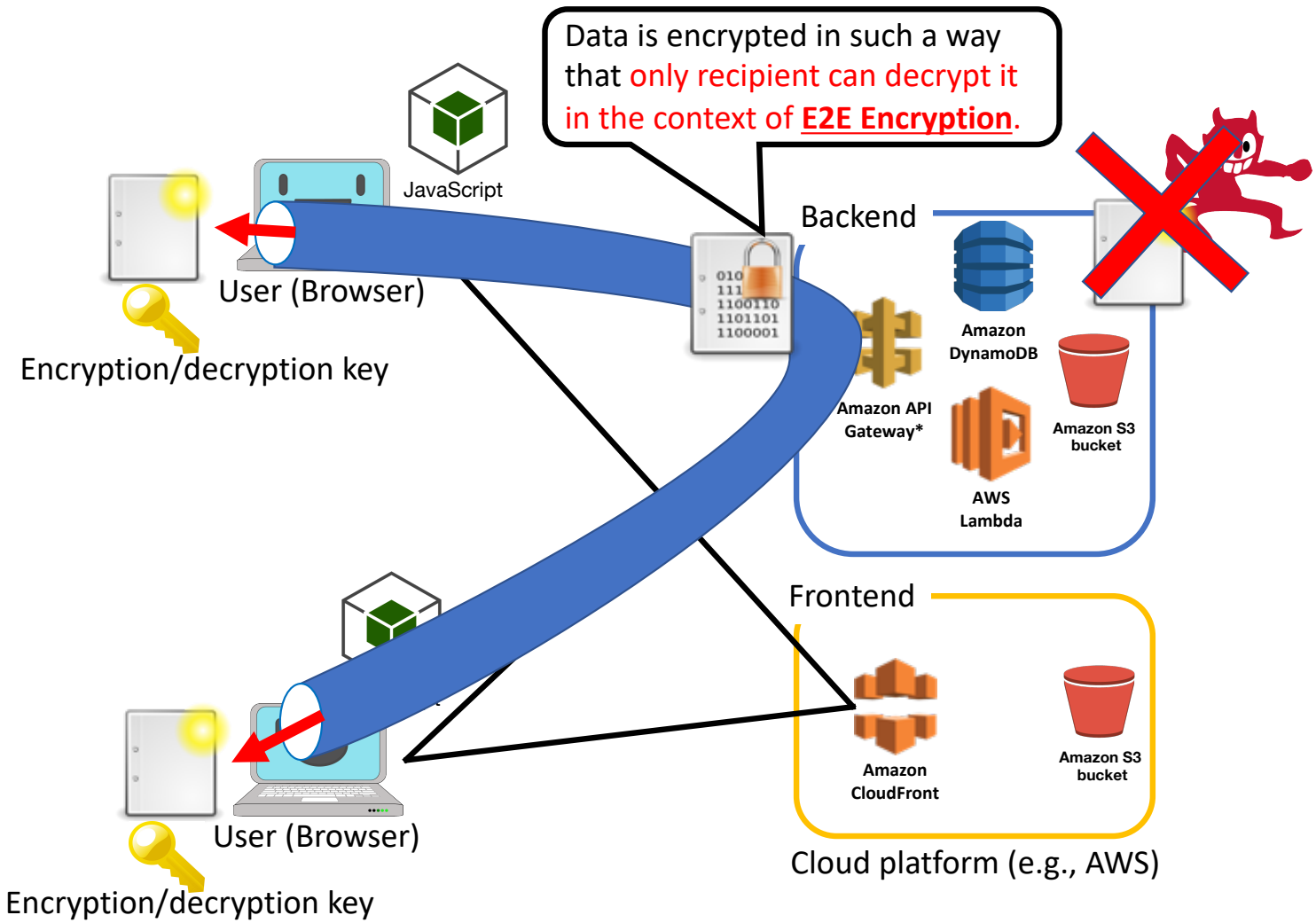


E2E セキュリティの観点からは、クラウドプラットフォーム自身ですら情報を取得可能であるべきではない



※ユーザがサーバとのみやり取りする場合、一方の主体 (End) はサーバとなる。

この例の場合、E2E セキュリティのためには、情報を共有する主体同士のみが復号可能なよう、やり取りするデータを全て暗号化する (E2E 暗号化)



E2E セキュリティを実現するには、構築するアプリケーション毎に「情報を共有する主体」を正しく定義する必要。

E2E セキュリティを実現しつつ、「事業者にとってリスクとなるような情報はなるべく不用意に取得しない」ことも重要。

Web App における End-to-End セキュリティ

改めて…

サービスで E2E セキュリティを考える意味

- ユーザプライバシの保護
- 事業者側のリスクの低減

Web アプリケーションにおける E2E セキュリティ

少なくとも片方のエンドは Web フロントエンド要素
⇒ Web フロントエンドでセキュリティを担保する方法が必要

と、いうわけで、今回は「Web のための E2E セキュリティ」として JavaScript におけるデータの暗号化 (E2E 暗号化) の「さわり」を紹介します。

JavaScript で暗号を使ってみよう [基礎編]

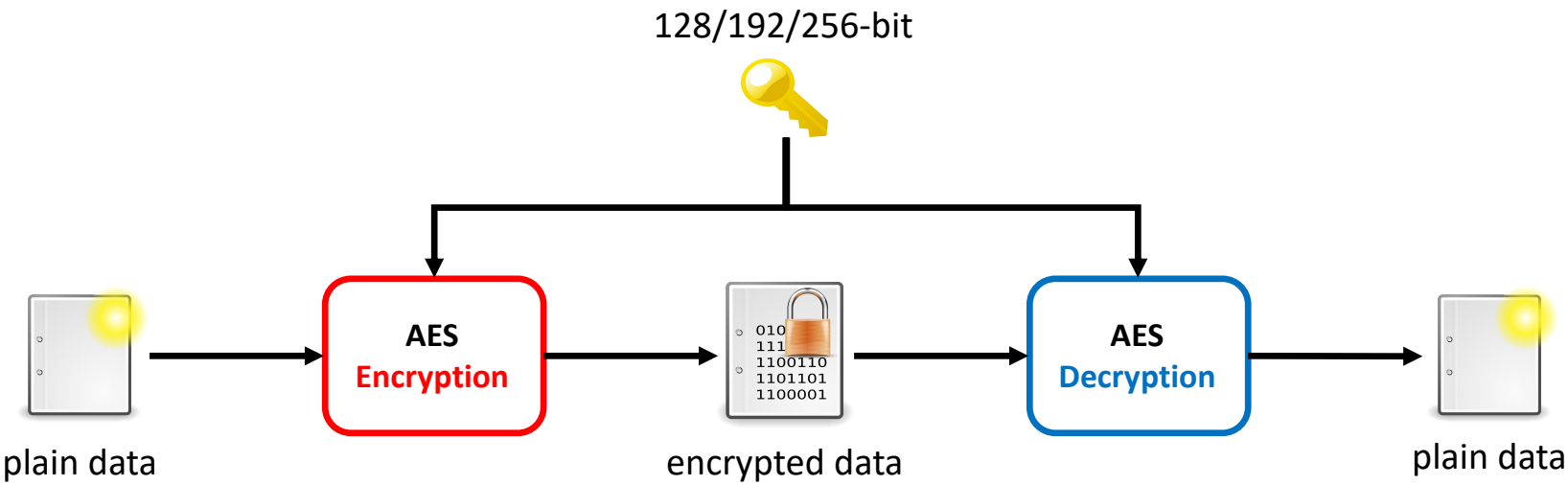
–今回はお試しで AES–

AES (Advanced Encryption Standard) とは？

AES

米国 NIST の標準暗号アルゴリズム (※共通鍵暗号)

- 鍵長は 3 種類: 128-bit, 192-bit, 256-bit
- 欧州 NESSIE、日本 CRYPTREC などの標準規格としても採択
- 現在まで致命的な欠陥は見つかっていない、安全性の高いデファクトスタンダードのアルゴリズム



共通鍵暗号・公開鍵暗号・ハッシュがどうのとか、それらを組み合わせて安全に使うためにはどうするかとか、そういう話は今回はしない。

とりあえず「AES という暗号アルゴリズムがあるんだ」くらいで十分。今回、まずは「とにかく JS で AES を使ってデータを暗号化してみること」から始める。

JavaScript における暗号の利用環境

一般的な統合ライブラリは C で書かれた OpenSSL だが、JavaScript から直接呼び出すことができない。

JavaScript から利用可能な暗号の統合ライブラリ:

- `WebCrypto API` (ブラウザ)[7]:
  W3C にて標準化が進む Web API (ブラウザのネイティブ API)。
- `Crypto` (Node.js)[8]:
  Node.js 環境にて利用可能な暗号ライブラリ (Node.js のネイティブ API)。OpenSSL のラッパー。
- `sjcl` (Node.js/ブラウザ)[9]:
  Stanford 大学暗号研究室で開発された pure JS なライブラリ。

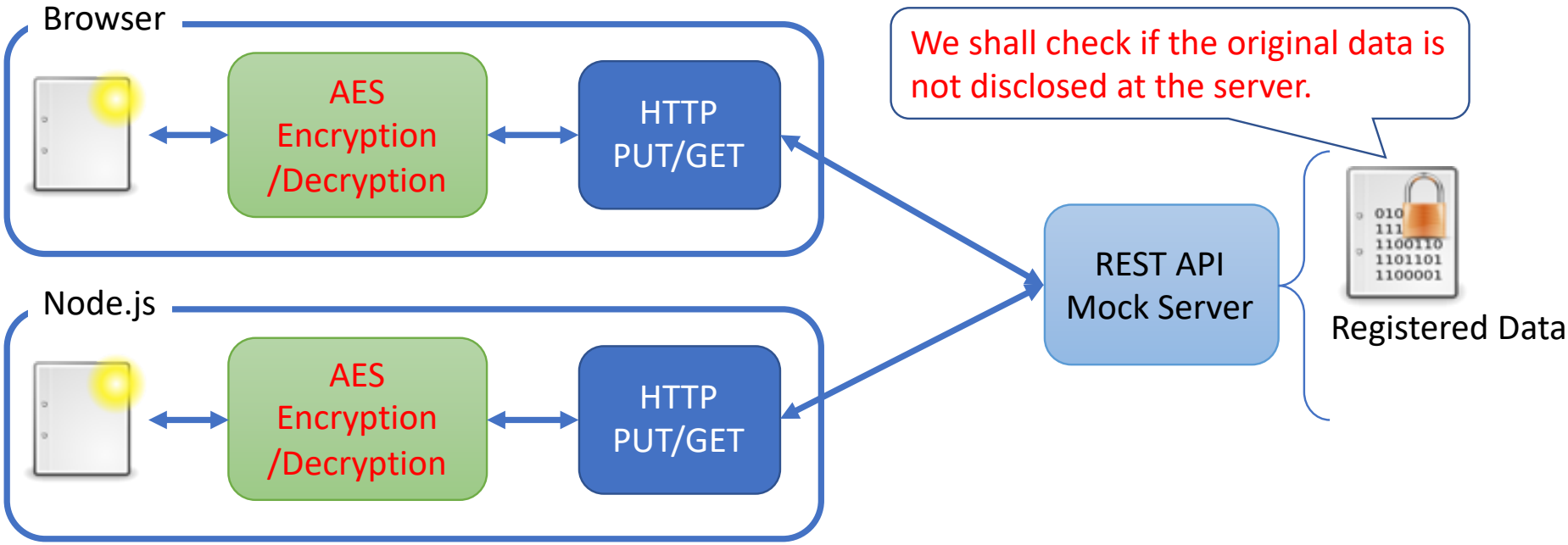
[7] `https://www.w3.org/TR/WebCryptoAPI/`
[8] `https://nodejs.org/api/crypto.html`
[9] `http://bitwiseshiftleft.github.io/sjcl/`

今回は、高速動作が期待されるネイティブ実装な統合ライブラリ 2 つを例にとる。

- WebCrypto API → ブラウザ向け
- Node Crypto → サーバ/スタンドアロン S/W (Node.js) 向け

今回やってみること

ブラウザ・Node.js をエンドとし、REST API で暗号化データを登録してみる。



単なるデータの HTTP PUT/GET の前段・後段に AES 暗号化・復号部分を組み込んでみる。
※モックサーバはローカルホストで立ち上げ。

環境

以下の環境が前提:

- Node.js (> v10) がインストール済 [10]
- ブラウザとして、Google Chrome (系ブラウザ)、もしくは Firefox がインストール済み
- Visual Studio Code や WebStorm などの統合開発環境がセットアップ済みだとなお良い。

[10] npm に加えて yarn も使えるとなお良い (インストールコマンド: npm i -g yarn)

JavaScript プロジェクトの準備

- プロジェクトの GitHub リポジトリ [11] を Clone

```
$ git clone https://github.com/zettant/e2e-security-01
$ cd e2e-security-class/sample
```

- 依存パッケージのインストール

```
$ yarn install  or  npm install
```

- ライブラリのビルド

```
$ yarn build  or  npm run build
```

[11] https://github.com/zettant/e2e-security-01

REST API モックサーバの準備

今回は SSL 接続可能な共有サーバを準備済
(`https://e2e.zettant.com/`)。

別途、検証用のサーバをローカルで立ち上げ可能。

モックサーバの立ち上げ

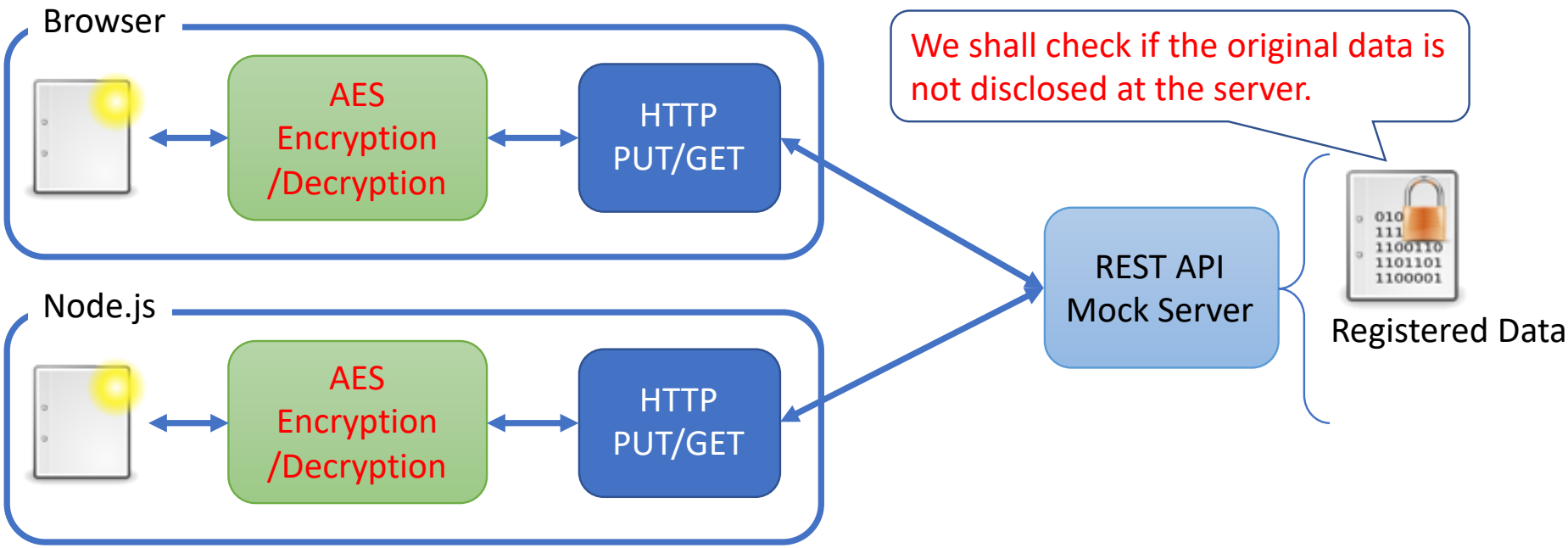
```
$ yarn start
```

起動すると、localhost の 3000 番ポートで HTTP リクエストを待ち受け開始する。

とりあえず試してみる: コマンドライン編

とりあえず試してみる: コマンドライン編

JavaScript (Node.js) をコマンドラインから実行してみて、動作イメージを確認してみる。



まずは生データを POST してサーバに登録してみる。

```
$ pwd
.../e2e-security-class/sample ←場所を確認

$ yarn execute post -r 'hello my secret world!!!' ← 登録 [12]
Register plaintext data to remote server
Data:  hello my secret world!!!
Registered id:  1 ← id=1 番として登録したというメッセージ
```

登録したデータを生のまま取得してみる。

```
$ yarn execute get -r 1 ← id=1 番の登録データを取得
Retrieve plaintext data to remote server
Registered Id:  1
Retrieved data:  hello my secret world!!! ← 取得した 1 番のデータ
```

[12] -r を抜くとローカルのモックサーバに接続

データを AES 暗号化したあと POST してサーバに登録してみる。

```
$ yarn execute post \
 -e -k 'my secret key' \ ← my secret key という鍵で暗号化
 -r 'hello my super secret world!!!'
Register encrypted data to remote server
Data:  hello my super secret world!!!
Key:  my secret key
Note:  Derived key binary in base64:
o12da4tawTjHd4woS+DeexcCR9EOyVKml6gOzZkGzDc=[13]
Registered id:  2 ← id=2 番として登録したというメッセージ
```

[13] 実際に使う AES 鍵のバイナリ表現 (base64)

暗号化して登録したデータを生のまま取得してみて、暗号化されていることを確認する。

```
$ yarn execute get -r 2 ← id=2 番の登録データを取得
Retrieve plaintext data to remote server
Registered Id:  2
Retrieved data:  EoeSsv5BFr6s1jZh3iMM1Pxa+wA4UxQnM30J2027kJU= ←
取得した 2 番のデータ
```

Base64 エンコードされているが、暗号化されており中身はまったくわからない。[14]

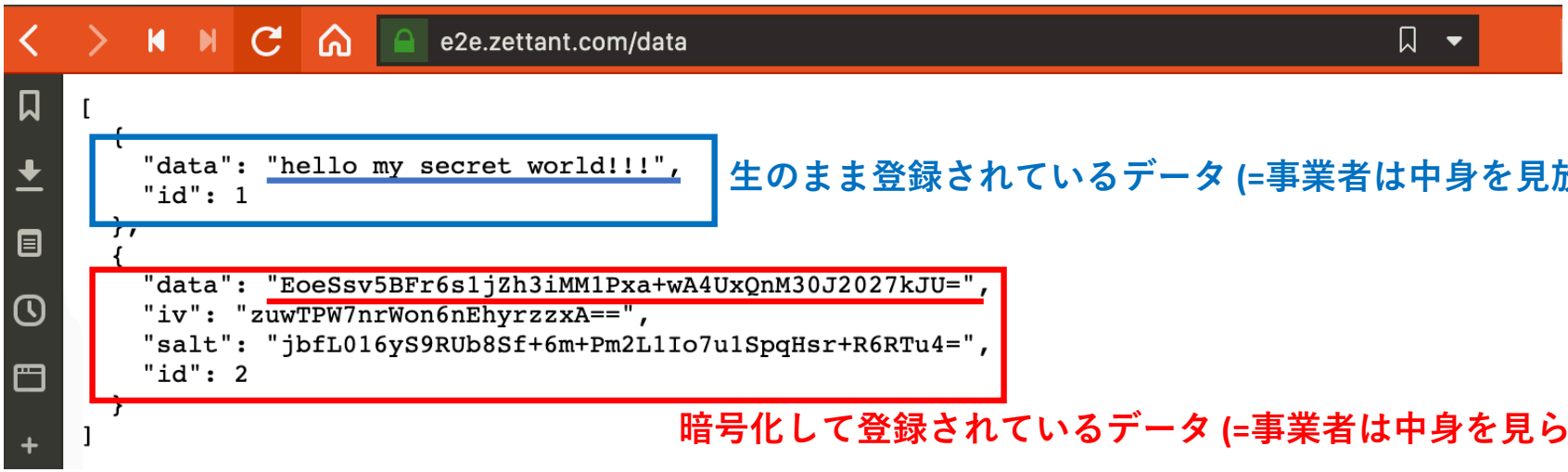
[14] Mac/Linux の場合、ターミナルで “echo [base64str] | base64 -D -” と実行してみると訳のわからないデータにデコードされることが確認できる。

取得した後、復号してみる [15]

```
$ yarn execute get \
 -d -k 'my secret key' \ ← my secret key という鍵で復号
 -r 2 ← id=2 の鍵を取得
Retrieve encrypted data to remote server
Id:  2
Key:  my secret key
Note:  Derived key binary in base64:
o12da4tawTjHd4woS+DeexcCR9EOyVKml6gOzZkGzDc=
Decrypted data:  hello my super secret world!!! ←正しく復号された
```

[15] key が間違っていると失敗する

https://e2e.zettant.com/data にアクセスすると、サーバでの登録状況が一覧できる。[16]



[16] ローカルホストに立てた場合は http://localhost:3000/data

とりあえず試してみる: ブラウザ編

とりあえず試してみる: ブラウザ編

1. `samples/src/post-get-browser.html` をブラウザで開く。
2. 真っ白な画面で「開発者ツール」を起動[17]、Web コンソールを表示する。

コマンドラインでやった「暗号化して登録→取得して復号」の一連の流れが初期設定のデータ・鍵を使って実行される。

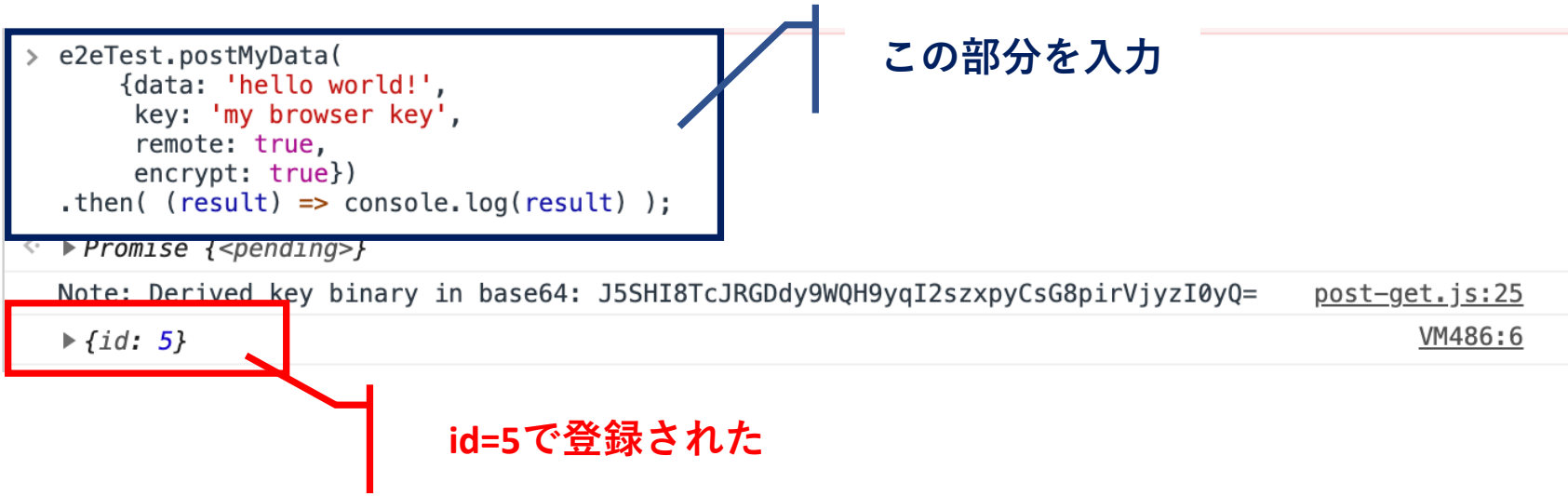
[17] Chrome: Ctrl(Cmd)+Alt(Opt)+i

以下の２パターンで色々試すことができる。

- `samples/src/post-get-browser.html` の中身を編集→リロード
- `samples/src/post-get-browser.html` を開いたまま、開発者ツールの Web コンソールから POST 関数と GET 関数を実行。

ここでは後者を試してみる。

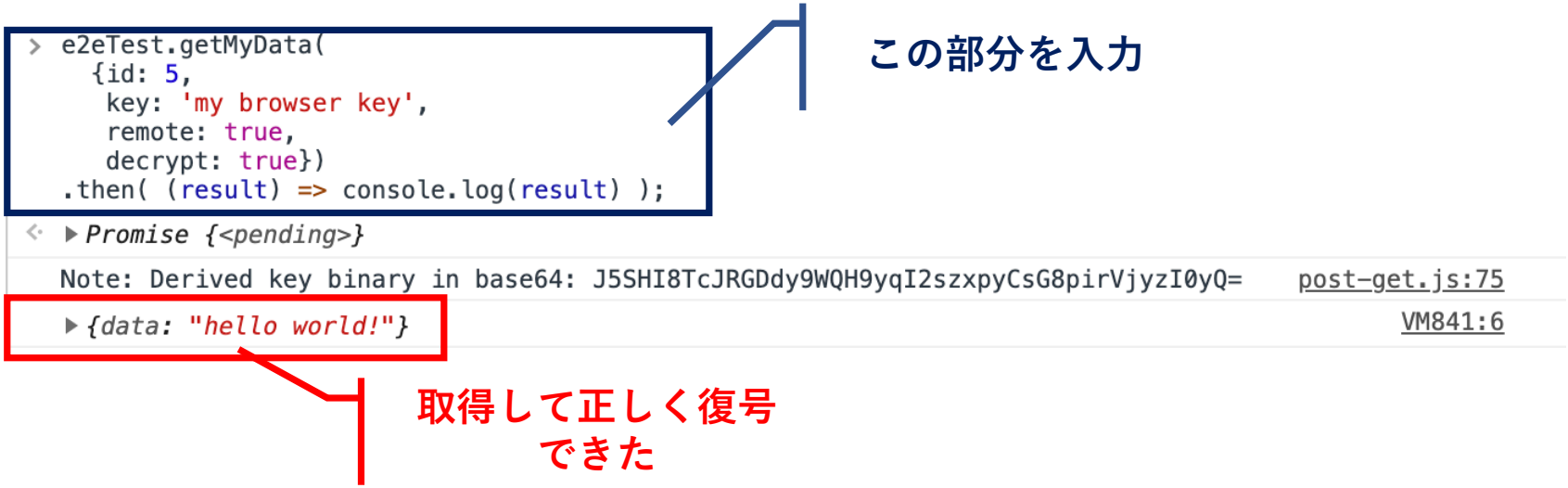
Web コンソールから暗号化・登録してみる。




```
e2eTest.postMyData(
    {data: 'hello world!', ← 暗号化するデータ
     key: 'my browser key', ← 鍵
     remote: true, ← リモートサーバの場合 true
     encrypt: true}) ← 暗号化する場合 true
.then( (result) => console.log(result) );

～中略～

VM486:6 {id:  5}
```

コマンドラインと同じく、https://e2e.zettant.com/data を表示すると暗号化されて登録されていることがわかる。

今度は Web コンソールから取得・復号してみる。




```
e2eTest.getMyData(
  {id: 5, ← id=5 のデータを取得
   key: 'my browser key', ← 鍵
   remote: true, ← リモートサーバの場合 true
   decrypt: true}) ← 復号する場合 true
.then( (result) => console.log(result) );

～中略～

data:  "hello world!"
```

もちろん、ブラウザ (WebCrypto) とコマンドラインで相互接続可能。

- ブラウザで暗号化・登録したデータを、コマンドラインで取得・復号
- コマンドラインで暗号化・登録したデータを、ブラウザで取得・復号

今まで登録したデータで試してみよう！

暗号化・復号の中身

今回の事例でのモード設定

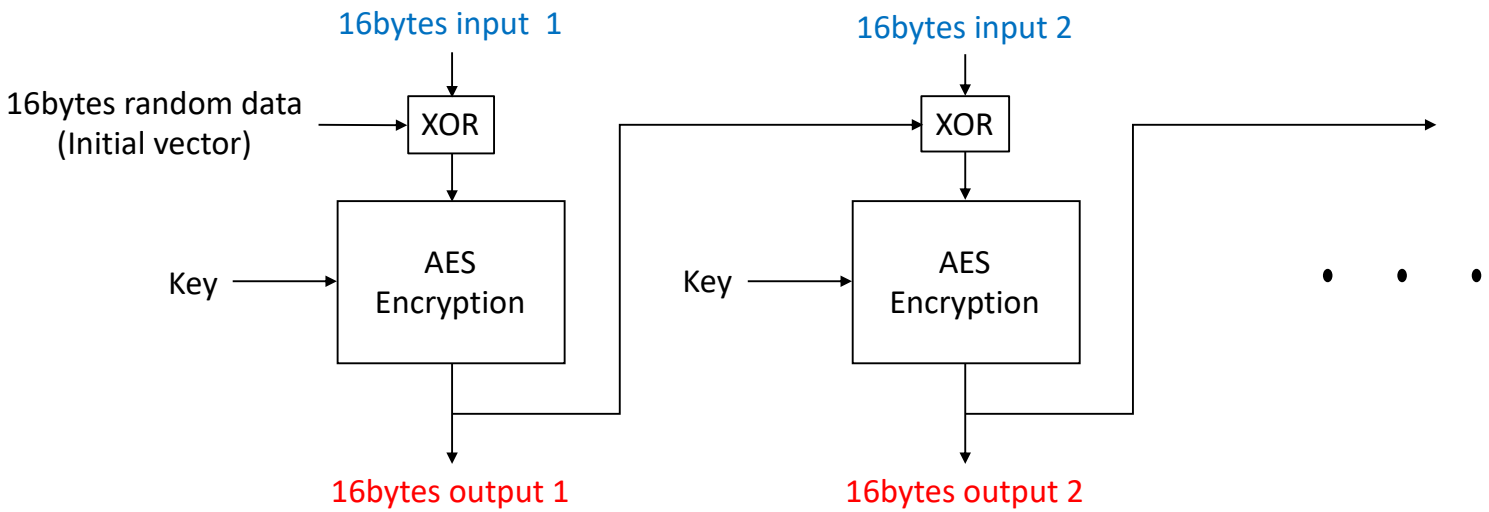
AES の「利用モード」

AES の処理 1 回で暗号化できるのはたった 16bytes にすぎない。長いデータを連続で暗号化するために、暗号化処理を連続して組み合わせる方法が利用モード。

安全性向上のため、一般的に

- 「先頭の 16 バイトはランダムな初期化ベクトルと混ぜる」
- 「前の 16 バイトのデータを継承して次の 16 バイトを処理」

などの工夫を入れた利用モードを用いる。



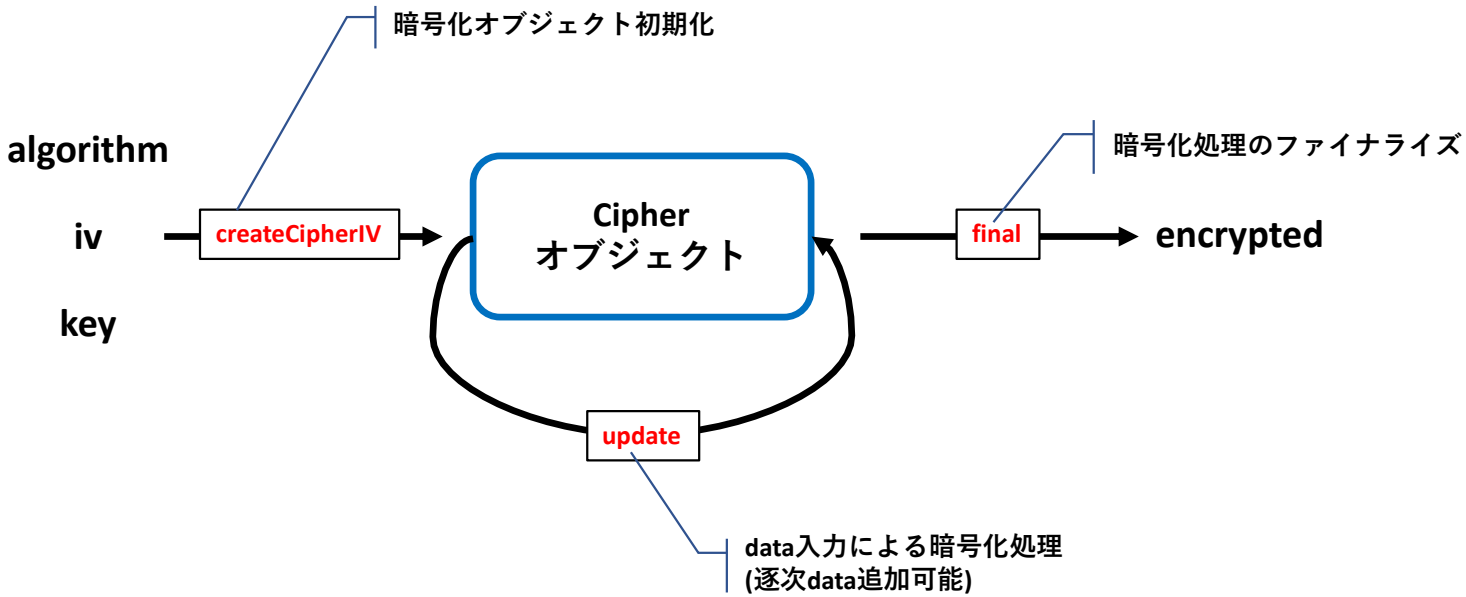
今回の事例では、256 ビット鍵 AES の CBC モード [18] を用いた。

[18] ISO 等で標準化されている一般的に安全なモード

Node.js Crypto ライブラリでの AES-CBC 暗号化・復号

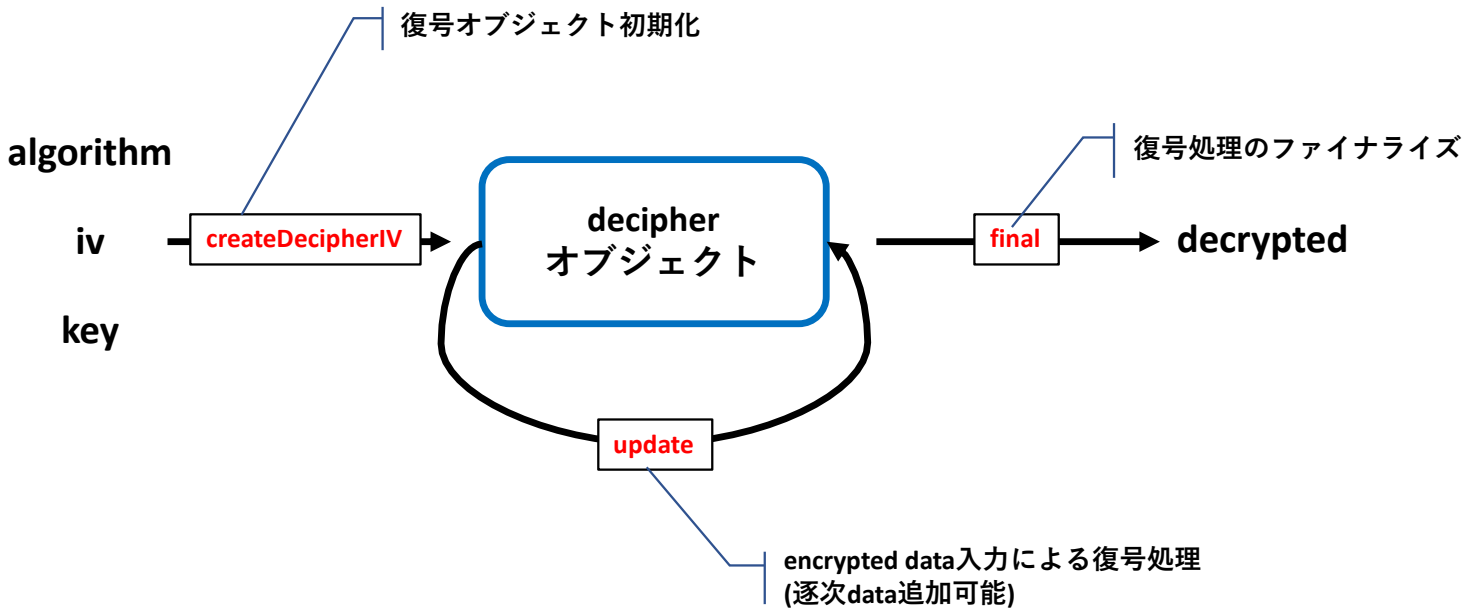
sample/src/encrypt-node.js: encrypt 関数

```
const crypto = require('crypto'); // crypto モジュールの読み出し
const algorithm = 'aes-256-cbc'; // AES-256-CBC モード

const iv = crypto.randomBytes(16); // Initial Vector の生成

const cipher = crypto.createCipheriv(algorithm, key, iv); // 暗号化 Object 初期化
let encrypted = cipher.update(uint8data, 'utf8', 'base64'); // 暗号化
encrypted += cipher.final('base64'); // 暗号化完了
```



sample/src/encrypt-node.js: decrypt 関数

```
const crypto = require('crypto'); // crypto モジュールの読み出し
const algorithm = 'aes-256-cbc'; // AES-256-CBC モード

const decipher = crypto.createDecipheriv(algorithm, key, iv); // 復号 Object 初期化
let decrypted = decipher.update(encrypted, 'base64', 'utf8'); // 復号
decrypted += decipher.final(); // 復号完了
```



ブラウザ WebCrypto API での AES-CBC 暗号化・復号

sample/src/encrypt-browser.js: encrypt 関数

```
const crypto = window.crypto; // WebCrypto API
const iv = crypto.getRandomValues(new Uint8Array(16)); // IV の生成

const importedKey = await crypto.subtle.importKey( // key オブジェクトの生成
  'raw', // バイナリ鍵のインポート指定
  key,
  { name: 'AES-CBC' }, // CBC モードに利用
  false, // key オブジェクトからの生鍵エクスポート可否
  ['encrypt', 'decrypt'] // 暗号化・復号に利用
);

const encrypted = await crypto.subtle.encrypt( // 暗号化を実行
  { name: 'AES-CBC', iv: uint8iv }, // CBC モードで暗号化
  importedKey,
  data
);
```

sample/src/encrypt-browser.js: decrypt 関数

```
const crypto = window.crypto; // WebCrypto API

const importedKey = await crypto.subtle.importKey( // key オブジェクトの生成
  'raw', // バイナリ鍵のインポート指定
  key,
  { name: 'AES-CBC' }, // CBC モードに利用
  false, // key オブジェクトからの生鍵エクスポート可否
  ['encrypt', 'decrypt'] // 暗号化・復号に利用
);

const decrypted = await crypto.subtle.decrypt( // 暗号化を実行
  { name: 'AES-CBC', iv: uint8iv }, // CBC モードで暗号化
  importedKey,
  encrypted
);
```

そのまま使うのはちょっと大変…

- Node.js とは全く違う API
- 鍵インポート API と暗号化・復号 API でモード指定が重複するなど、不可解な仕様になっている…
- ブラウザごとの実装差異が大きい (利用可能オプションなど)

補足: API が違うのがめんどくさい…

統合 **API** を提供するライブラリを使って楽をすると良い **(手前味噌)**

js-crypto-utils (jscu)

GitHub: `https://github.com/junkurihara/jscu`
NPM: `https://www.npmjs.com/package/js-crypto-utils`

- Node.js/ブラウザで全く同じ API を提供。→ コード再利用性向上
- ブラウザごとの実装差異を (ある程度) 吸収。
- MIT ライセンス。好きに使って！

jscu を使うと Node.js もブラウザも次のような短いコードになる。

sample/src/encrypt-universal.js: encrypt 関数

```
const uint8iv = jscu.random.getRandomBytes(16); // IV の生成

const encrypted = await jscu.aes.encrypt( // AES 暗号化
  uint8data,
  key, // バイナリ鍵
  {name: 'AES-CBC', iv: uint8iv} // CBC モードの指定
);
```

sample/src/encrypt-universal.js: decrypt 関数

```
const decrypted = await jscu.aes.decrypt( // AES 復号
  encrypted,
  key, // バイナリ鍵
  {name: 'AES-CBC', iv: uint8iv} // CBC モードの指定
);
```

まとめ

まとめ

お疲れ様でした。

- E2E セキュリティの概念・重要性の紹介
- Web における E2E セキュリティを実現するため、JS での AES 実行方法の紹介

次回以降…リクエスト次第ですが、

- もう少し詳しく：AES の利用方法について
- 公開鍵暗号と Hybrid 暗号化
- ハッシュ・署名と HMAC
- RFC とアルゴリズム・フォーマット

などを予定。

宣伝 1

E2E 暗号化ファイル転送サービス「iTransfy」を提供しています。

宣伝 2

Zettant ではイケイケの仲間を募集しています。

1. 今回は共通鍵暗号
2. 公開鍵暗号& Hybrid Encryption
3. ハッシュ・署名と HMAC
4. 超マニアック講座：RFC とアルゴリズム・フォーマット

Appendix

This page is not counted.