Cisco Spark, Tropo REST, Pythonの概要

大園　通 / OHZONO TOHRU
Sr. Systems Engineer

DevNet Express [Tokyo]
March 15, 2017 • Tokyo, Japan

cisco DevNet
developer.cisco.com

Cisco Spark, Tropo REST, Pythonの概要

大園　通 / OHZONO TOHRU
Sr. Systems Engineer

本来は同列で並べるような
内容ではないが、
ゆるい関連性はあるので、
順を追って説明します

はじめに

おことわり

本プレゼンでは、わかりやすさ重視で、
大胆に噛み砕いた説明を行います。
厳密さは多少犠牲にしているため、ご了承ください。

LEARN

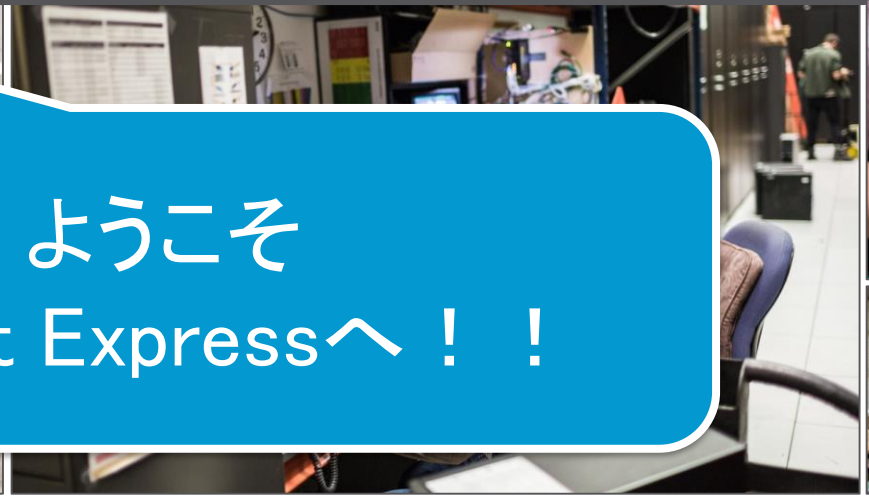
CODE

INSPIRE









LEARN CODE INSPIRE

ようこそ
DevNet Expressへ！！


学んで
刺激を受けて
次につなげる
LEARN
CODE
INSPIRE
コードを
書いて

今日の過ごし方

楽しみ方は人それぞれですが・・・

内容的には、
プログラミングに
慣れていない方には
そこそこ難易度の高い
内容です

「動いた」を楽しむ

「動かない」を楽しむ

試行錯誤しながら、
大いに遊んでみましょう！

日本人的な読み方の例

読み方は人それぞれですが・・・

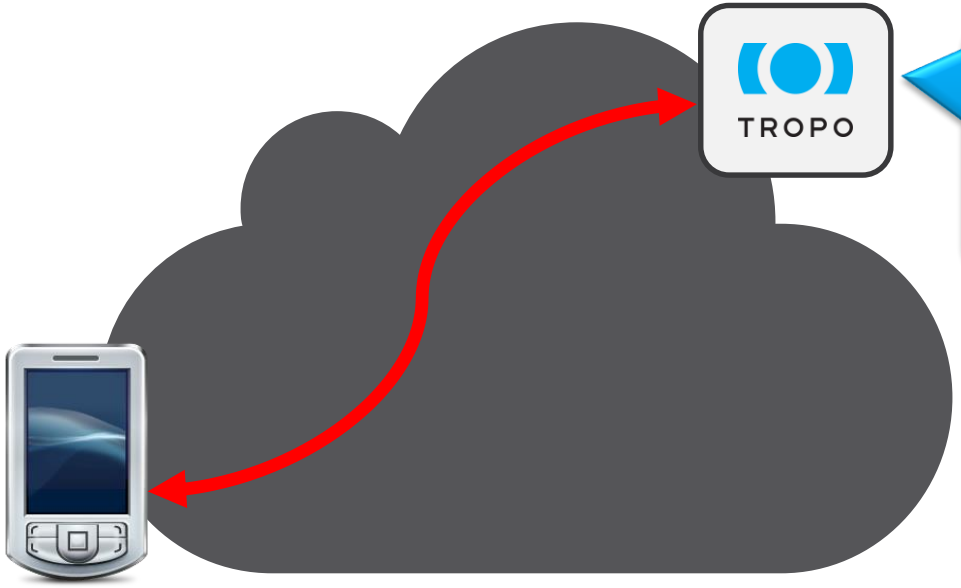
REST	レスト
JSON	ジェイソン
Python	パイソン
Tropo	トローポ

Git	ギット
GitHub	ギットハブ

「ジット」、「ジットハブ」と
読む人もいるので
文脈で判断

Tropo

Tropo



公衆電話網を利用した
アプリを開発したり
実行したりできる
クラウド上のプラットフォーム

https://www.tropo.com/

Tropo – 主な機能の例

多くの国の電話網と連携可能

電話会議

SMSの送受信

Text to Speech

音声の録音

音声認識

Tropo – 主な機能の例

多

Tropoのクラウドプラットフォーム上で
スクリプト実行することで
Tropoの提供する様々な
機能にアクセスできます

電話会議

```
say(”ようこそTropoの世界へ”)
conference("1337")
```

Speech

認識

Tropo – 主な機能の例

多

Tropoのクラウドプラットフォーム上で
スクリプト実行することで
Tropoの提供する様々な
機能にアクセスできます

電話会議

```
say(”ようこそTropoの世界へ”)
conference("1337")
```

スクリプトによって、
動作や機能をガンガン
実装していきます！
開発者ありきな
プラットフォームといえる

認識

Tropo

本日午後のハンズオンでは、
Tropoのハンズオンは実施対象外になります。
ハンズオンの資料自体はあります。

Cisco Spark

Cisco Spark
コミュニケーションをよりシンプルに

音声 ビデオ	会議	ファイルの 共有	1:1またはチーム でのテキスト メッセージ	モバイル デスクトップ アプリ	デスクトップ 会議室の ビデオ端末

Cisco Sparkプラットフォーム

統合されたユーザ体験

Message

Meeting

Call

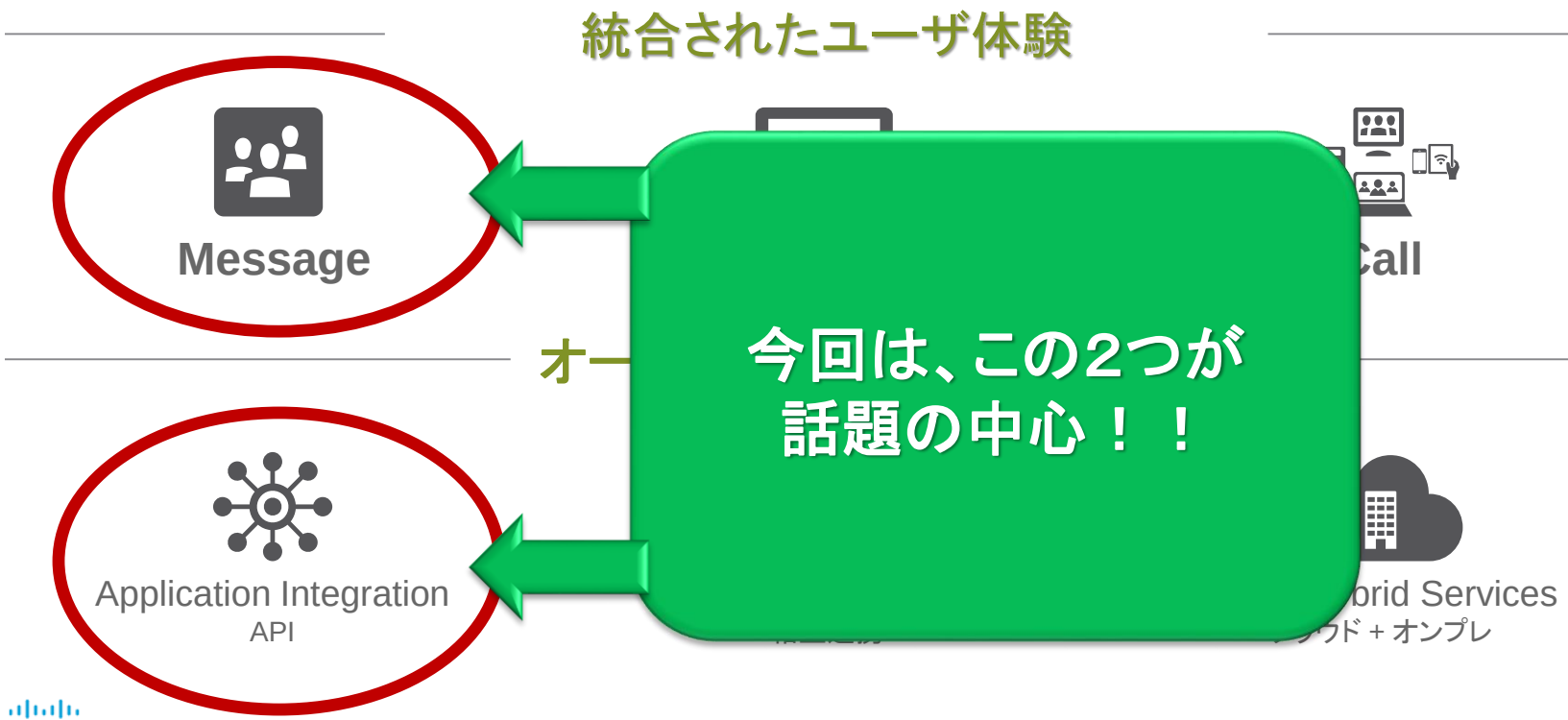
オープンなプラットフォーム

Application Integration
API

Partner Services
相互連携

Spark Hybrid Services
クラウド + オンプレ

Cisco Sparkプラットフォーム



Cisco Spark - Message

Message

Demo

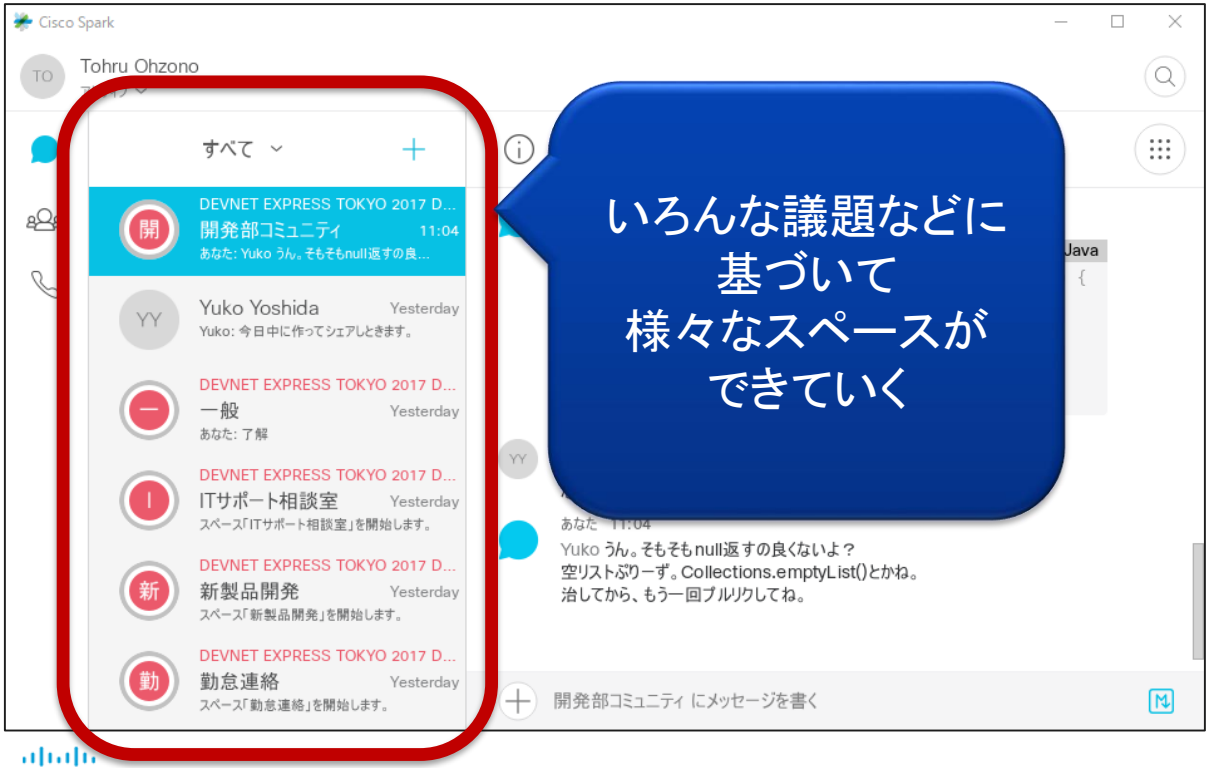
実際に見て、
触ってみるのが、
一番分かりやすい

Cisco Spark - Message

Cisco Spark - Message

スペースとよばれる
仮想的な場所で
テキストやビデオを
使って
社内外の人達と
連携する

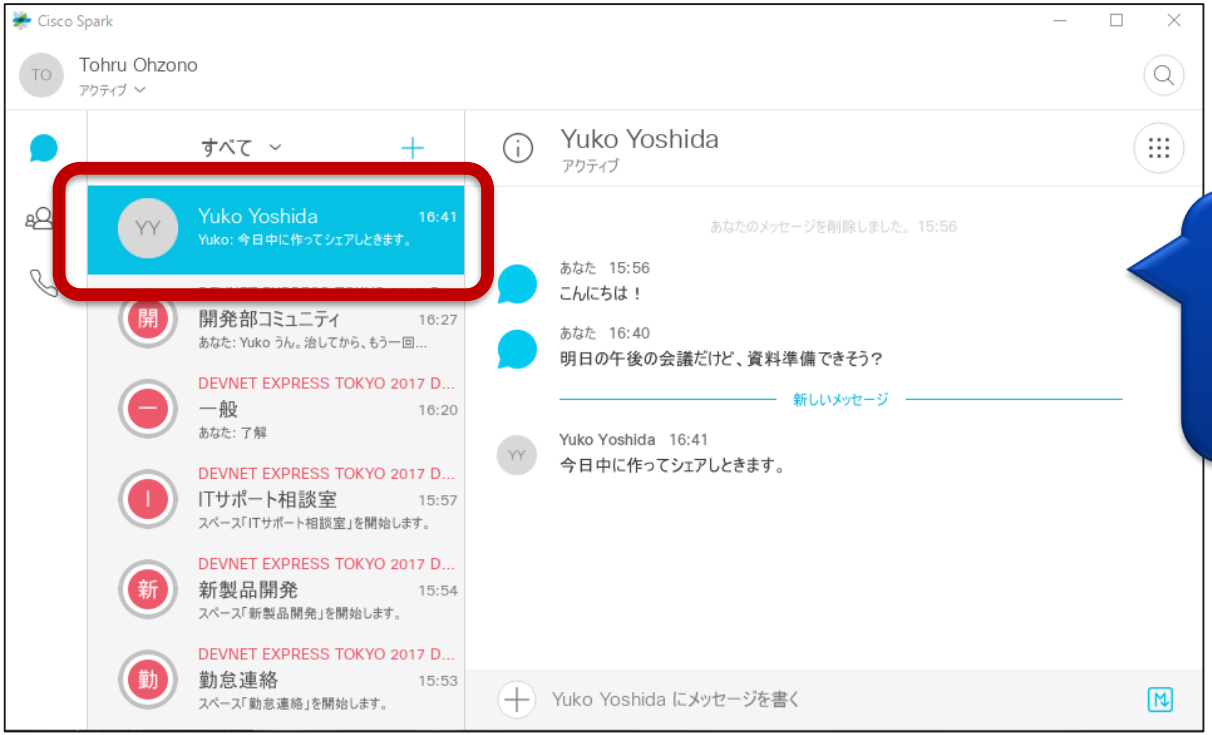
Cisco Spark - Message



Cisco Spark - Message

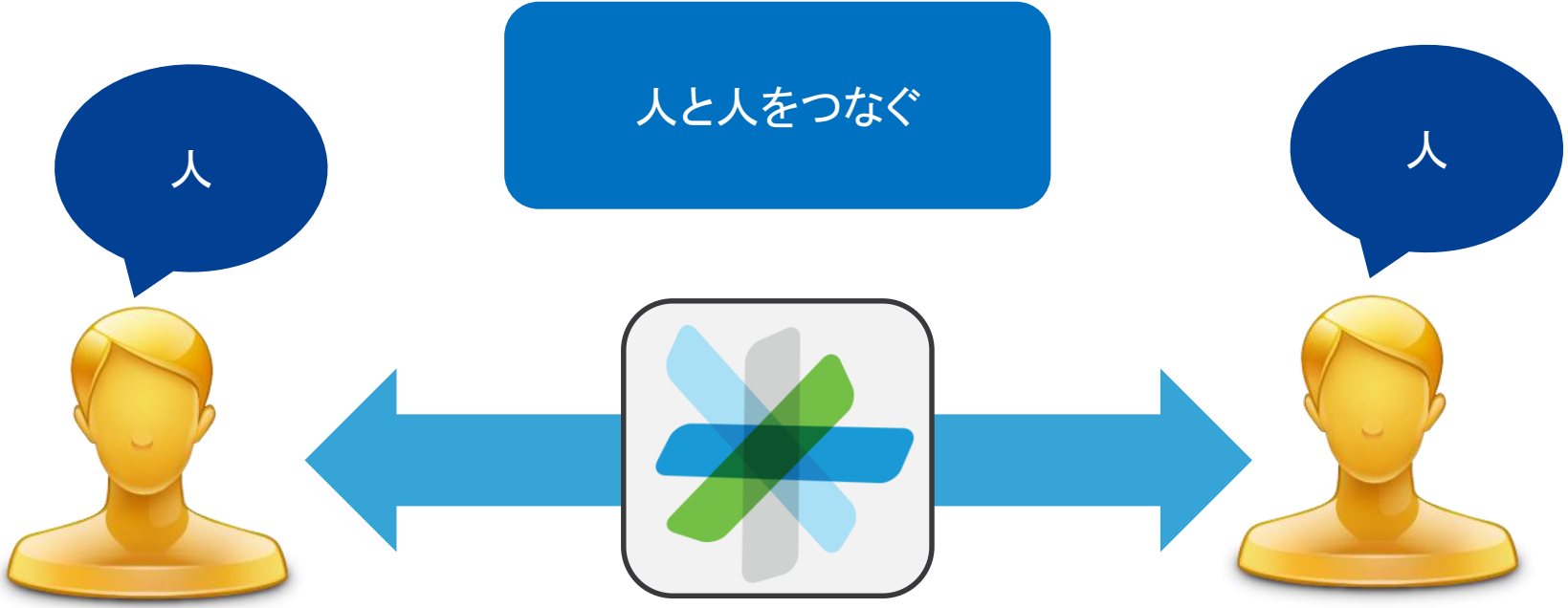
大勢いるスペースでも
@mention(@メンション)機能を使って、
スペース内の特定の人に
話しかけることもできる。
この例では、”Yuko”に対して
スペース内で話しかけている。
(投稿内容はスペース内の全員に見える)

Cisco Spark - Message



1対1でのやりとりもできる

Sparkは何をつなぐか？
- いくつかの例 -



Sparkは何をつなぐか？
- いくつかの例 -

Sparkは何をつなぐか？
- いくつかの例 -

人(人々)とアプリをつなぐ

人々

アプリ

Sparkは何をつなぐか？
- いくつかの例 -

さらにその先には、
別のアプリ、
監視カメラ、機械、乗り物
などなど

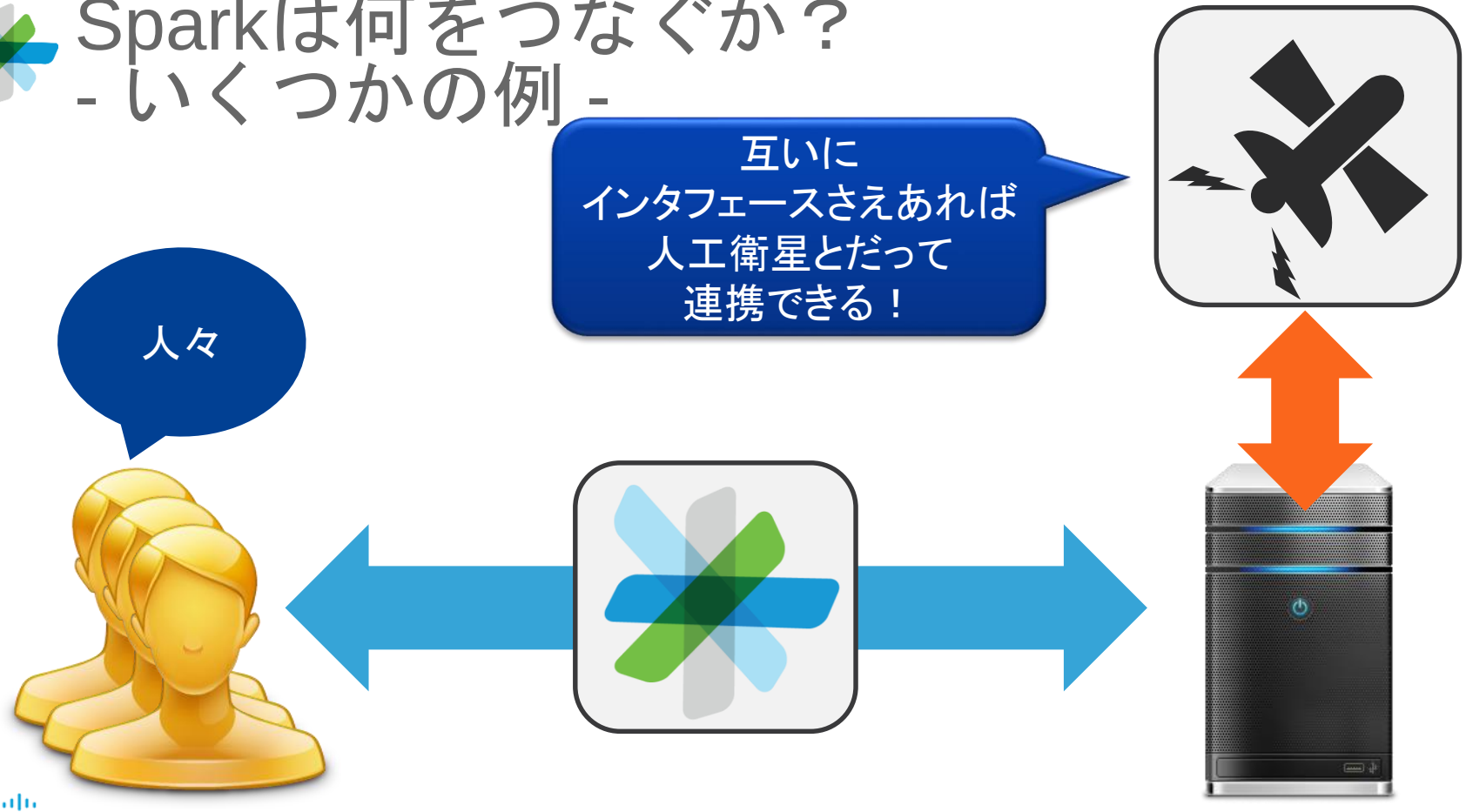
人々

Sparkは何をつなぐか？
- いくつかの例 -

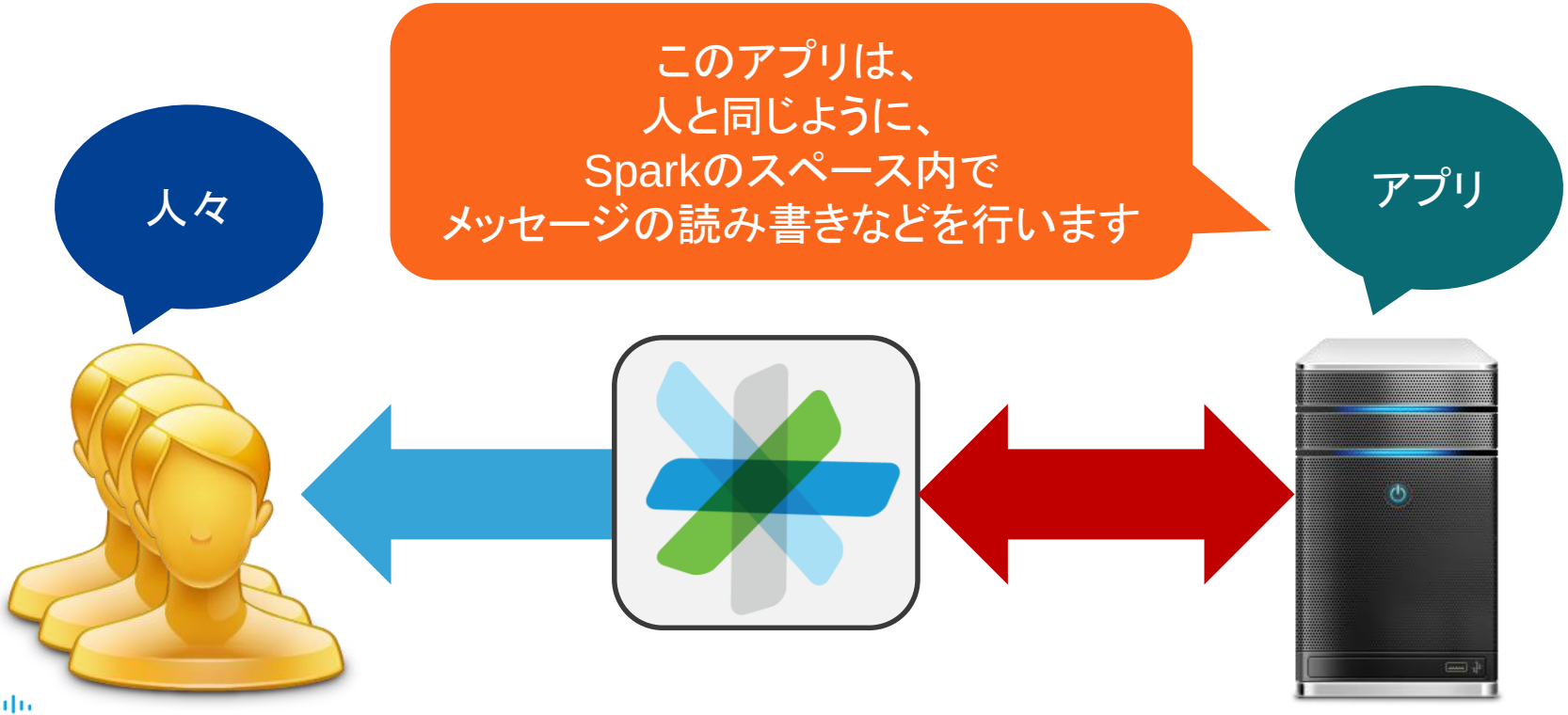
インタフェースが
あるもの同士なら
可能性は無限大！！

人々

Sparkは何をつなぐか？
- いくつかの例 -



Sparkは何をつなぐか？
- いくつかの例 -

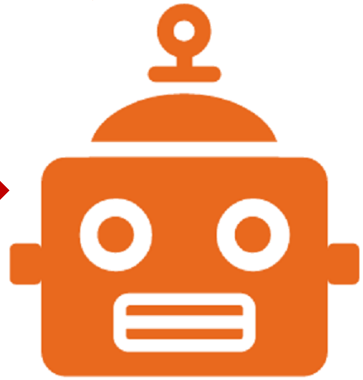


Sparkは何をつなぐか？
- いくつかの例 -

人になり代わって、
その先のアプリ、システム等と
連携して作業するアプリは
Bot(ボット)と呼ばれる

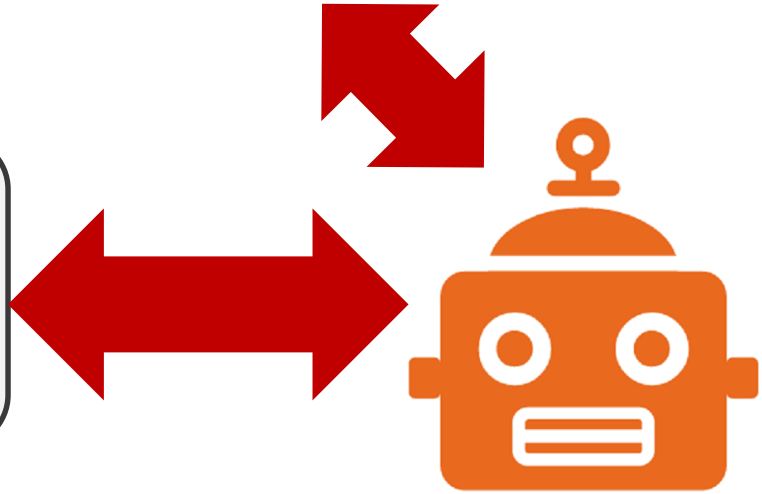
人々

Bot



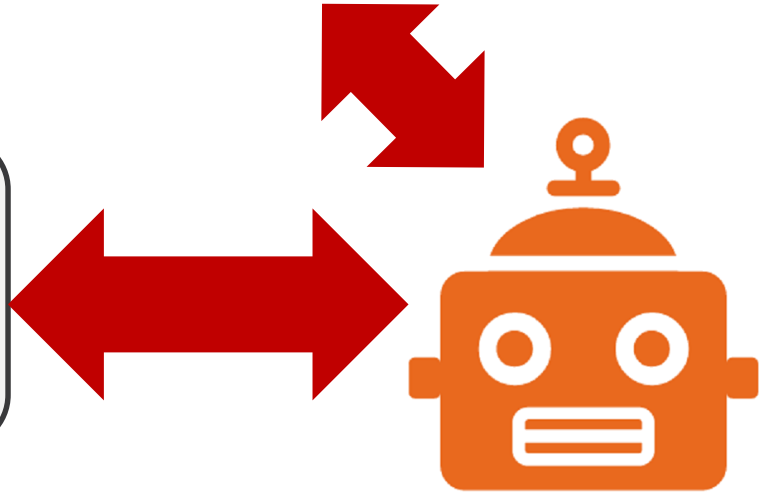
Sparkは何をつなぐか？
- いくつかの例 -

Botを介して
Sparkと様々な
ものがつながる
ことができる！！



Sparkは何をつなぐか？
- いくつかの例 -

Botを介して
Sparkと様々な
ものがつながる
ことができる！！



Sparkは何をつなぐか？
- いくつかの例 -

Sparkのスペース内で
これらが繋がりあうことで
新しい価値を生み出せる
可能性がある

人(人々) ⟷ 人(人々)

人(人々) → Bot

Bot → 人(人々)

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

空港職員同士でのやりとり

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

お客様から、
“CSCO11便”の
運行情報を
質問された場合

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

同じスペースにいる
運行情報システムと連携するBotに
話しかけて、運行情報ゲット。
お客様に情報を伝える

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

お客様から、
忘れ物の
相談をされた場合

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

同じスペースにいる
紛失物検索システムと連携するBotに
話しかけて、登録情報を確認。
お客様をご案内

人(人々)、Bot、その先のシステムがつながり合う例

空港職員コミュニティスペースのコンセプトデモ

お客様ご案内ロボット
がお困り中

人(人々)、Bot、その先のシステムがつながり合う例

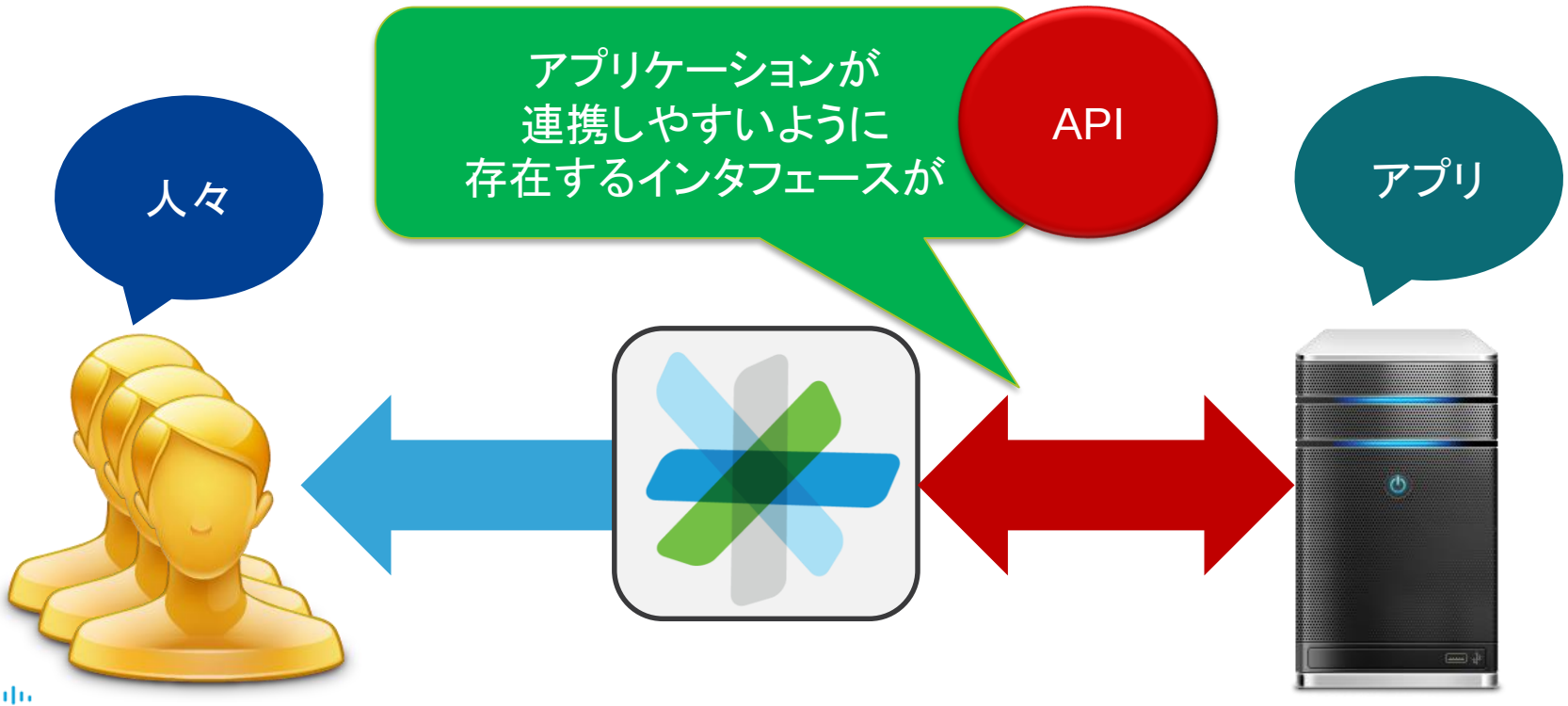
空港職員コミュニティスペースのコンセプトデモ

人(人々)、Bot、その先のシステムがつながり合う例

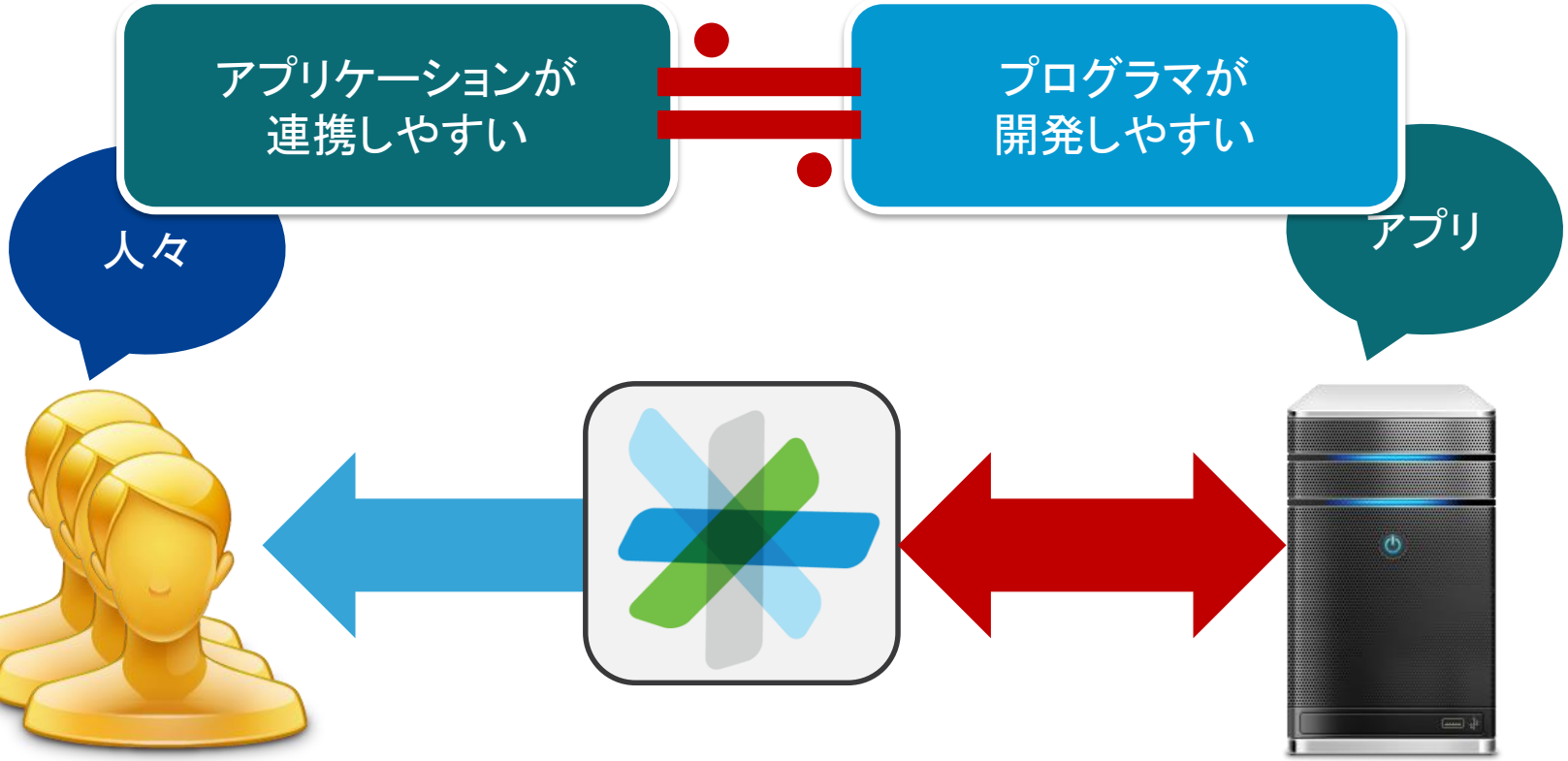
空港職員コミュニティスペースのコンセプトデモ

ロボットと連動したBot
が人に助けを請う

Spark API



Spark API



Spark API

Spark Message APIの機能の例

Spark上のメッセージ

- 投稿
- リスト取得
- 詳細取得
- 削除

Spark上のスペース

- 作成
- 更新
- リスト取得
- 詳細取得
- 削除

その他、メッセージが投稿された際などにイベントを受け取ったりも可能

Spark API

朗報！！

SparkのAPIを使って
アプリ開発を始めるのに
追加のコストは
必要ありません！

Sparkのユーザに
なった瞬間に、
開発者にもなれます！

Sparkの無償版の
機能に対応するAPIは、
やはり無償で使えます！

Spark For Developers

Cisco Sparkは、 REST API

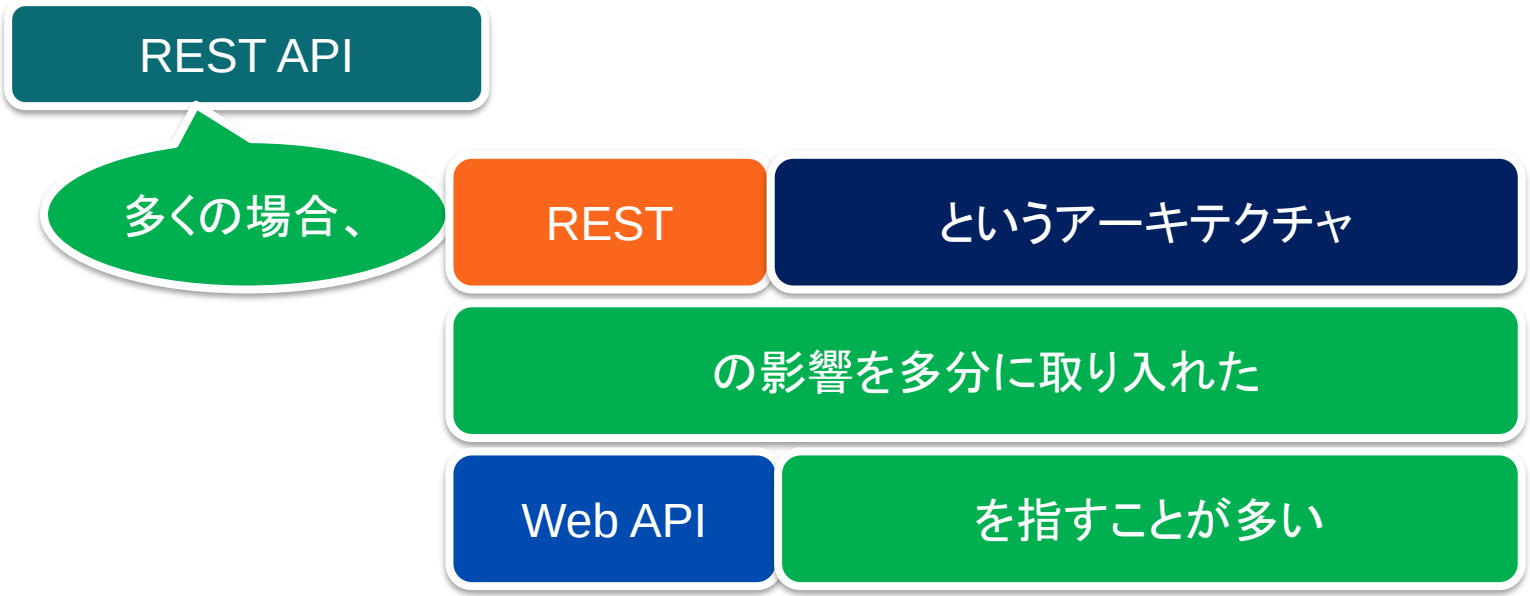
を提供しています

`developer.ciscospark.com`

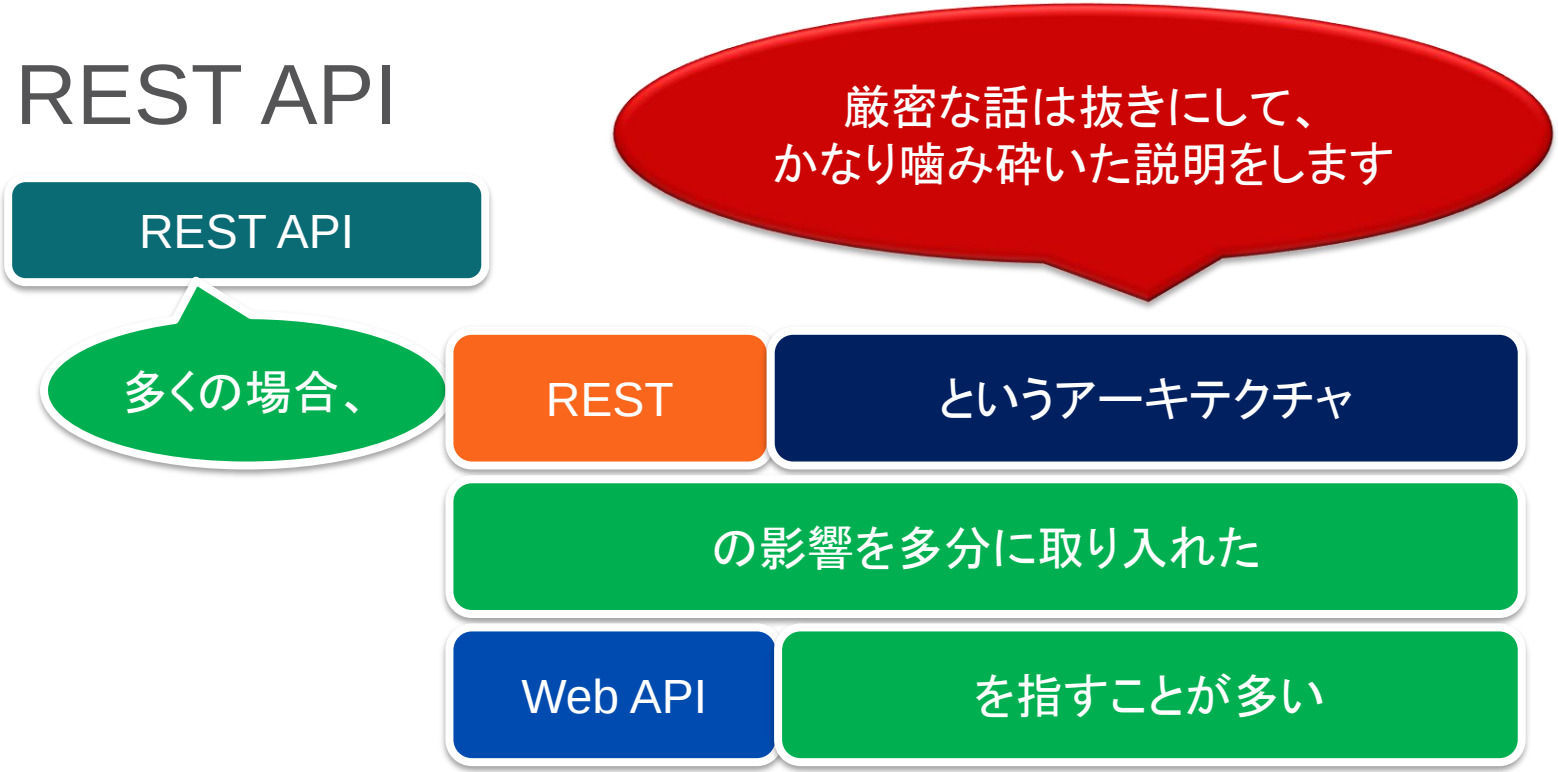
Application Integration
APIs

REST API

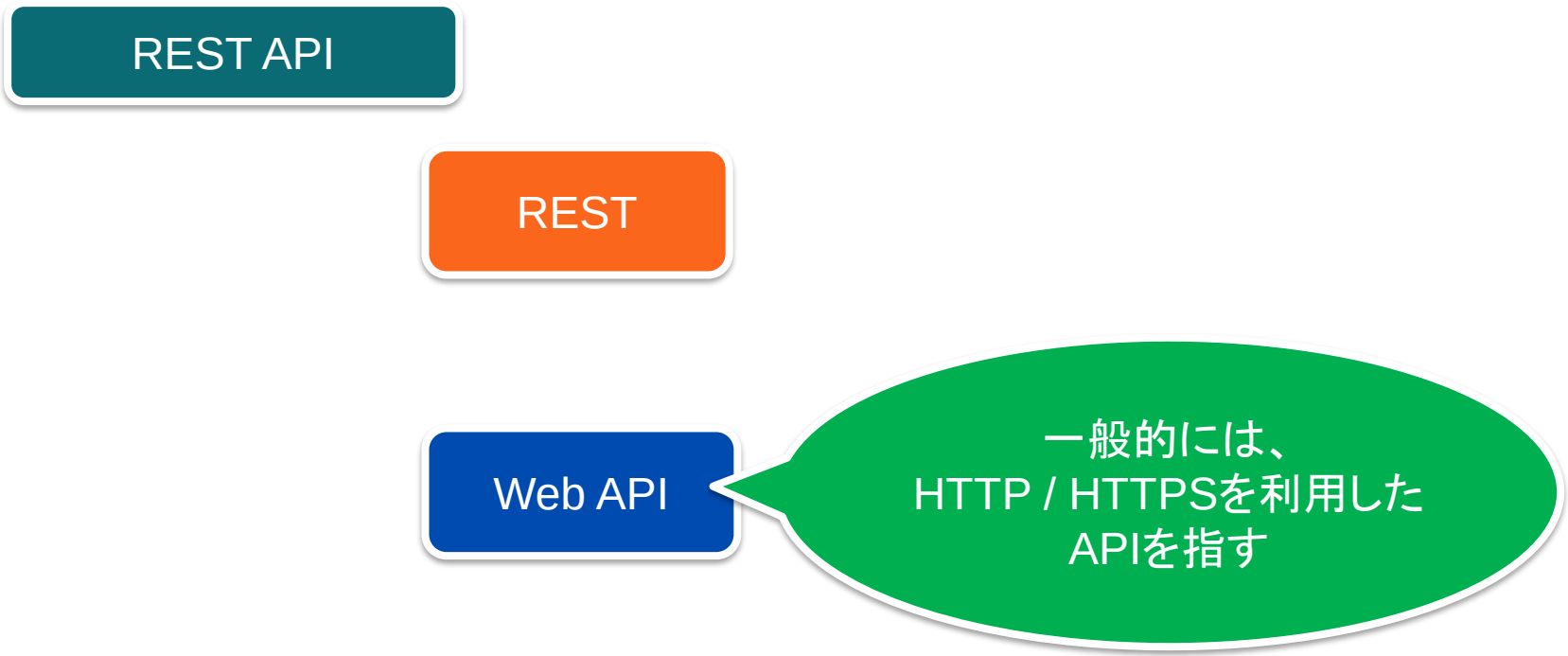
REST API



REST API



REST API



HTTP

HTTP

まずは、HTTPの説明

本日は、
HTTP/1.1と呼ばれるバージョンの話です。
HTTP/2では、事情がまた異なります。

HTTP

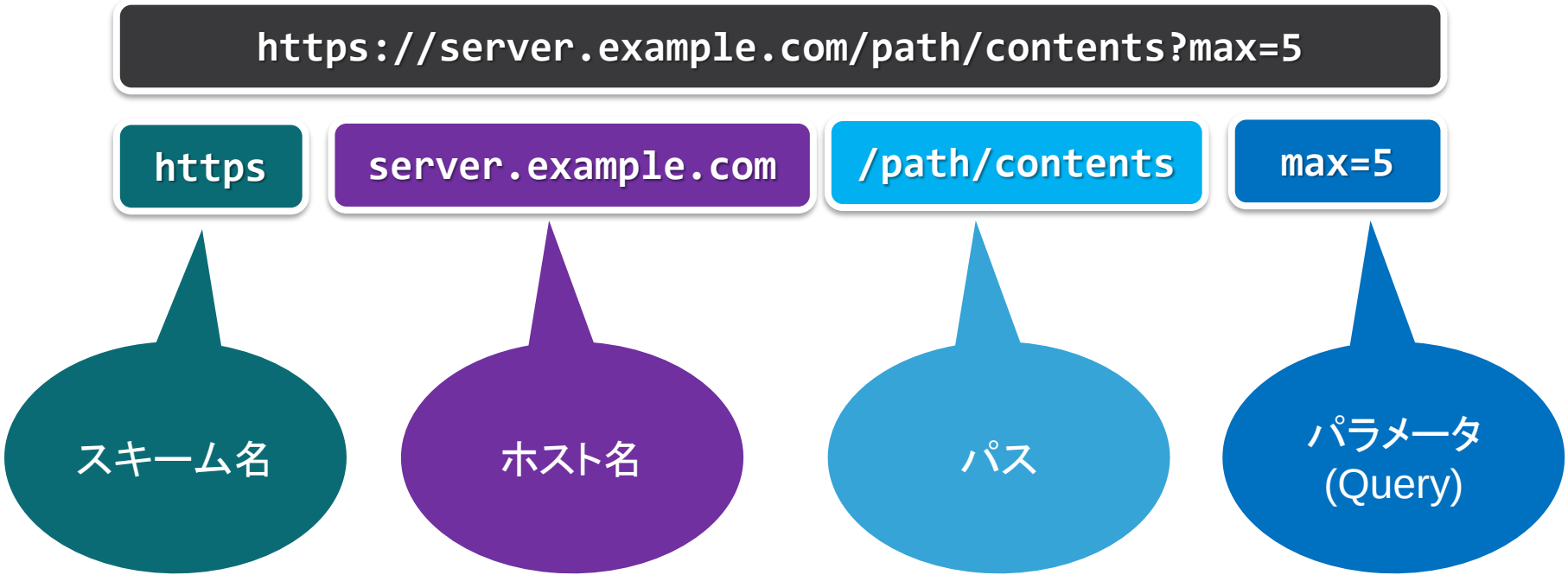
```
https://server.example.com/path/contents?max=5
```

ネットワーク上のリソース(情報資源)の場所を表す
URLの例です

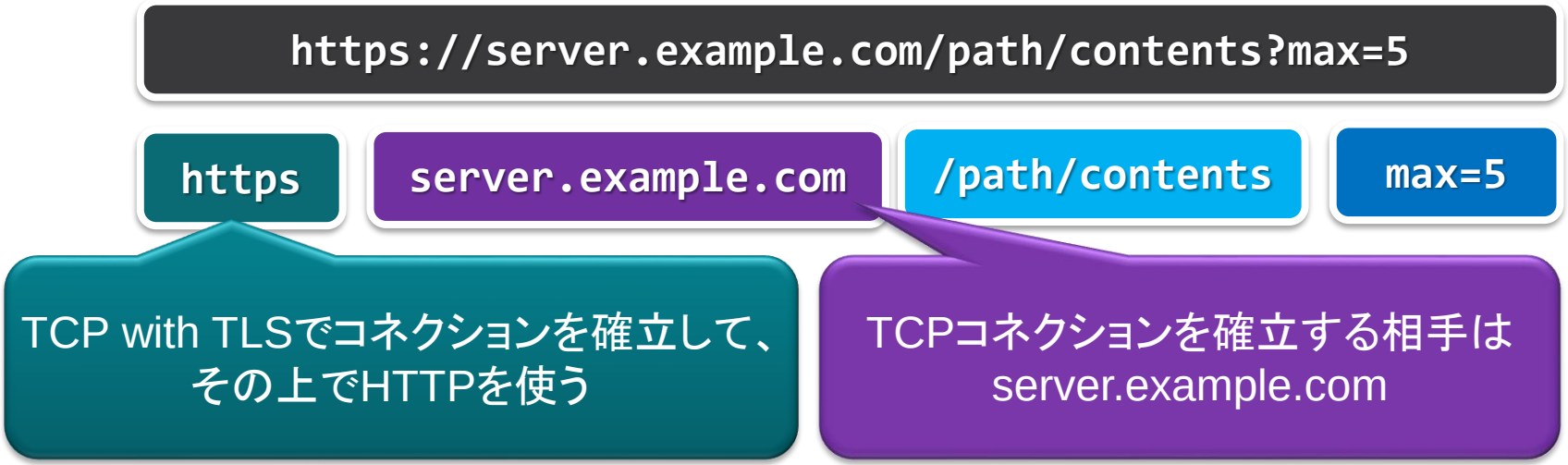
このURLが指し示す場所から
リソースを取得するアプリを
考えてみよう

HTTP

HTTP



HTTP



HTTP

https://server.example.com/path/contents?max=5

https | server.example.com | /path/contents | max=5

TCP with TLSでコネクションが確立されて
通信が可能な状態になる

HTTP

https://server.example.com/path/contents?max=5

https | server.example.com | /path/contents | max=5

このパス(場所)のリソースが欲しい

結果を絞りたい

アプリ

server.example.com

HTTP

https://server.example.com/path/contents?max=5

th/contents

max=5

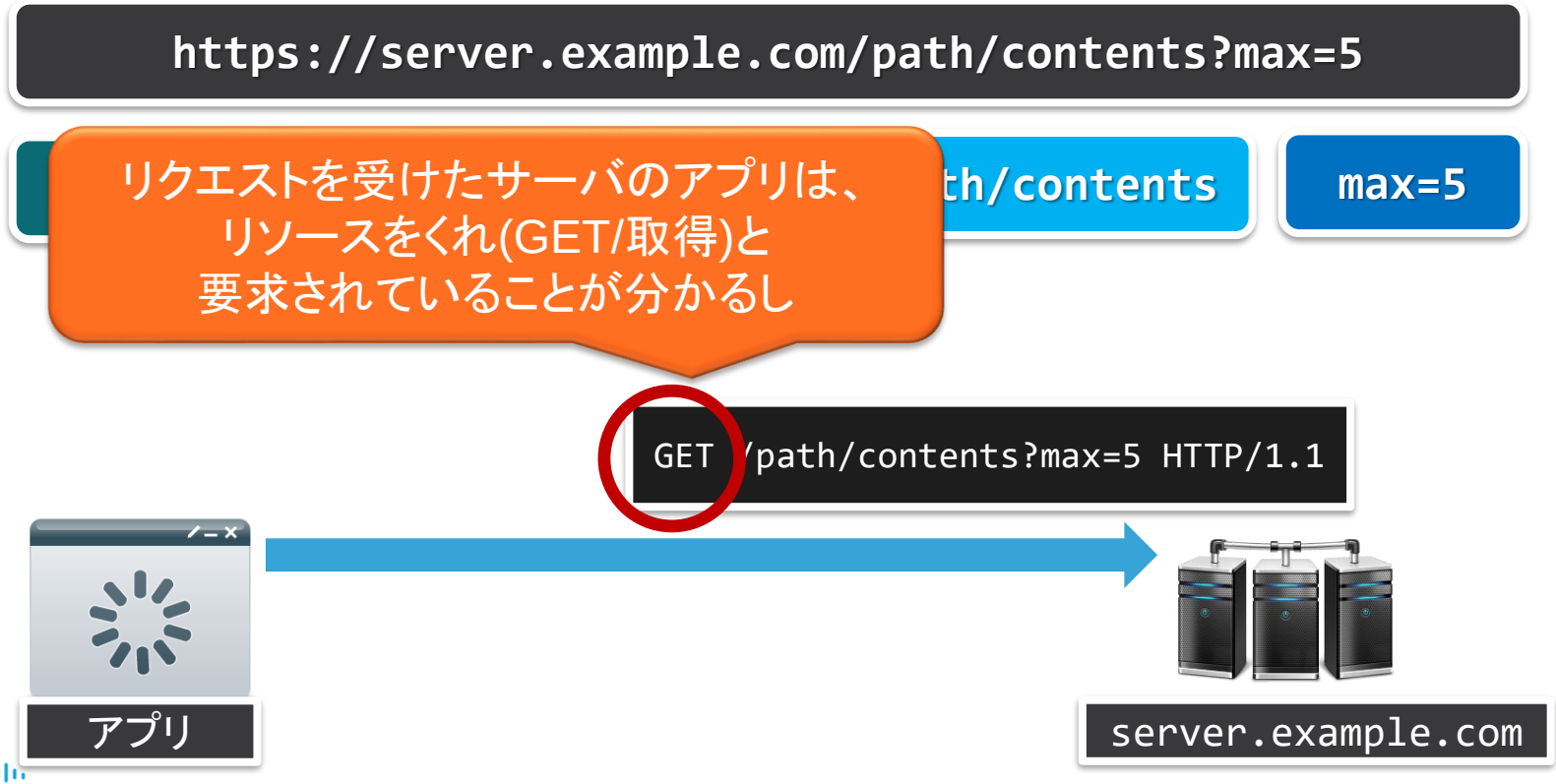
このようなテキストのリクエストを
確立されたTCPコネクション上で送る

```
GET /path/contents?max=5 HTTP/1.1
```

アプリ

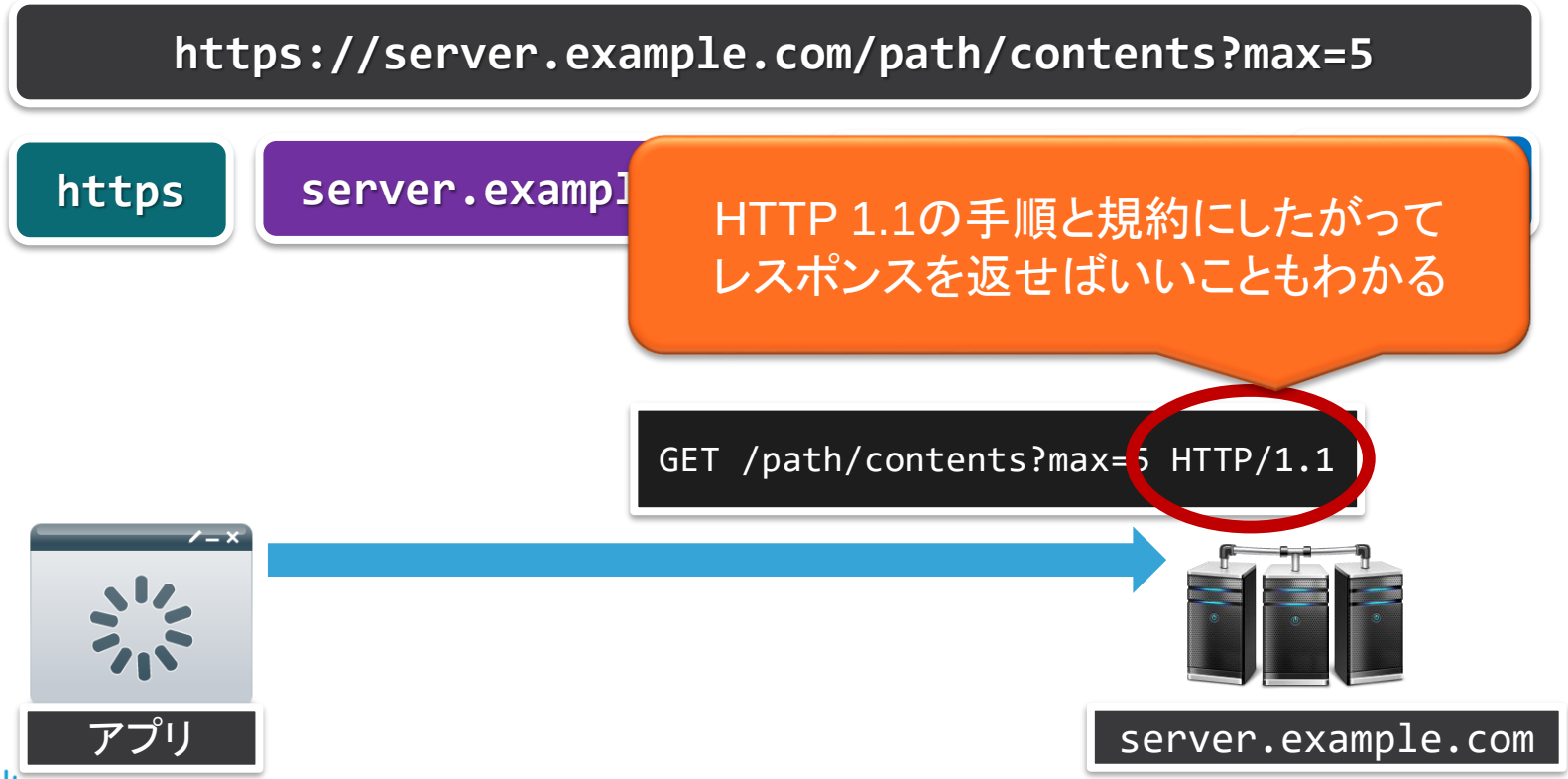
server.example.com

HTTP



HTTP

HTTP



HTTP

https://server.example.com/path/contents?max=5

https

server.example.com

要求された通りの
リソースを探したりすることで
レスポンスで返すことができる

GET /path/contents?max=5 HTTP/1.1

アプリ

server.example.com

HTTP

https://server.example.com/path/contents?max=5

https | serv | /path/contents | max=5

リクエストされたリソースを
このようなテキストのレスポンス
として返します

```
HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！
```

アプリ

server.example.com

HTTP

https://server.example.com/path/contents?max=5

htt | th/contents | max=5

HTTP 1.1の手順と規約にしたがった
レスポンスであることがわかる

HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！

アプリ

server.example.com

HTTP

https://server.example.com/path/contents?max=5

https | server.example.com | /path/contents | max=5

リソースが正常に
取得できていることもわかる

HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！

アプリ

server.example.com

HTTP

https://server.example.com/path/contents?max=5

https | server.example.com | /path/contents | max=5

リソースの中身が
取得できます

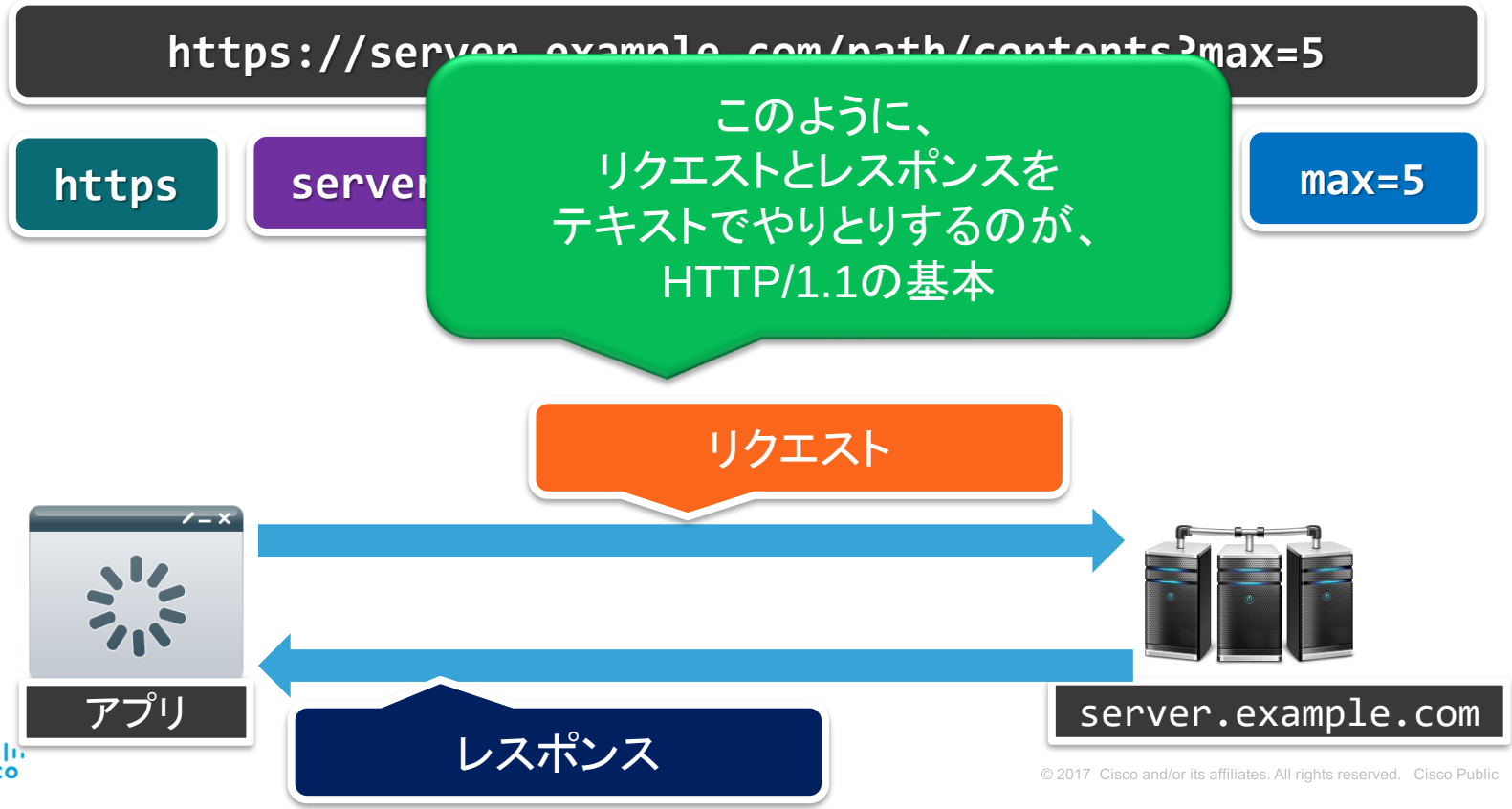
HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！

アプリ

server.example.com

HTTP



HTTP

実際には、
情報を受け取った側は、
これだけでは処理しづらいし、
拡張性もない

```
HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！
```

アプリ

server.example.com

HTTP

文字列の
エンコーディングは？

文字列情報は
何バイト
読み込めばいいの

文字列の情報しか
受け取れないの？

アプリ

```
HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！
```

server.example.com

HTTPヘッダ

リクエスト と レスポンス

それぞれに ヘッダ

を付加することで、相手の処理しやすさと
拡張性を確保することができる

```
HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！
```

アプリ

server.example.com

HTTPレスポンスヘッダ

レスポンスにヘッダを付けた例

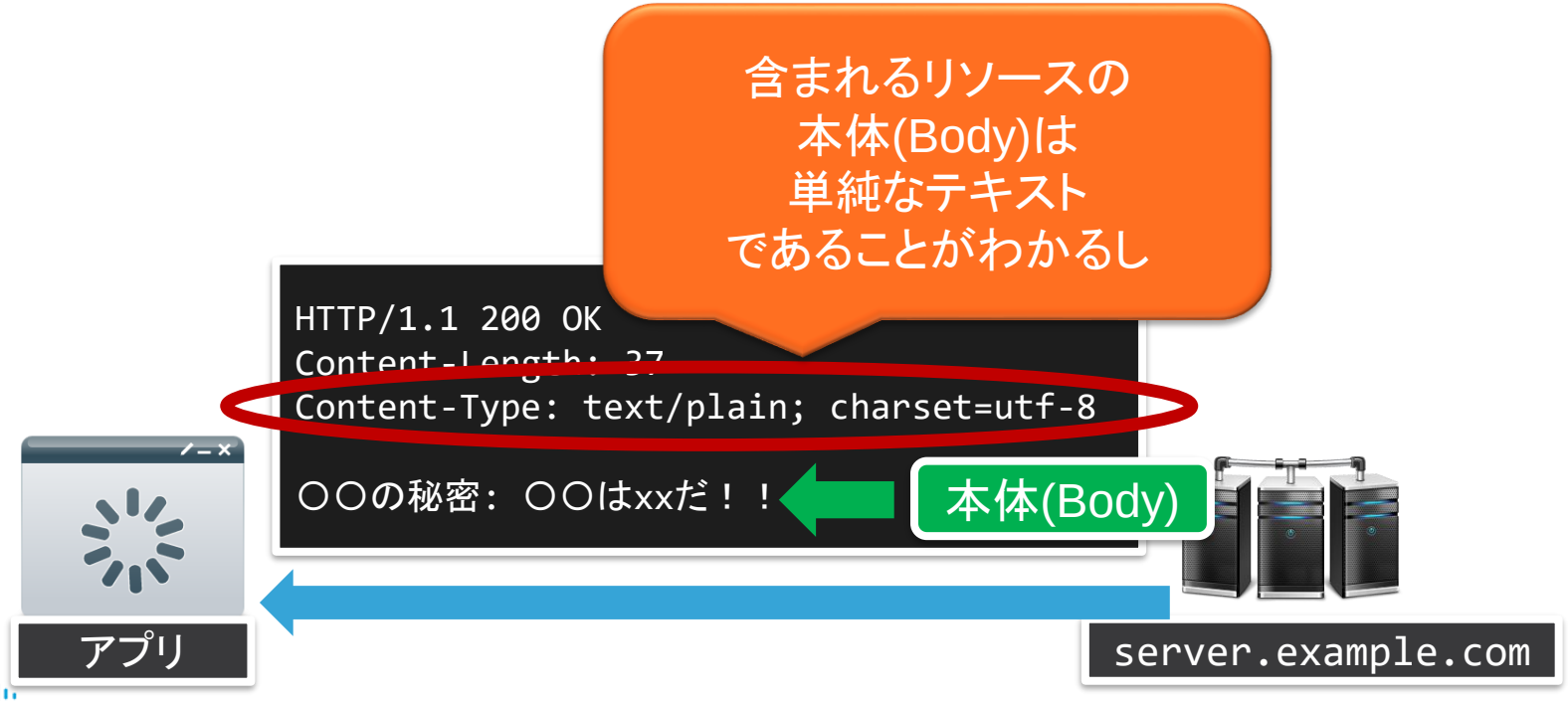
受信したアプリは、
このレスポンスを適切に処理
できるようになる

```
HTTP/1.1 200 OK
Content-Length: 37
Content-Type: text/plain; charset=utf-8

〇〇の秘密：〇〇はxxだ！！
```

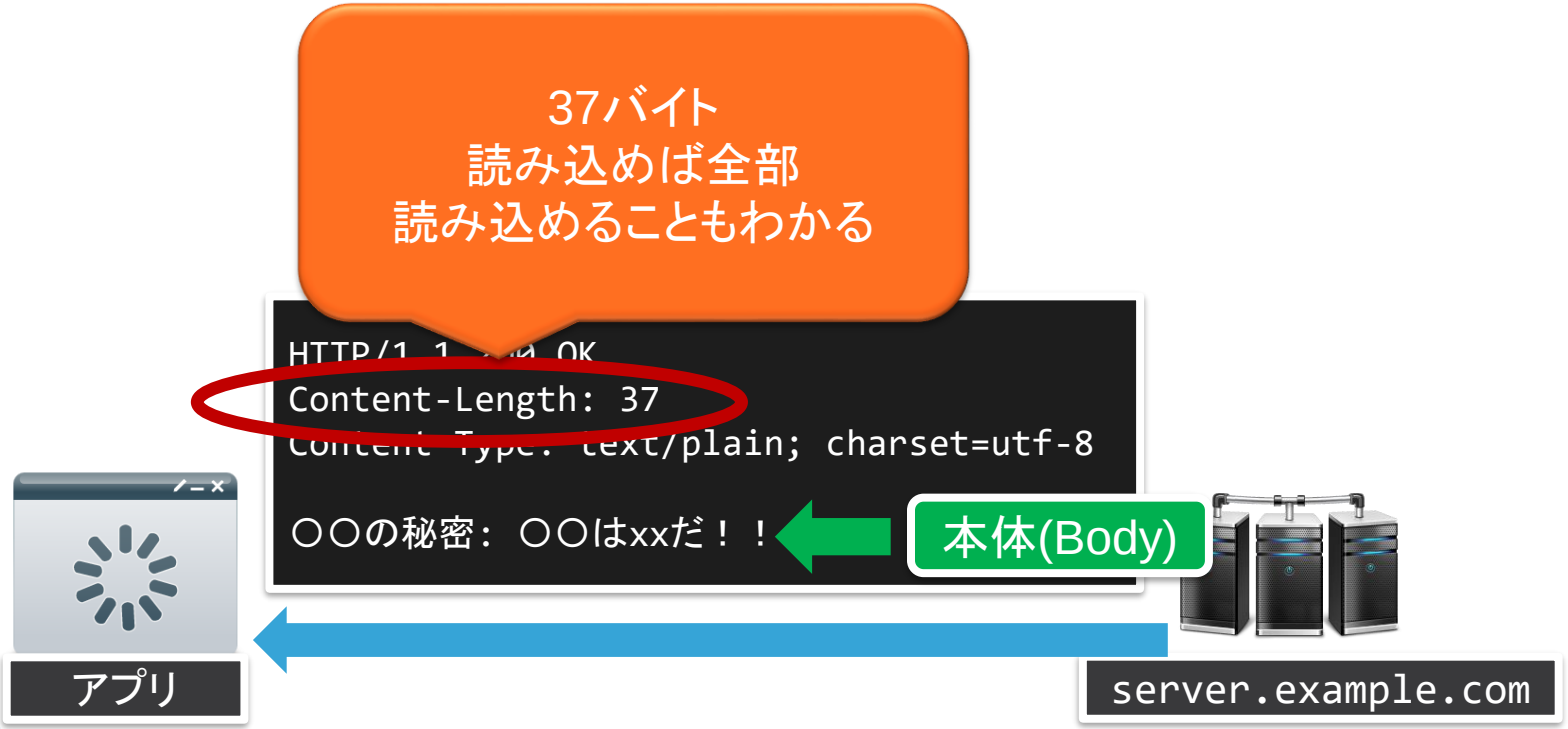
アプリ

server.example.com

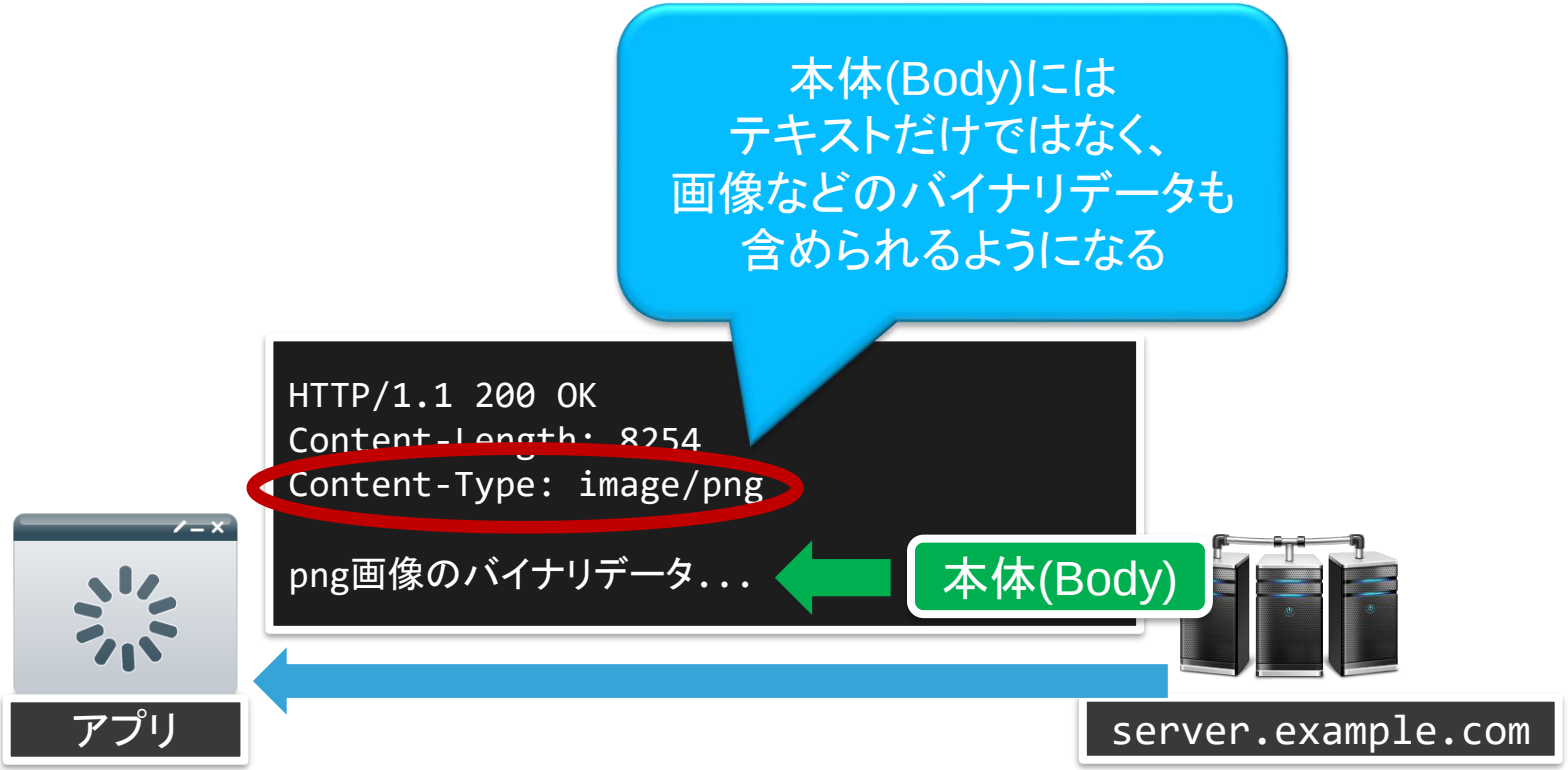
HTTPレスポンスヘッダ



HTTPレスポンスヘッダ



HTTPレスポンスヘッダ



HTTPリクエストヘッダ

```
GET /path/contents?max=5 HTTP/1.1
Host: server.example.com
Accept: text/plain
```

HTTPリクエストヘッダ

リクエストにヘッダを付けた例

リクエスト時も同様に
リクエストを受信した側で、
適切に処理できるようになる

```
GET /path/contents?max=5 HTTP/1.1
Host: server.example.com
Accept: text/plain
```

アプリ

リクエスト

server.example.com

HTTPリクエストヘッダ

サーバ側に
ユーザの認証・認可情報を渡して
認証や認可処理を実行することもある

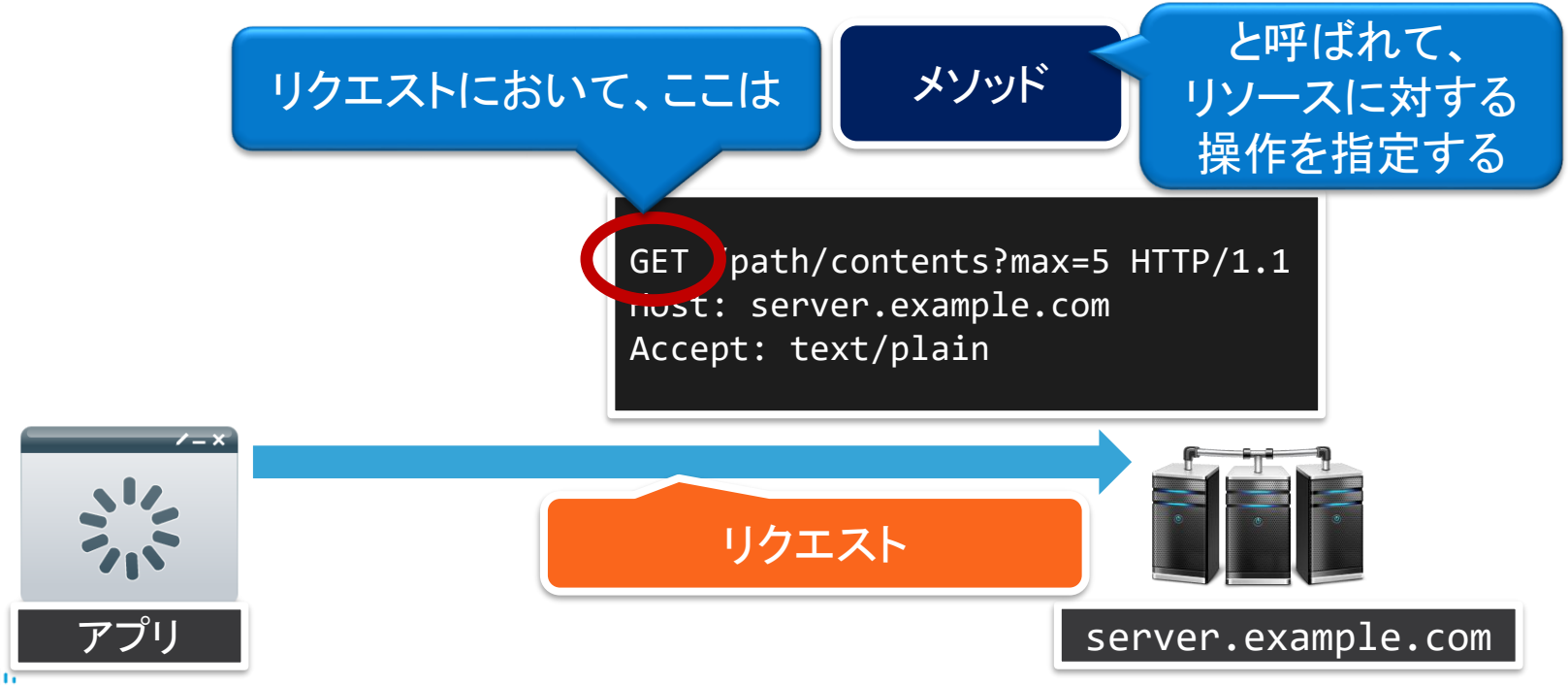
```
GET /path/contents?max=5 HTTP/1.1
Host: server.example.com
Accept: text/plain
Authorization: Bearer NDM5MjYz……
```

アプリ

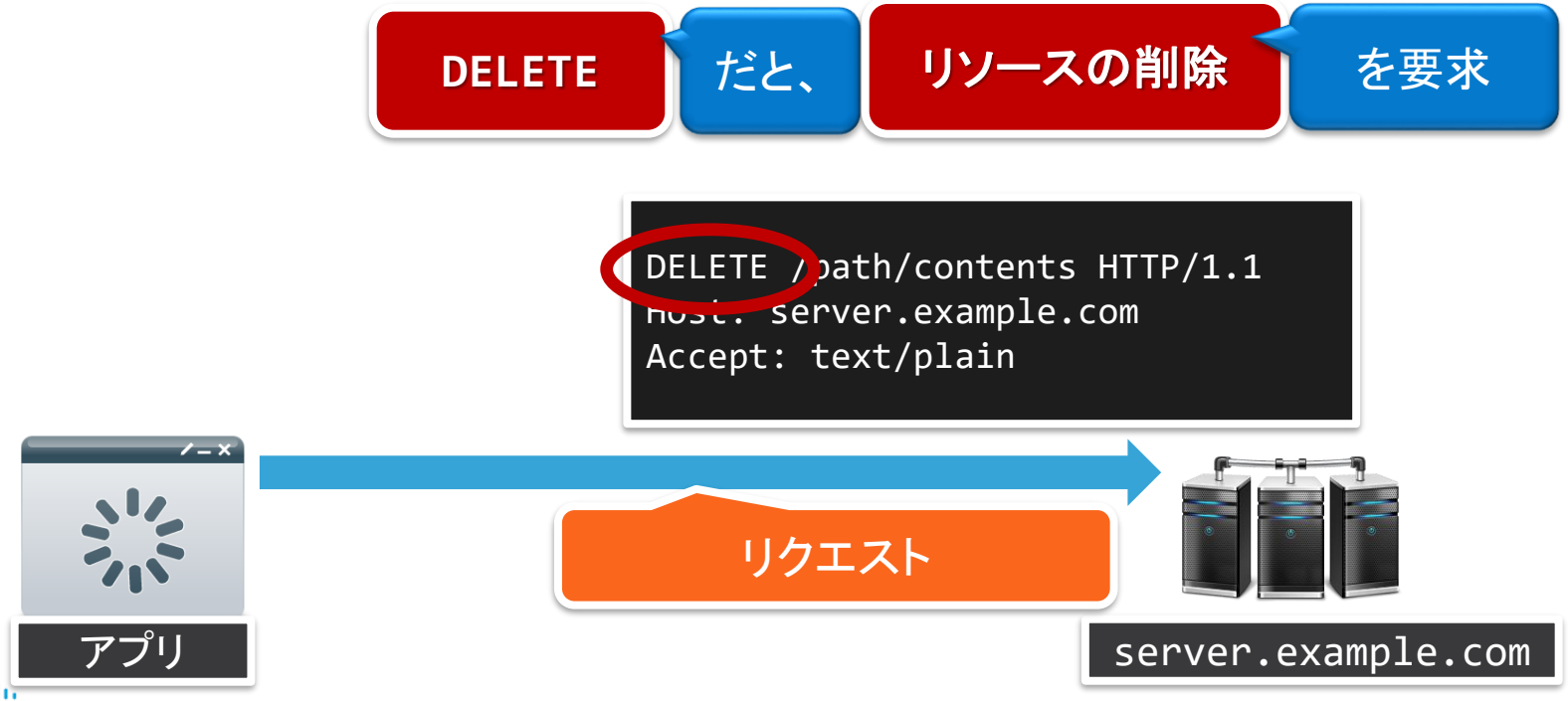
リクエスト

server.example.com

HTTPメソッド



HTTPメソッド



HTTPメソッド

メソッド の例

GET

POST

PUT

DELETE

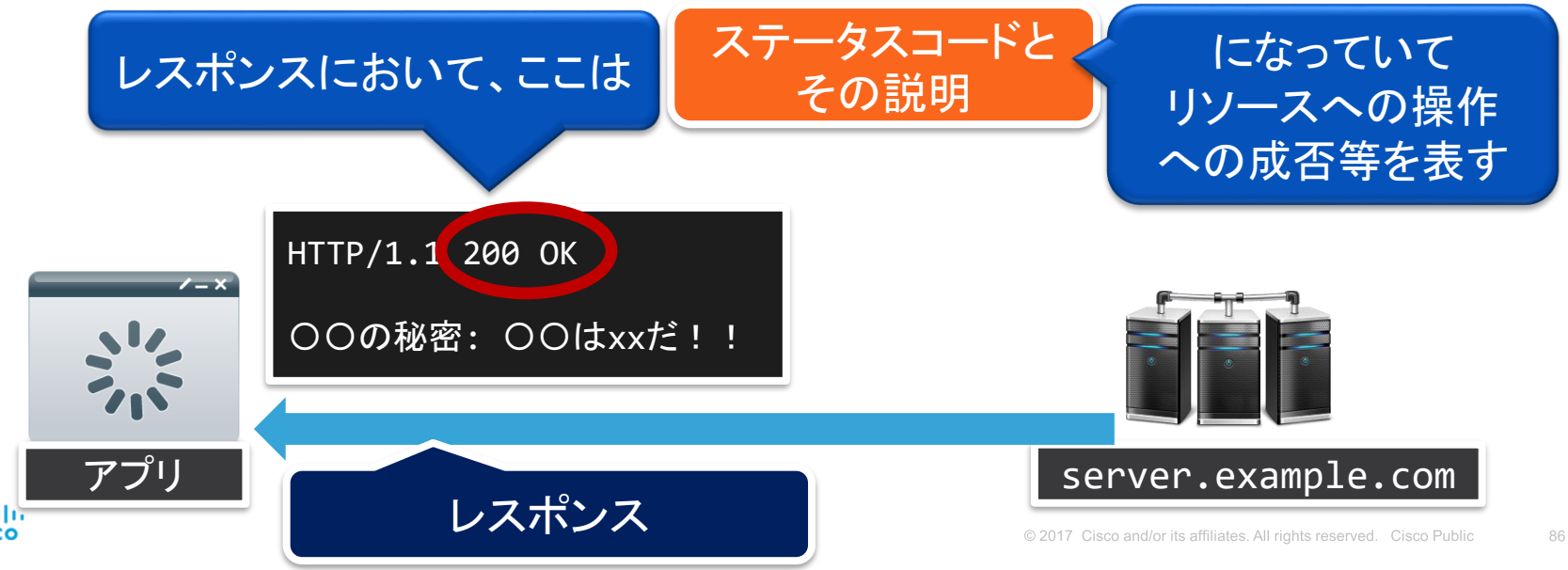
今日は、これらの
メソッドが重要になるが
あとで説明

アプリ

リクエスト

server.example.com

HTTPレスポンスコード



HTTPレスポンスコード

ステータスコードとその説明

の例

200 OK	成功
204 No Content	成功、本文(Body)なし(本文見る必要なし)
404 Not Found	探したけど、リソースが見つからなかった

アプリ

```
HTTP/1.1 200 OK

〇〇の秘密：〇〇はxxだ！！
```

レスポンス

server.example.com

REST API

REST API

HTTPの仕組みの上で、さらに、
一定の規約、決まり事を設けることで、
クライアントとサーバ間で
一貫したルールをもって、やりとりを行う

REST API

実際に例で
考えるのが
手っ取り早いので

Cisco Spark君に
再登場してもらいます

REST APIの例

Sparkの
スペース内のメッセージをイメージします

Sparkクライアントでの呼び方 → スペース / Space

Spark API上での名称 → ルーム / Room

歴史的な経緯(割愛)で、
呼び方には差異がありますが、
本日の範囲では同じものを指すと理解しておく

REST APIの例

リソースを表すURLは例えばこんな感じ

```
https://api.ciscospark.com/v1/messages
```

REST APIの例

リソースを表すURLは例えばこんな感じ

```
https://api.ciscospark.com/v1/messages
```

APIを提供するサーバのアドレス

REST APIの例

リソースを表すURLは例えばこんな感じ

```
https://api.ciscospark.com/v1/messages
```

リソースのパス

Sparkのスペースの場合は、 `/v1/rooms`

など、リソースの種類によって異なる

REST APIの例

リソースを表すURLは例えばこんな感じ

```
https://api.ciscospark.com/v1/messages
```

拡張性や互換性のためなど、
パスにバージョンが含まれる場合も多い

REST APIの例

リソースを表すURLには、2パターンある

```
https://api.ciscospark.com/v1/messages
```

```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```

REST APIの例

リソースを表すURLには、2パターンある

```
https://api.ciscospark.com/v1/messages
```

IDなし

```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```

IDあり

REST APIの例

リソースを表すURLには、2パターンある

```
https://api.ciscospark.com/v1/messages
```

IDなし

```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```

IDあり

IDあり、なしのパスが多段階でネストする場合もあります。
Spark APIのように、ネストはないシンプルな構造の方が、
APIとしては美しい(と思います / 個人的な感想)

REST APIの例

```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```

こいつを狙い撃ち

IDありの方は、
特定のメッセージを指す

このリソースは
世界にただ1つだけ
存在するのが原則

REST APIの例

```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```

一般には、以下の操作が可能と考えられる。
GET/PUT/DELETEは
操作に対応するHTTPのメソッド

GET	特定のメッセージの取得
PUT	特定のメッセージの更新
DELETE	特定のメッセージの削除

REST APIの例

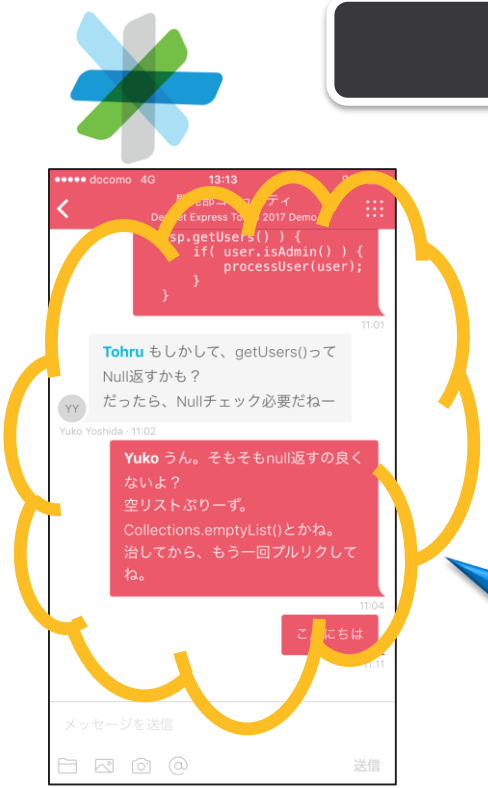
```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```

一般には、以下の操作が可能と考えられる。
GET/PUT/DELETEは
操作に対応するHTTPのメソッド

Sparkは既存のメッセージの
更新をサポートしないので、
メッセージの操作には
PUTは利用しない
(Sparkの機能から来る制限)

GET	特定のメッセージの取得
PUT	特定のメッセージの更新
DELETE	特定のメッセージの削除

REST APIの例




```
https://api.ciscospark.com/v1/messages
```

IDなしの方は、
一般のメッセージを指す
やや抽象的な概念

特定のどれかを
指すわけではない

REST APIの例

```
https://api.ciscospark.com/v1/messages
```

一般には、以下の操作が可能と考えられる。
GET/POST/PUT/DELETEは
操作に対応するHTTPのメソッド

GET	メッセージリストの取得
POST	新規メッセージの投稿
PUT	全てのメッセージの更新
DELETE	全てのメッセージの削除

cisco

REST APIの例

```
https://api.ciscospark.com/v1/messages
```

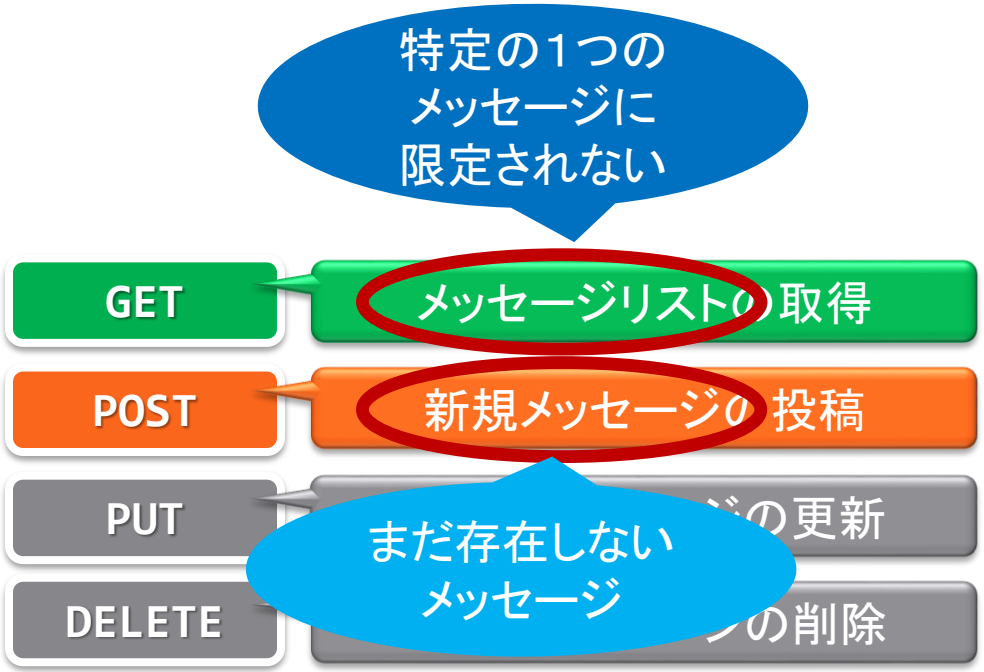
一般には、以下の操作が可能と考えられる。
GET/POST/PUT/DELETEは
操作に対応するHTTPのメソッド

Spark APIは複数の
メッセージに対する
一括更新、削除はサポート
しないので、
PUT/DELETEは利用しない
(Sparkの機能から来る制限)

GET	メッセージリストの取得
POST	新規メッセージの投稿
PUT	全てのメッセージの更新
DELETE	全てのメッセージの削除

REST APIの例

```
https://api.ciscospark.com/v1/messages
```



REST APIの例

表にするとこんな感じ！！

	IDなしのパス (例: .../messages)	IDありのパス (例: .../messages/Y2lzY29z...)
説明	該当パス上の一般的な リソースを表す	該当パスが表す特定の リソースを指す
GET	リソースのリストの取得	特定のリソースの詳細を取得
POST	リソースの新規作成	- (通常は使用しない)
PUT	リソース全体の更新	特定のリソースを変更する
DELETE	リソース全体の削除	特定のリソースを削除する

REST APIの例

	IDなしのパス (例: .../messages)	
説明	該当パス上の一般的な リソースを表す	 リソースを指す
GET	リソースのリストの取得	特定のリソースの詳細を取得
POST	リソースの新規作成	- (通常は使用しない)
PUT	リソース全体の更新	特定のリソースを変更する
DELETE	リソース全体の削除	特定のリソースを削除する

GETが最も安全な操作。
他の操作は、通常、リソースに
なんらかの変更がなされる

REST APIの例

	IDなしのパス (例: .../messages)	IDありのパス (例: .../messages/Y2lzY29z...)
説明	該当パス上の一般的なリソースを表す	
GET	リソースのリストの取得	
POST	リソースの新規作成	
PUT	リソース全体の更新	特定のリソースを変更する
DELETE	リソース全体の削除	特定のリソースを削除する

これらの機能は
提供されていない場合も多い
※ 大きな変化をもたらす可能性が高く、
通常は、危険度が高い処理

REST APIの例

Sparkの特定のメッセージ
を取得する例で、
もう少しREST APIの中身に
踏み込んでみよう

Spark APIを
処理するサーバ

REST APIの例

ある特定の
メッセージを
取得したい！

Spark APIを
処理するサーバ

アプリ

api.ciscospark.com

Sparkでメッセージを取得

```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```

特定のメッセージリソースの取得なので、IDありのパスに

GET

Sparkでメッセージを取得

```
https://api.cisc...
```

こんな感じでSparkのAPIサーバに
リクエストを出します

```
GET /v1/messages/Y2lzY29z... HTTP/1.1
Host: api.ciscospark.com
Accept: application/json
Authorization: Bearer NDM5MjYz……
Content-Type: application/json; encoding=utf-8
```

アプリ

api.ciscospark.com

Sparkでメッセージを取得

```
https://api.ciscospa
```

認証・認可のために
Botなどのトークン情報を渡しているので、
Spark APIサーバ側で、
Botなどの認証・認可等ができる

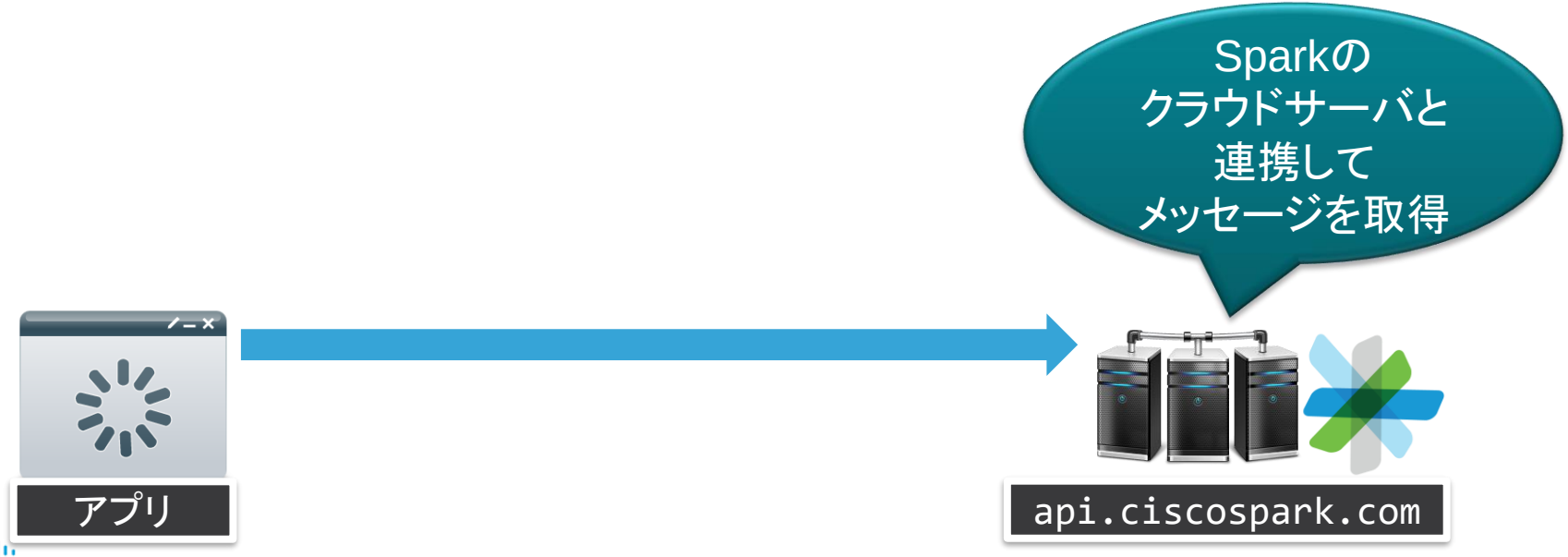
```
GET /v1/messages
Host: api.ciscospark.com
Accept: application/json
Authorization: Bearer NDM5MjYz……
Content-Type: application/json; encoding=utf-8
```

アプリ

api.ciscospark.com

Sparkでメッセージを取得

```
https://api.ciscospark.com/v1/messages/Y2lzY29z...
```



Sparkでメッセージを取得

Spark APIサーバは、
こんな感じのレスポンスを返す

```
HTTP/1.1 200 OK
Content-Length: 578
Content-Type: application/json; charset=utf-8

{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  (一部省略…)
  “text”: “こんにちは”,
  (一部省略…)
  “created”: “2017-03-06T02:56:41.512Z”
}
```

アプリ

api.ciscospark.com

JSON

```
HTTP/1.1 200 OK
Content-Length: 578
Content-Type: application/json; charset=utf-8

{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

JSON

```
HTTP/1.1 200 OK
Content-Length: 578
Content-Type: application/json; charset=utf

{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

リソースの中身の
本文(Body)の形式は何でもよいが

JSON を利用するREST APIが多い

JSON

```
HTTP/1.1 200 OK
Content-Length: 578
Content-Type: application/json; charset=utf-8

{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

これが JSON

説明用に、
実際のSparkの
レスポンスから
一部省略している

JSON

```
HTTP/1.1 200 OK
Content-Length: 578
Content-Type: application/json; charset=utf-8

{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

見た目以上に
多くの構成要素
で構成されている

- メッセージID
- 存在するスペース
- テキスト
- 作成日時
- 等々

JSON

JSONの例

```
{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

JSON

JSONの例

```
{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

1つの情報の塊を、{ }で囲んで
表現します。
この例では、Sparkのメッセージ情報
を表す情報の塊

JSON

JSONの例

情報の中身の各メンバーは

“キーとなる文字列” : 値

のペアになっていて

, で区切られる

```
{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

JSON

JSONの例

```
{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

このメッセージは、

roomId

というキー名とペアになっている値の

“Y2lzY29zcGF…”

という文字列のIDが指し示すスペースにある

JSON

JSONの例

```
{
  “id”: “Y2lzY29z…”,
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんにちは”,
  “created”: “2017-03-06T02:56:41.512Z”
}
```

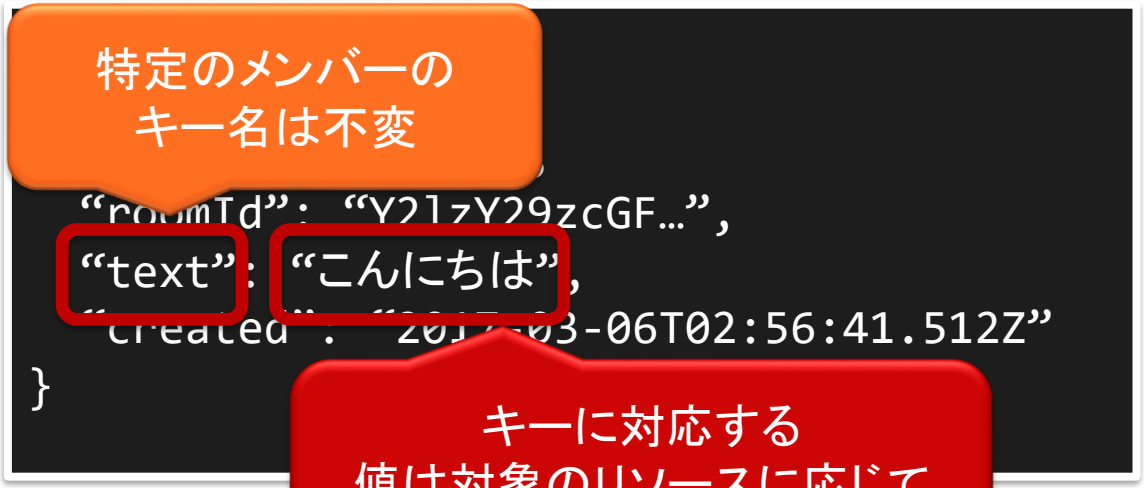
そのメッセージの文字列は、

`text` **というキー名とペアになっている値の**

`“こんにちは”` **という文字列**

JSON

JSONの例



Sparkでスペース内のメッセージの一覧を取得

Sparkのスペースの
メッセージの一覧を取得する例も
見てみよう

Spark APIを
処理するサーバ

Sparkでスペース内[illegible]ージの一覧を取得

スペース内の
メッセージの
一覧を取得したい！

Spark APIを
処理するサーバ

アプリ

api.ciscospark.com

Sparkでスペース内のメッセージの一覧を取得

```
https://api.ciscospark.com/v1/messages
```

メッセージリソースの一覧取得なので、IDなしのパスに

GET

Sparkでスペース内のメッセージの一覧を取得

```
https://api.ciscospark.com/v1/messages?roomId=Y2lzY29zcGF...&max=2
```

スペースは限定する必要があるので
パラメータとして指定します

Sparkでスペース内のメッセージの一覧を取得

```
https://api.ciscospark.com/v1/messages?roomId=Y2lzY29zcGF...&max=2
```

パス自体にIDが含まれる場合との
違いに注意しよう。
※ ここでは、メッセージの一覧の取得条件を
パラメータで指定して限定している

アプリ

api.ciscospark.com

Sparkでスペース内のメッセージの一覧を取得

```
https://api.ciscospark.com/v1/messages?roomId=Y2lzY29zcGF...&max=2
```

リストする最大件数も指定したいので
パラメータをさらに追加します

パラメータを複数指定する場合は　&　で区切る

アプリ

api.ciscospark.com

Sparkでスペース内のメッセージの一覧を取得

https://api.ciscospark.

こんな感じでSparkのAPIサーバに
リクエストを出します

GF...&max=2

```
GET /v1/messages?roomId=Y2lzY29zcGF...&max=2 HTTP/1.1
Host: api.ciscospark.com
Accept: application/json
Authorization: Bearer NDM5MjYz……
Content-Type: application/json; encoding=utf-8
```

アプリ

api.ciscospark.com

Sparkでスペース内のメッセージの一覧を取得

```
https://api.ciscospark.com/v1/messages?roomId=Y2lzY29zcGF...&max=2
```

Sparkの
クラウドサーバと
連携して
メッセージの一覧
を取得

アプリ

api.ciscospark.com

Sparkでスペース内のメッ

Spark APIサーバは、
こんな感じのレスポンスを返す。
(JSONをかなり省略しています)

```
https://api.cis                                    zcGF...&max=2
```

```
HTTP/1.1 200 OK
Content-Length: 578
Content-Type: application/json; charset=utf-8

{
  “items”: [
    {
      “id”: “Y2lzY29z…01”,
      “roomId”: “Y2lzY29zcGF…”,
      (一部省略…)
      “text”: “こんにちは”,
      (以降省略…)
  ]
}
```

アプリ

api.ciscospark.com

JSONの配列

```
{
  “items”: [
    {
      “id”: “Y2lzY29z...01”,
      “roomId”: “Y2lzY29zcGF...”,
      “text”: “こんにちは”,
      “created”: “2017-03-08T02:11:28.415Z”
    },
    {
      “id”: “Y2lzY29z...02”,
      “roomId”: “Y2lzY29zcGF...”,
      “text”: “おはよう”,
      “created”: “2017-03-13T06:31:46.747Z”
    }
  ]
}
```

今回のJSONはこんな感じ。
(説明に必要な部分以外省略している)

JSONの配列

情報の塊が複数含まれている

JSONの配列

```
{
  “items”: [
    {
      “id”: “Y2lzY29z...01”,
      “roomId”: “Y2lzY29zcGF...”,
      “text”: “こんにちは”,
      “created”: “2017-03-08T02:11:28.415Z”
    },
    {
      “id”: “Y2lzY29z...02”,
      “roomId”: “Y2lzY29zcGF...”,
      “text”: “おはよう”,
      “created”: “2017-03-13T06:31:46.747Z”
    }
  ]
}
```

配列(リスト)は、[ ]で囲んで
表現します。
配列には複数の情報の塊や
複数の値を含めることができます

JSONの配列

```
{
  “items”: [
    {
      “id”: “Y2lzY29z...01”,
      “roomId”: “Y2lzY29zcGF...”,
      “text”: “こんにちは”,
      “created”: “2017-03-08T02:11:28.415Z”
    },
    {
      “id”: “Y2lzY29z...02”,
      “roomId”: “Y2lzY29zcGF...”,
      “text”: “おはよう”,
      “created”: “2017-03-13T06:31:46.747Z”
    }
  ]
}
```

items

配列

というキー名とペアになっている値の

に複数のメッセージ情報が含まれている

Sparkでスペースにメッセージを投稿

Sparkのスペースに
メッセージを投稿する場合を
考えてみよう

Spark APIを
処理するサーバ

Sparkでスペースにメッセージを投稿

```
https://api.ciscospark.com/v1/messages
```

メッセージリソースの一覧取得なので、IDなしのパスに

POST

Sparkでスペースにメッセージを投稿

```
POST /v1/messages HTTP/1.1
Host: api.ciscospark.com
Accept: application/json
Authorization: Bearer NDM5MjYz……
Content-Type: application/json; encoding=utf-8

{
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんばんは”
}
```

こんな感じでSparkのAPIサーバに
リクエストを出します

アプリ

api.ciscospark.com

Sparkでスペースにメッセージを投稿

essages

```
POST /v1/messages HTTP/1.1
Host: api.ciscospark.com
Accept: application/json
Authorization: Bearer NDM5MjYz……
Content-Type: application/json; e

{
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんばんは”
}
```

投稿するので、
投稿対象のスペースのIDや、
投稿する文字列をJSONで
リクエストの本文(Body)の中で指定

アプリ

api.ciscospark.com

Sparkでスペースにメッセージを投稿

```
POST /v1/messages HTTP/1.1
Host: api.ciscospark.com
Accept: application/json
Authorization: Bearer NDM5MjYz……
Content-Type: application/json; e

{
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんばんは”
}
```

この時点で、
メッセージにはIDは存在しない
(まだメッセージ自体存在しない)

アプリ

api.ciscospark.com

Sparkでス

投稿者は、
認証・認可情報として指定した
トークンに対応するBotなどになります

```
POST /v1/messag
Host: api.ciscospark.com
Accept: application/json
Authorization: Bearer NDM5MjYz……
Content-Type: application/json; encoding=utf-8

{
  “roomId”: “Y2lzY29zcGF…”,
  “text”: “こんばんは”
}
```

アプリ

api.ciscospark.com

Sparkでスペースにメッセージを投稿

```
https://api.ciscospark.com/v1/messages
```

Sparkの
クラウドサーバと
連携して
メッセージを投稿

アプリ

api.ciscospark.com

Sparkでスペースにメッセ

Spark APIサーバは、
こんな感じのレスポンスを返す

```
ht
```

```
HTTP/1.1 200 OK
Content-Length: 578
Content-Type: application/json; charset=utf-8

{
  “id”: “Y2lzY29z…03”,
  “roomId”: “Y2lzY29zcGF…”,
  (一部省略…)
  “text”: “こんばんは”,
  (一部省略…)
  “created”: “2017-03-08T07:05:55.981Z”
}
```

アプリ

api.ciscospark.com

Sparkでスペースにメッセージを投稿

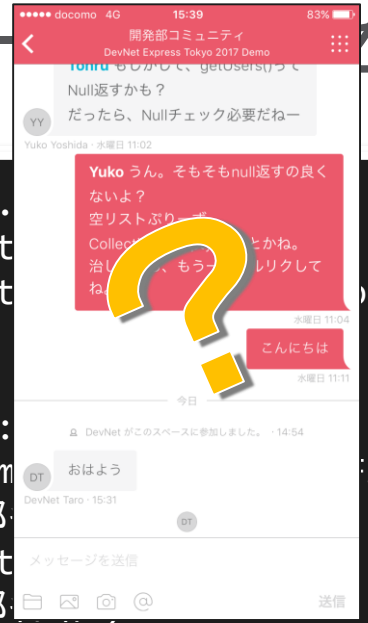
投稿されたメッセージを表すJSON。
メッセージの取得時と同じ形式。
この時点でメッセージのIDも出来上がっている。

```
ht
HTTP/1.
Content
Content-Type: app                    json; charset=utf-8

{
  “id”: “Y2lzY29z…03”,
  “roomId”: “Y2lzY29zcGF…”,
  (一部省略…)
  “text”: “こんばんは”,
  (一部省略…)
  “created”: “2017-03-08T07:05:55.981Z”
}
```

アプリ

api.ciscospark.com

Sparkでスペース内にメッセージを投稿




```
HTTP/1.
Content
Content

{
  “id”:
  “room                    :…”,
  (一部
  “text
  (一部
  “created”: “2017-03-08T07:05:55.981Z”
}
```

結果的に
Sparkクライアント上で
何が起こるかは
自分の目で
確かめてみよう！！

アプリ

api.ciscospark.com

アプリ？

Postman
というGUIベースの
Web APIの
テストクライアント

このアプリは
今日は何を使う？

Pythonの
実験用プログラムを
自分で編集して
実行します

REST APIのメリットの例
– プログラミング言語の視点から -

REST API

HTTPの仕組みの上で動作する

HTTPのやり取りができる言語ならREST APIが実行できる

ほとんどの言語でHTTPは利用できる

ほとんどの言語でREST APIが実行できる

REST APIのメリットの例
– プログラミング言語の視点から -

REST API

HTTPの仕組みの上で動作する

HTTPのやり取りができる言語ならREST APIが実行できる

ほとんどの言語でHTTPは利用できる

もちろん
Pythonでも
できる

ほとんどの言語でREST APIが実行できる

Python

Pythonとは

ハンズオンの
資料から抜粋

インタプリタ形式でオブジェクト指向の言語であり、
ダイナミックセマンティクスに対応した
高水準プログラミング言語です。

Pythonとは

ハンズオンの
資料から抜粋

なんじゃそりゃあ？！

インタプリタ形式でオブジェクト指向の言語であり、
ダイナミックセマンティクスに対応した
高水準プログラミング言語です。

Pythonとは

ハンズオンの
資料から抜粋

とりあえず、
さわってみて
雰囲気を
味わってみないと
何もはじまらない

インタプリタ形式でオブジェクト指向の言語であり、
ダイナミックセマンティクスに対応した
高水準プログラミング言語です。

Pythonとは

ハンズオンの
資料から抜粋

インタプリタ形式[redacted]の[redacted]
[redacted]
[redacted]プログラミング言語です。

Pythonとは

ハンズオンの
資料から抜粋

インタプリタ形式[redacted]の[redacted]
[redacted]
[redacted]プログラミング言語です。

今日はこの辺を
味わってみる

簡単なPythonプログラムから始めてみる

```
print(‘Hello, Python!!’)
```

簡単なPythonプログラムから始めてみる

```
print(‘Hello, Python!!’)
```

プログラミングとは
コンピュータに対して指示を与えていく作業です。
1つ1つの指示はごく単純

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

プログラミングとは
コンピュータに対して指示を与えていく作業です。
1つ1つの指示はごく単純

プログラミング言語では、
その指示の与え方に決まり(文法)がある。
文法を守らないとエラーになる

簡単なPythonプログラムから始めてみる

```
print(‘Hello, Python!!’)
```

表示せよ

という指示

簡単なPythonプログラムから始めてみる

```
print(‘Hello, Python!!’)
```

表示する内容

‘ ’ または、 “ ” で囲むと、文字列を表すことになる

Hello, Python!! という文字列を 表示せよ の指示になる

※ 実際には、「表示して改行せよ」

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

これを実行すると、
コンピュータは何らかの方法で

Hello, Python!! **と表示する**

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

実行？
どうやって？

これを実行すると、
コンピュータは何らかの方法で

Hello, Python!!

と表示する

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

例

こいつをファイルとして保存する

例えば、

hello.py

実行？
どうやって？

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

例

こいつをファイルとして保存する

例えば、

hello.py

実行？
どうやって？

```
> python hello.py
Hello, Python!!
```

コマンドラインなどから

pythonコマンド

にファイル名を渡して実行

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

例

こいつをファイルとして保存する

例えば、

hello.py

指示した通りの文字列が
表示される

```
> python hello.py
Hello, Python!!
```

コマンドラインなどから

pythonコマンド

にファイル名を渡して実行

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

簡単なPythonプログラムから始めてみる

```
print('Hello, Python!!')
```

pythonコマンド

```
> python hello.py
Hello, Python!!
```

コマンドを実行すると、
ファイルを読み込んで、
プログラミングされた指示を
1つずつ解釈しながら
実行していきます

このような特徴を持つ
プログラムの実行環境を

インタプリタ

と呼ぶ

簡単なPythonプログラムから始めてみる

```
print(‘Hello, Python!!’)
print(‘How are you?’)
print(‘See you again!!’)
```

上から順番に指示した通りに
実行されるのが基本

変数

```
who = ‘Tohru’

print(‘My name is ’ + who)
```

変数を使うと
値を覚えておいて、
後で参照できる

変数

変数

```
who = ‘Tohru’

print(‘My name is ’ + who)
```

=(イコール記号)

に続けて

代入したい値を指定している

この例では

Tohru

という文字列を代入

変数

```
who = ‘Tohru’

print(‘My name is ’ + who)
```

代入された値を
ここで参照

この例では

Tohru

という文字列が代入
されている

変数

```
who = ‘Tohru’

print(‘My name is ’ + who)
```

足し算の記号(演算子)を使って、
文字列を連結できる

変数

```
who = ‘Tohru’

print(‘My name is ’ + who)
```

```
My name is Tohru
```

実行するとこうなる

変数

```
a = 2
b = 3

print(a + b)
print(a - b)
print(a * b)
```

では、これは？
(考えてみよう)

変数

‘ ’や” ”で囲まれていないので
これは文字列ではなく、
数値です

```
a = 2
b = 3

print(a + b)
print(a - b)
print(a * b)
```

変数

変数

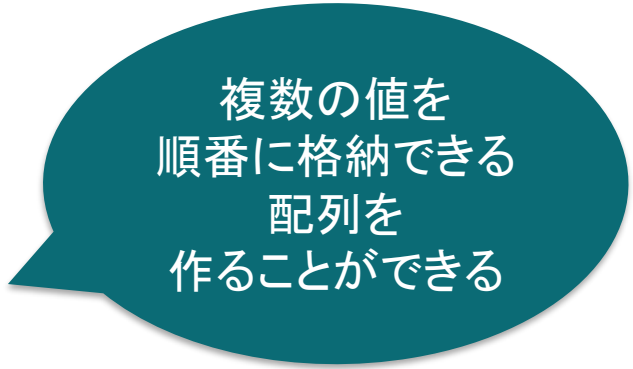
```
a = 2
b = 3

print(a + b)
print(a - b)
print(a * b)
```

```
5
-1
6
```

実行するとこうなる

配列

```
arr = [‘C’, ‘i’, ‘sco’]
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```



配列

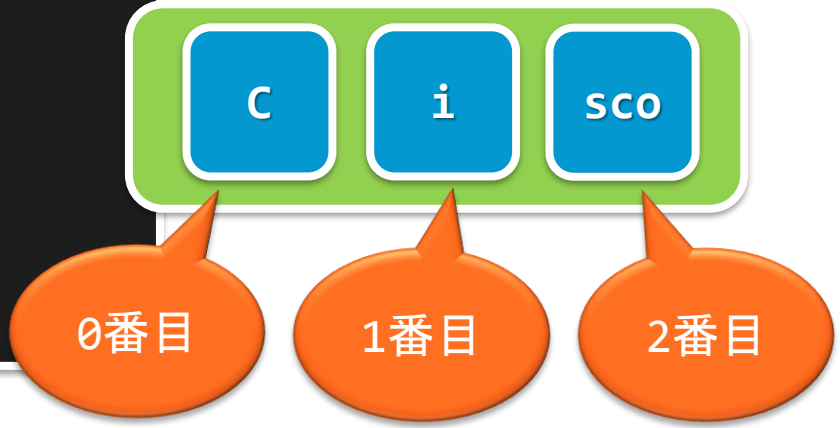
[ ]で囲ってカンマ(,)で区切って、
その中に値を入れていきます

```
arr = ['C', 'i', 'sco']
who = arr[0] + arr[1] + arr[2]

print('Hello, ' + who)
```

こんな感じで
順番に値が入っているイメージ

配列

```
arr = [‘C’, ‘i’, ‘sco’]
who = arr[0] + arr[1] + arr[2]
```



アクセスするときは、

変数名[中身の番号]

でアクセスする

0番目から開始することに注意

配列

```
arr = [‘C’, ‘i’, ‘sco’]
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

順番に中身にアクセスして
連結しているので

配列

```
arr = [‘C’, ‘i’, ‘sco’]
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

```
Hello, Cisco
```

実行するとこうなる

配列の中身の変更

```
arr = [‘C’, ‘i’, ‘sco’]
arr[0] = ‘B’
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

配列の中身の変更

```
arr = [‘C’, ‘i’, ‘sco’]
arr[0] = ‘B’
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

配列は中身の追加、削除、変更などができます。
この例では、0番目を‘B’に変更

配列の中身の変更

```
arr = [‘C’, ‘i’, ‘sco’]
arr[0] = ‘B’
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

```
Hello, Bisco
```

実行するとこうなる

タプル

```
arr = (‘C’, ‘i’, ‘sco’)
arr[0] = ‘B’
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

タプル

( )で囲んで、カンマ(,)で区切ると
中身の変更できない配列(タプル)
を作ることができる

```
arr = (‘C’, ‘i’, ‘sco’)
arr[0] = ‘B’
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

タプル

```
arr = (‘C’, ‘i’, ‘sco’)
arr[0] = ‘B’
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

変更しようとすると
失敗します

```
 arr[0] = ‘B’
TypeError: ‘tuple’ object does not support item assignment
```

タプル

```
arr = (‘C’, ‘i’, ‘sco’)
who = arr[0] + arr[1] + arr[2]

print(‘Hello, ’ + who)
```

```
Hello, Cisco
```

変更している個所をなくすと
こんな感じ

ディクショナリ

```
dic = { ‘name’ : ‘Tohru’, ‘email’ : ‘tohzono@example.com’ }

print(dic[‘name’])
print(dic[‘email’])
```

ディクショナリ

{ }で囲んで、カンマ(,)で区切って
ディクショナリ
というものを作ることができる

```
dic = { ‘name’ : ‘Tohru’, ‘email’ : ‘tohzono@example.com’ }

print(dic[‘name’])
print(dic[‘email’])
```

ディクショナリ

```
dic = { ‘name’ : ‘Tohru’, ‘email’ : ‘tohzono@example.com’ }

print(dic[‘name’])
print(dic[‘email’])
```

キーとなる値 : キーに対応する値

のペアになっていて

, で区切られる

JSONに
非常に似ている！！

ディクショナリ

```
dic = { ‘name’ : ‘Tohru’, ‘email’ : ‘tohzono@example.com’ }

print(dic[‘name’])
print(dic[‘email’])
```

値にラベル付けするようなイメージ

ディクショナリ

```
dic = { ‘name’ : ‘Tohru’, ‘email’ : ‘tohzono@example.com’ }

print(dic[‘name’])
print(dic[‘email’])
```

アクセスするときは、

変数名[キーとなる値]

でアクセスする

ディクショナリ

```
dic = { ‘name’ : ‘Tohru’, ‘email’ : ‘tohzono@example.com’ }

print(dic[‘name’])
print(dic[‘email’])
```

```
Tohru
tohzono@example.com
```

実行するとこうなる

分岐処理

```
a = 2

if a == 1:
    print(‘aは1です。’)
    print(‘aは2ではないです。’)

if a == 2:
    print(‘aは2です。’)
    print(‘aは1ではないです。’)
```

分岐処理

```
a = 2

if a == 1:
    print(‘aは1です。’)
    print(‘aは2ではないです。’)

if a == 2:
    print(‘aは2です。’)
    print(‘aは1ではないです。’)
```

条件に応じて、
処理を分けたい場合があります

分岐処理

```
a = 2

if a == 1:
    print(‘aは1です。’)
    print(‘aは2ではないです。’)

if a == 2:
    print(‘aは2です。’)
    print(‘aは1ではないです。’)
```

こんな感じで

if 条件:

という書き方をする

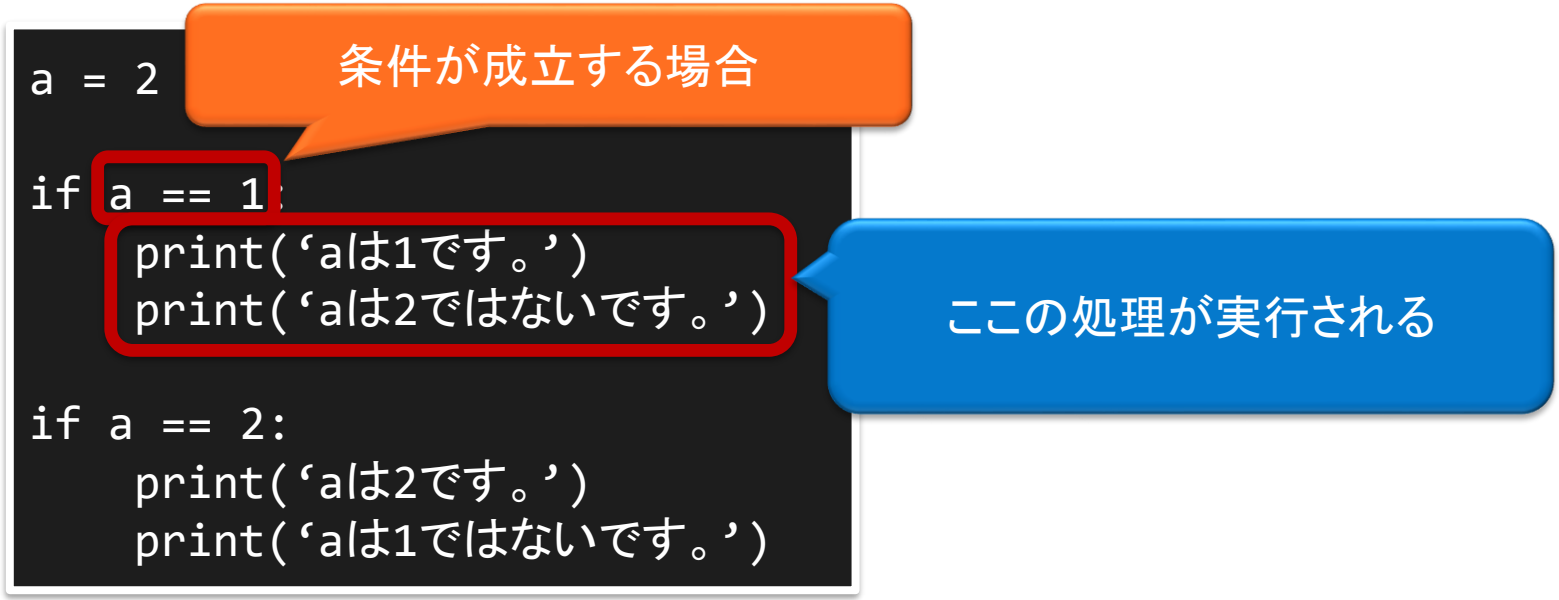
最後のコロン(:)を
忘れないように

分岐処理

```
a = 2

if a == 1:
    print(‘aは1です。’)
    print(‘aは2ではないです。’)

if a == 2:
    print(‘aは2です。’)
    print(‘aは1ではないです。’)
```

左辺と右辺が等しいか
どうかを確認する記号(演算子)

分岐処理



分岐処理

```
a = 2

if a == 1:
    print(‘aは1です。’)
    print(‘aは2ではないです。’)
```

ここのインデントはかなり重要

インデントによって、
条件成立時に実行される
範囲が決まる。
実行したい処理は、
同じインデントを使って書く

分岐処理

```
a = 2

if a == 1:
    print(‘aは1です。’)
    print(‘aは2ではないです。’)
```

基本的には、
半角スペース4つを入れよう!!

分岐処理

```
a = 2

if a == 1:
    print(‘aは1です。’)
    print(‘aは2ではないです。’)

if a == 2:
    print(‘aは2です。’)
    print(‘aは1ではないです。’)
```

実行するとこうなる

```
aは2です。
aは1ではないです。
```

繰り返し処理

```
for i in range(3):
    print(‘Hello, Python!!’)
    print(‘See you again!!’)
```

繰り返し処理

```
for i in range(3):
    print(‘Hello, Python!!’)
    print(‘See you again!!’)
```

同じ処理を繰り返したい
ことがあります

繰り返し処理

```
for i in range(3):
    print('Hello, Python!!')
    print('See you again!!')
```

今日は、詳細な理解はあきらめて、
形でとらえる

繰り返し処理

```
for i in range(3):
    print(‘Hello, Python!!’)
    print(‘See you again!!’)
```

こんな感じで

for i in range(繰り返したい回数):

という書き方をする

最後のコロン(:)を
忘れないように

繰り返し処理

```
for i in range(3):
    print('Hello, Python!!')
    print('See you again!!')
```

インデントされている
この範囲が繰り返される

繰り返し処理

```
for i in range(3):
    print(‘Hello, Python!!’)
    print(‘See you again!!’)
```

```
Hello, Python!!
See you again!!
Hello, Python!!
See you again!!
Hello, Python!!
See you again!!
```

実行するとこうなる

繰り返し処理

```
arr = [‘C’, ‘i’, ‘sco’]

for value in arr:
    print(value)
```

配列などを
取り出しながら
繰り返しすることも
できる

こちらが正しく理解できれば、
for i in range(繰り返したい回数):
も適切に理解できるはずだが、
初めてプログラミングを学ぶ場合は、
難しいと思うので、とりあえずは、
こんな感じという捉え方でよいと思います

cisco

繰り返し処理

```
arr = [‘C’, ‘i’, ‘sco’]

for value in arr
    print(value)
```

配列の中身を取り出しながら、
繰り返します。

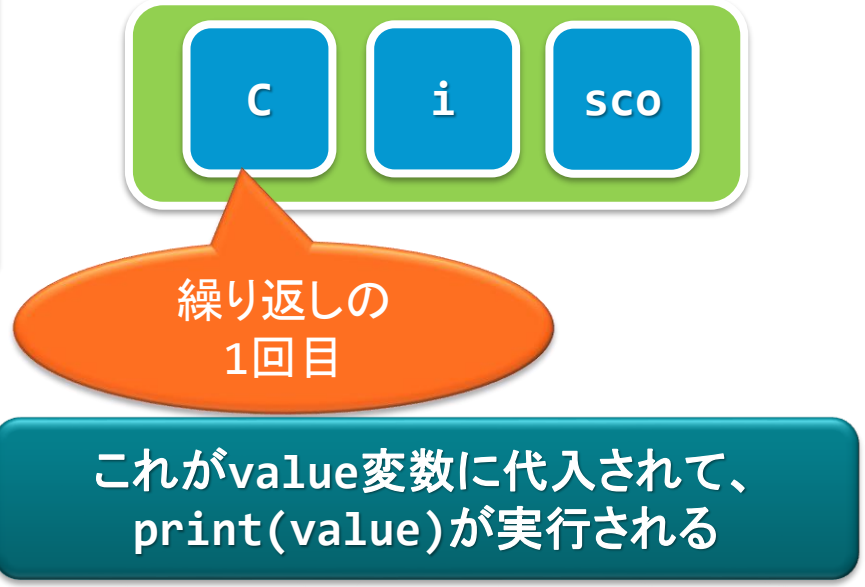
繰り返し処理

```
arr = [‘C’, ‘i’, ‘sco’]

for value in arr:
    print(value)
```

取り出した値は、
指定した変数に代入される

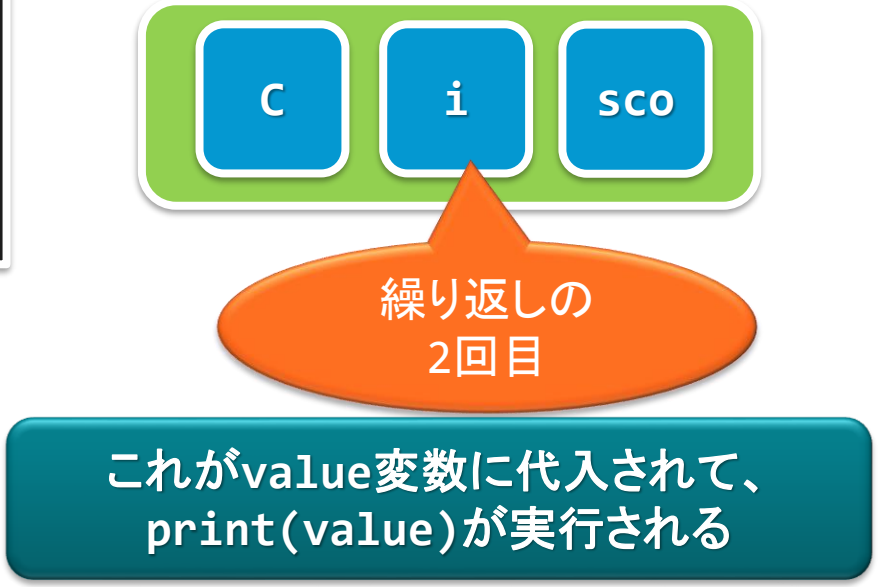
繰り返し処理

```
arr = [‘C’, ‘i’, ‘sco’]

for value in arr:
    print(value)
```



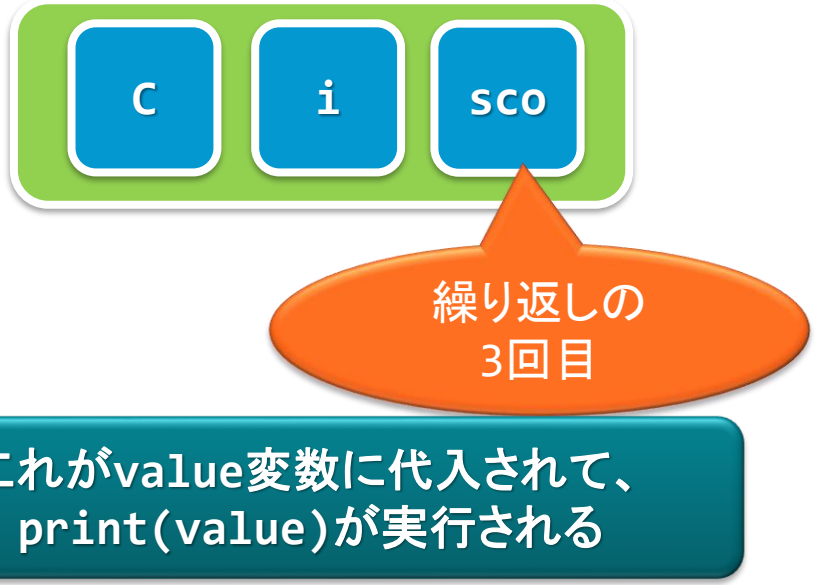
繰り返し処理

```
arr = [‘C’, ‘i’, ‘sco’]

for value in arr:
    print(value)
```



繰り返し処理

```
arr = [‘C’, ‘i’, ‘sco’]

for value in arr:
    print(value)
```



繰り返し処理

```
arr = [‘C’, ‘i’, ‘sco’]

for value in arr:
    print(value)
```

```
C
i
sco
```

実行するとこうなる

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

よく使う処理は
関数というものにしておくと便利

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

これが関数の定義

def 関数名(関数に渡す引数のリスト):

の後のインデントで区切られた範囲が、
関数の定義範囲

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

定義されているだけなので、
この時点では実行されない

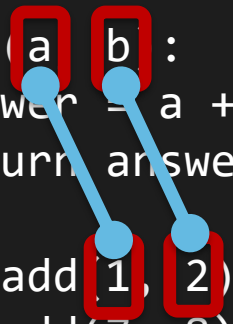
関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

定義された名前で
呼び出しています

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```



対応する順番の引数に
値が代入された後に
関数が実行される

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

a + b
を計算して、
answer変数に代入

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

answer変数の値を
結果として返す(return)
という意味

結果は変数に代入して
受け取れます

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

渡す値を変えて、
同じように何度でも呼び出せる

関数

```
def add(a, b):
    answer = a + b
    return answer

ans1 = add(1, 2)
ans2 = add(7, 8)

print(ans1)
print(ans2)
```

実行するとこうなる

```
3
15
```

便利なモジュールの利用

モジュール

今日の段階では、
いろんな関数が含まれていて
プログラミングの時に
便利に使えるもの程度の理解で良いと思います

頭の良い誰かの作ったモジュールを活用すると、
本来複雑な処理がとんでもなく簡単に！

便利なモジュールの利用

```
https://server.example.com/contents
```

サーバから
リソース取得する、
この処理、何行で書けますか？

```
GET /contents HTTP/1.1
```

アプリ

server.example.com

便利なモジュールの利用

```
import requests

res = requests.get('https://server.example.com/contents')

print(res.text)
```

成功したかのチェックなどを
しなければ3行

（本来は必ずチェックすべきだが）

便利なモジュールの利用

```
import requests

res = requests.get('                                    s')

print(res.text)
```

誰かが作ったモジュールを取り込む

import 取り込むモジュール名

で取り込みます

便利なモジュールの利用

```
import requests

res = requests.get('https://server.example.com/contents')

print(res.text)
```

取り込んだモジュールの関数を呼び出す

モジュール名.関数名(引数のリスト)

のように呼び出します

便利なモジュールの利用

```
import requests

res = requests.get('https://server.example.com/contents')
```

結果は変数に代入して
受け取れます

便利なモジュールの利用

```
import requests

res = requests.get('https://server.example.com/contents')

print(res.text)
```

結果の中身にアクセスするために、
こんな感じで、ドット(.)付きでアクセスする場合がある

- res.status_code → HTTPステータスコード
- res.text → レスポンスのテキスト

などなど、
モジュール側の
実装による

便利なモジュールの利用

```
import requests

res = requests.get('https://server

print(res.text)
```

今日の範囲外である、
“クラス”というものを
理解していないと
正確な理解は難しい。
本日の範囲では、
こんなものかで受け入れる

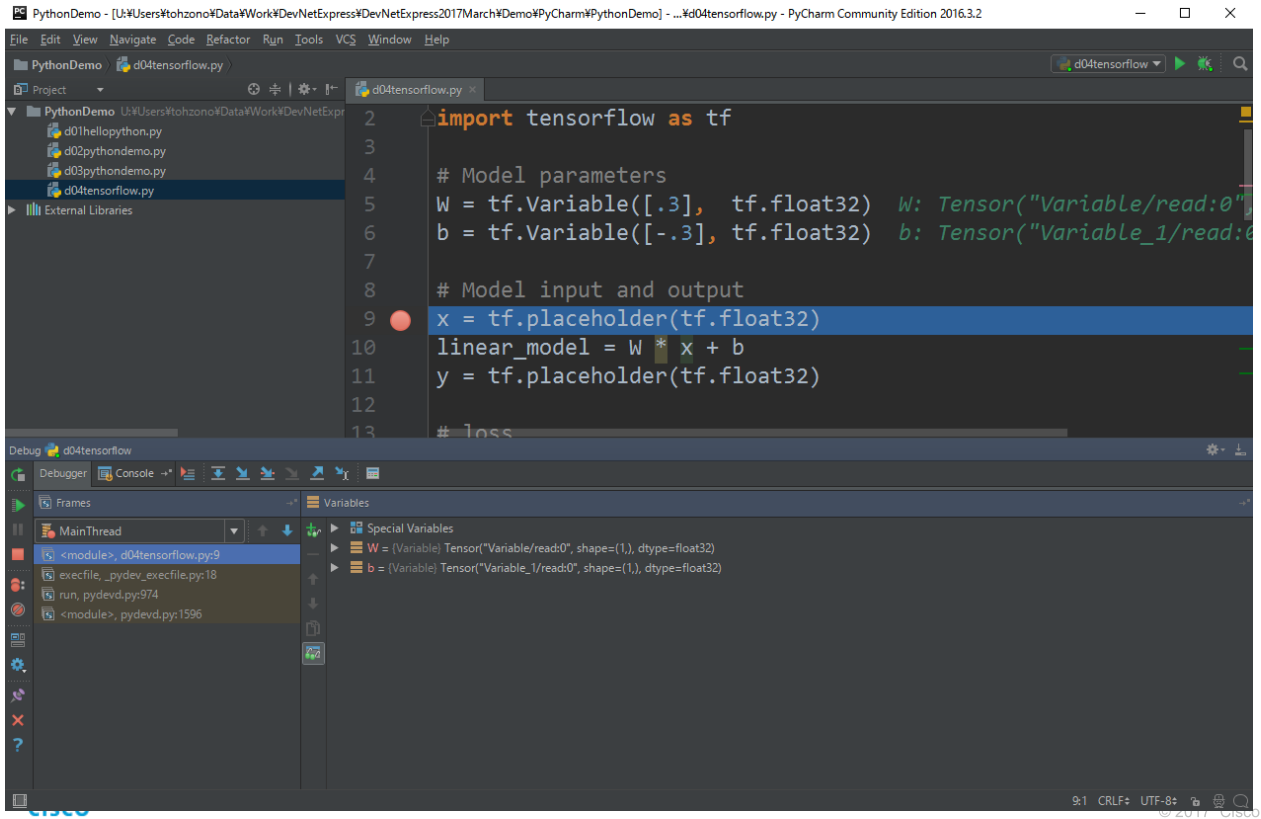
結果の中身にアクセスするために、
こんな感じで、ドット(.)付きでアクセスする場合がある

res.status_code → HTTPステータスコード

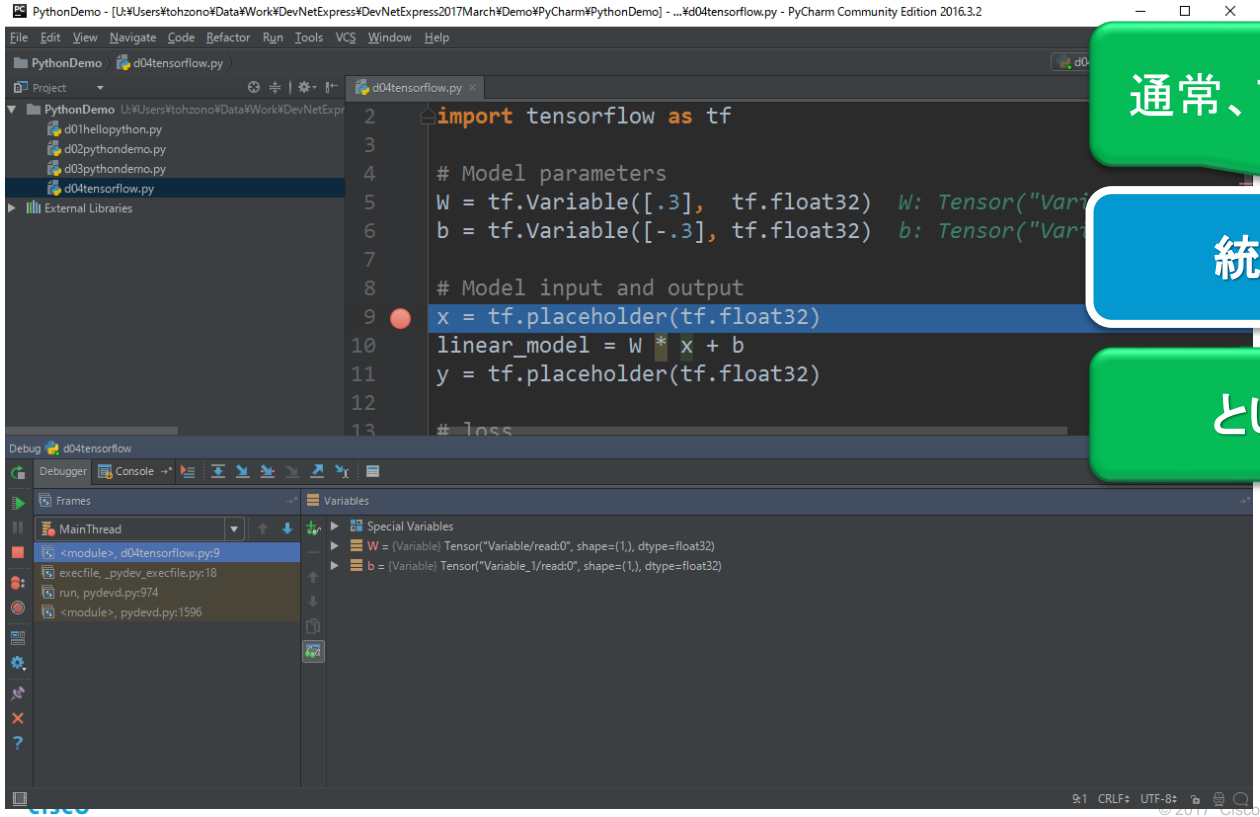
res.text → レスポンスのテキスト

などなど、
モジュール側の
実装による

慣れてきたらIDEを活用しよう



慣れてきたらIDEを活用しよう



通常、プログラミングなどには

統合開発環境(IDE)

というツール群を使う

慣れてきたらIDEを活用しよう

統合開発環境(IDE)

- ソースコードを書く際に
  次に書く候補を教えてくれたり、
  間違いを指摘してくれたり、
  様々なコード作成支援機能
- デバッグの支援機能
- 単体試験の支援機能
- Git / GitHubなどの
  バージョン管理システムとの連携

などなど、
アプリ開発を支援する
強力な機能が提供される

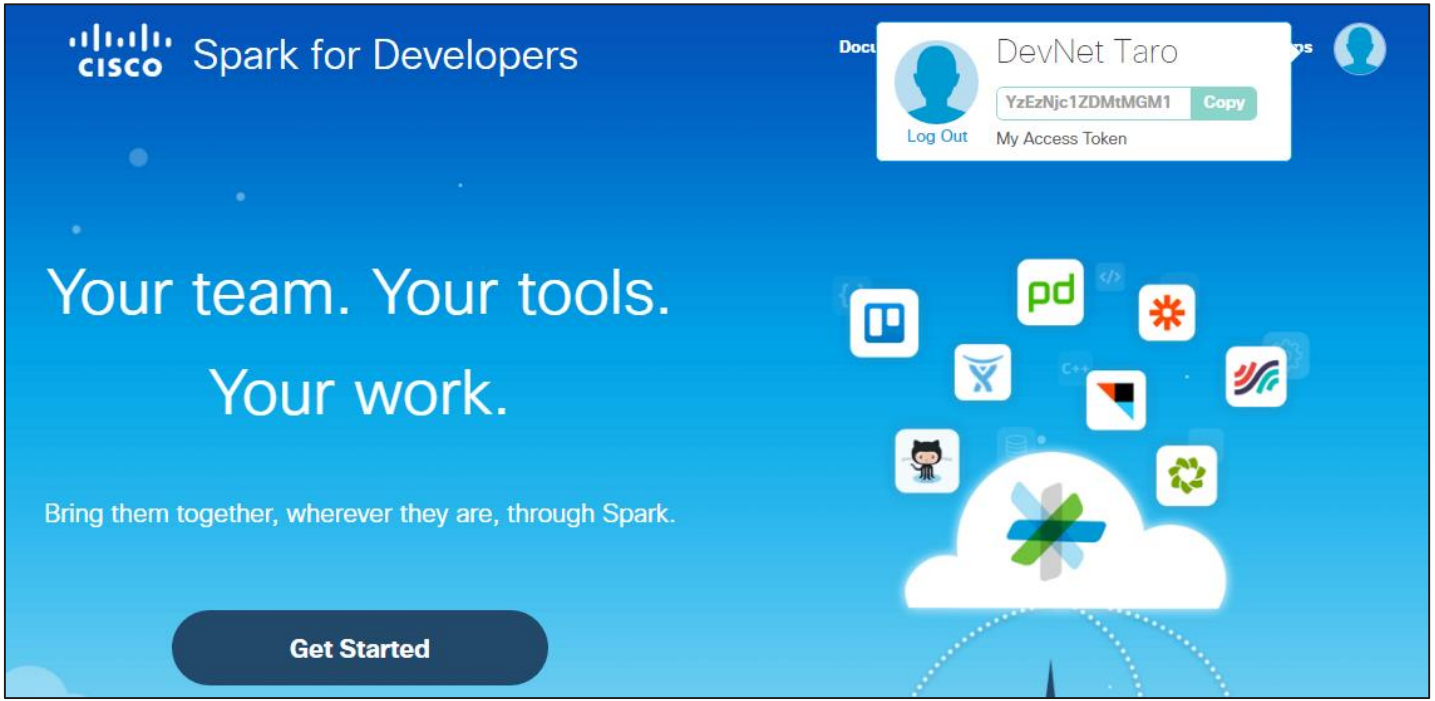
慣れてきたらIDEを活用しよう

統合開発環境(IDE)

PyCharm
Eclipse + PyDev
Visual Studio
などPythonをサポートするIDEは多く存在する

補足

開発者用トークンの注意

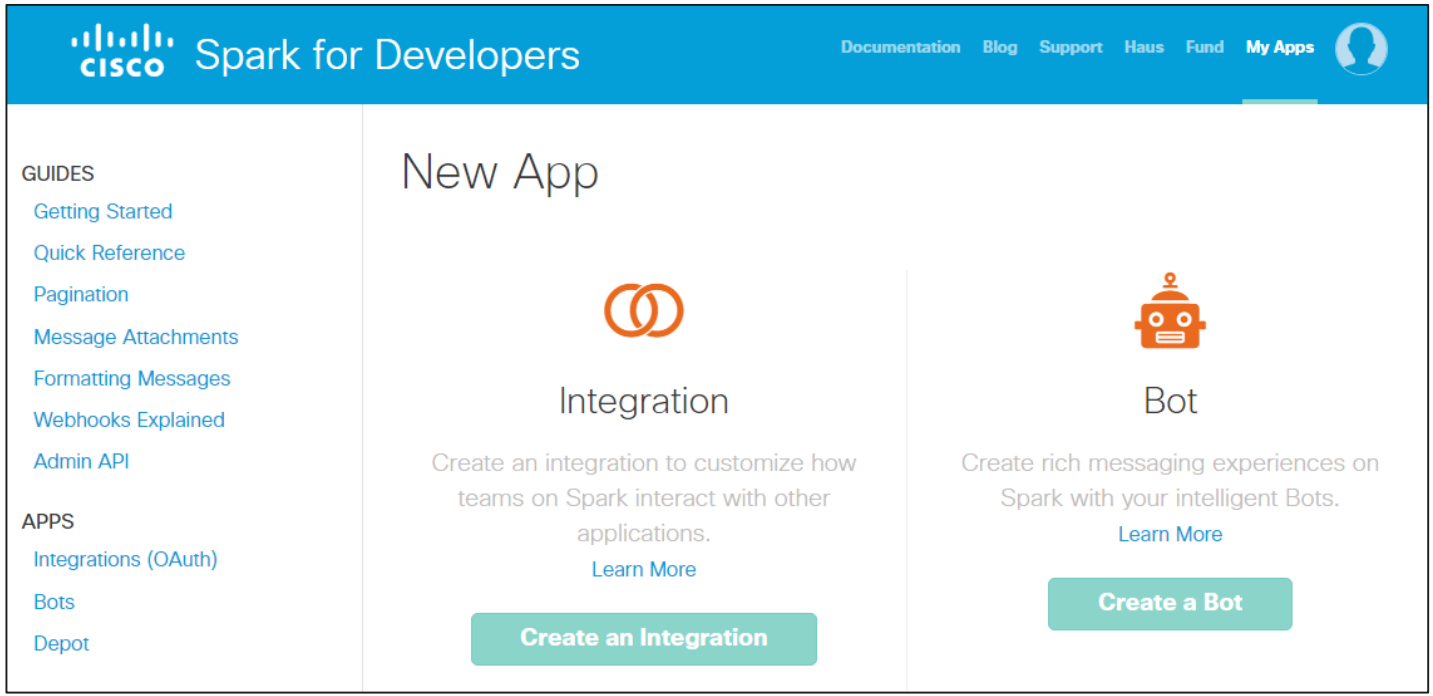


開発者用トークンの注意

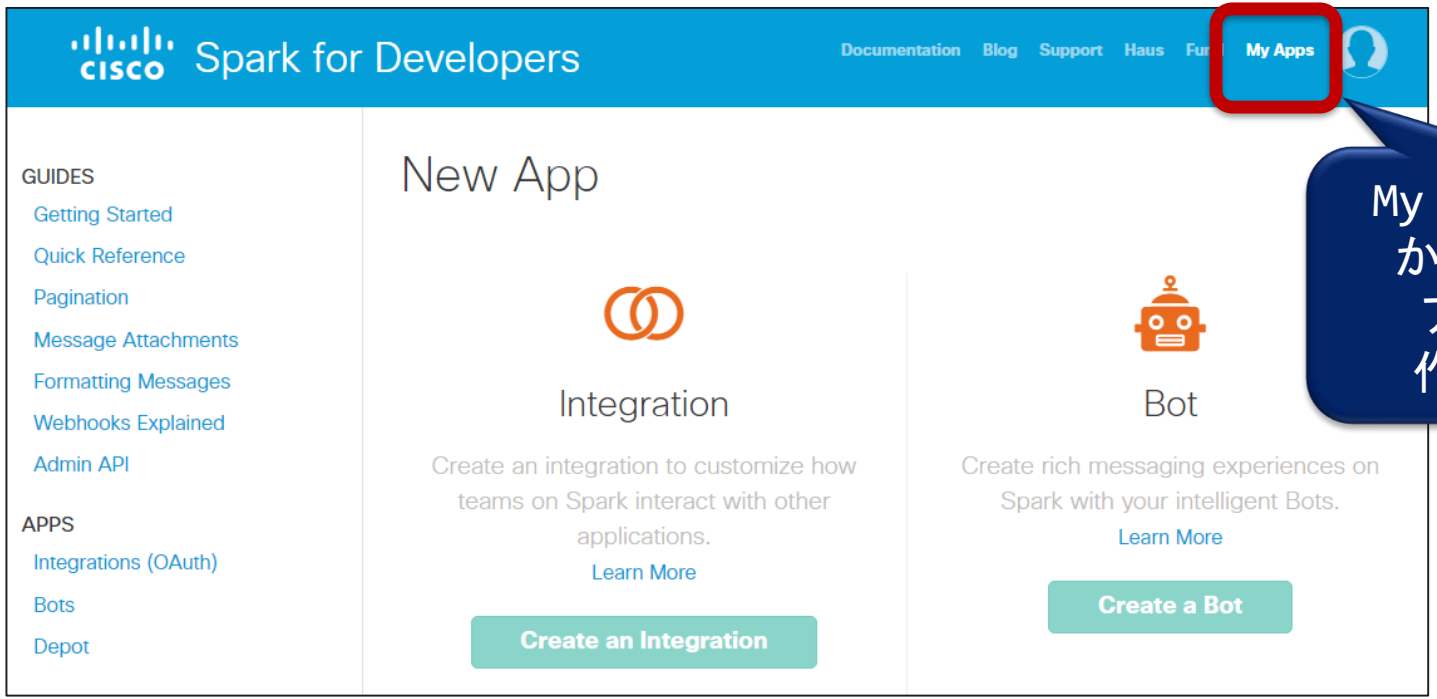
ここで取得できるトークンは
開発者トークンです

開発、実験時など
限られた用途でのみ利用可能です。
運用環境などでは
絶対に利用しないでください

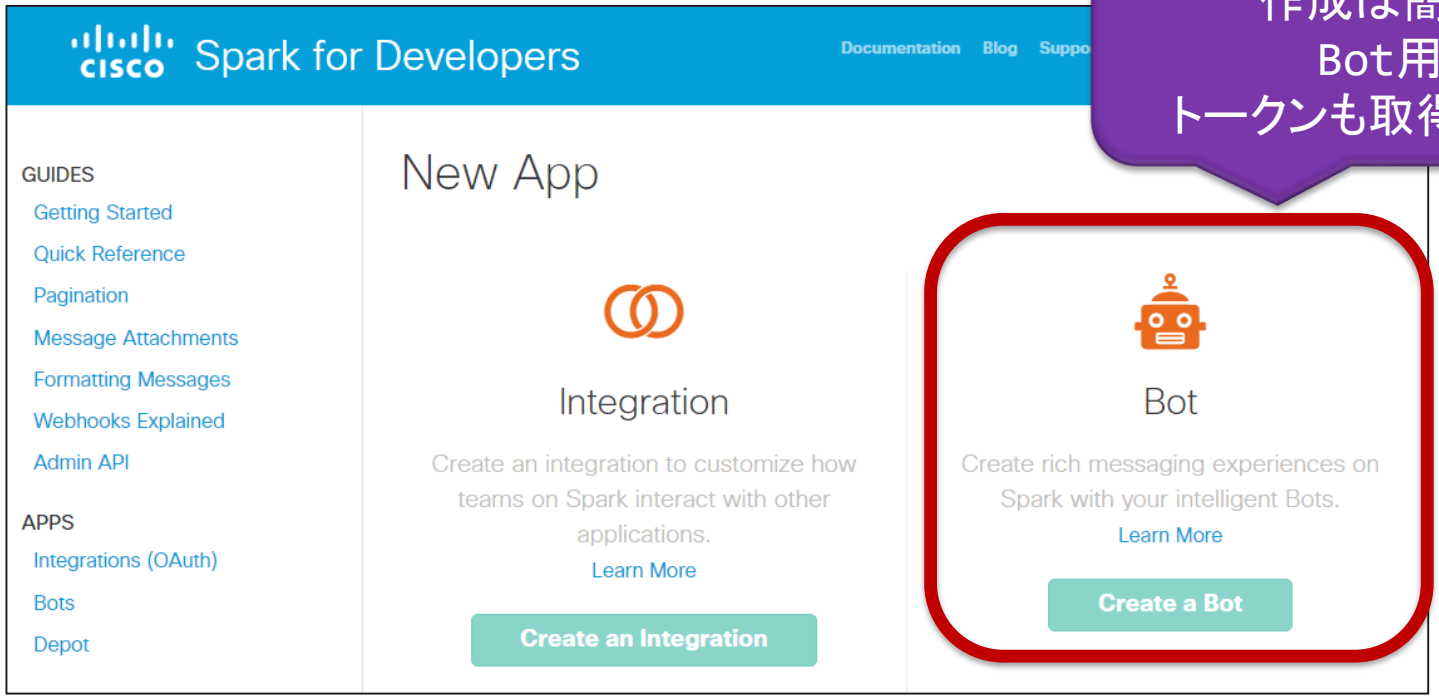
アプリケーション用のトークン



アプリケーション用のトークン

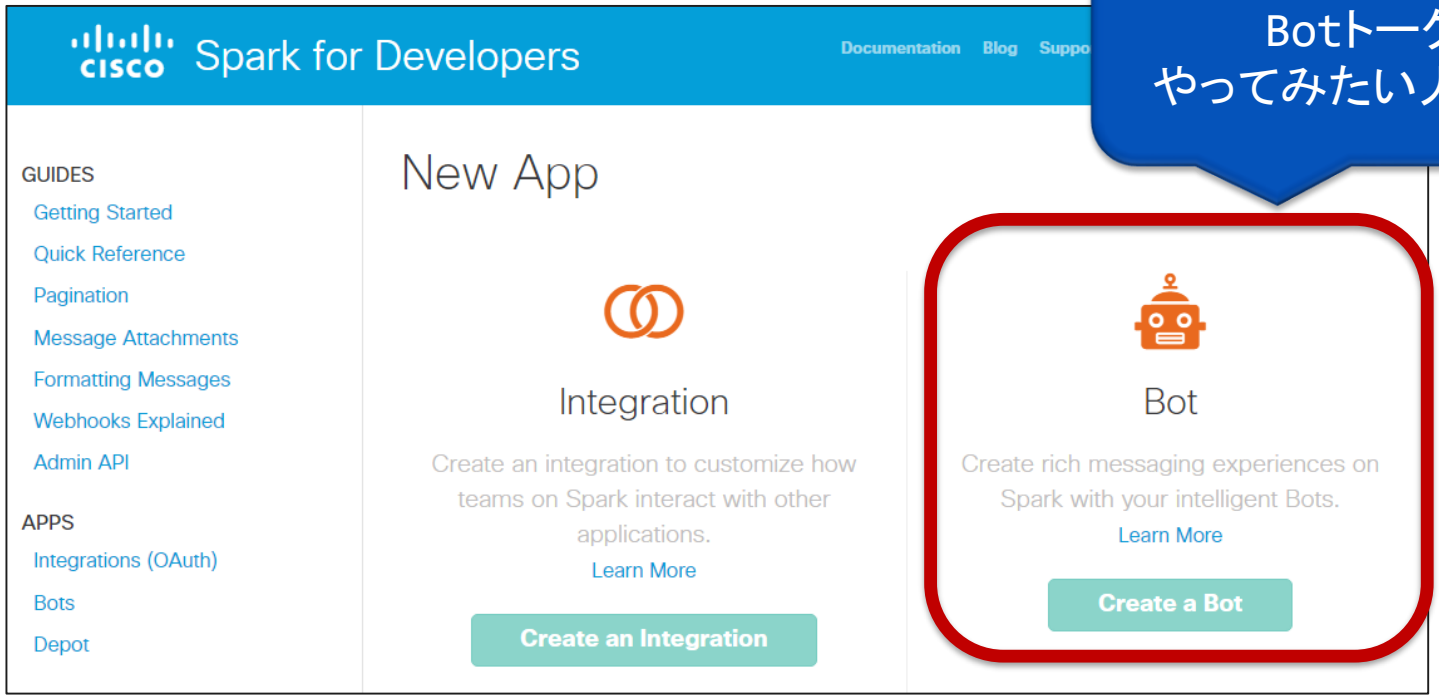


アプリケーション用のトークン



Botアカウントの
作成は簡単！！
Bot用の
トークンも取得できます

アプリケーション用のトークン



ハンズオンを
Botトークンで
やってみたい人は要相談

Botアカウント

トークンを保護しよう

トークンは流出しないよう適切に保護しましょう

例

トークンが保存される設定ファイルへの
アクセス権限を適切に
（Linuxの場合、パーミッションを“600”にするなど ）

トークンは暗号化して保存を推奨

verify = False?

```
res = requests.post('https://example.com/', … , verify = False)
```

verify = False?

```
res = requests.post('https://example.com/', … , verify = False)
```

本来は、このような
コードを書いてはいけません

verify = False?

```
res = requests.post('https://example.com/', … , verify = False)
```

本来は、このような
コードを書いてはいけません

正しく理解した上で、
意図的にやっている場合にのみ
(したくないけど、)許容される

verify = False?

```
res = requests.post('https://example.com/', … , verify = False)
```

TLSの証明書検証などを
無効にするって意味？

verify = False?

```
res = requests.post('https://example.com/', … , verify = False)
```

TLSの証明書検証などを
無効にする意味？

通信経路上で
認証情報やメッセージ本文中の
重要な情報などを含めて
盗み見られても構わないという宣言です

verify = False?

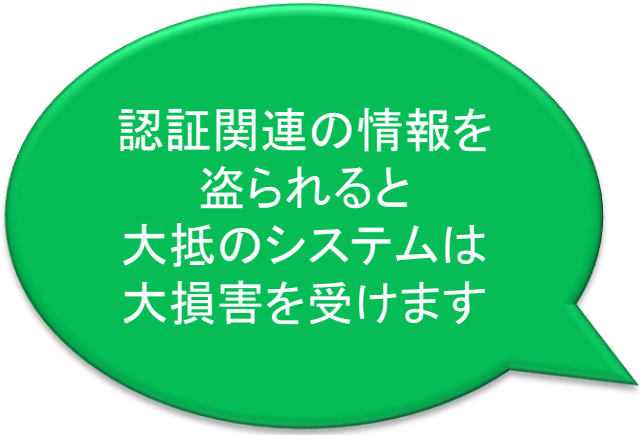
```
res = requests.post('https://example.com/', … , verify = False)
```

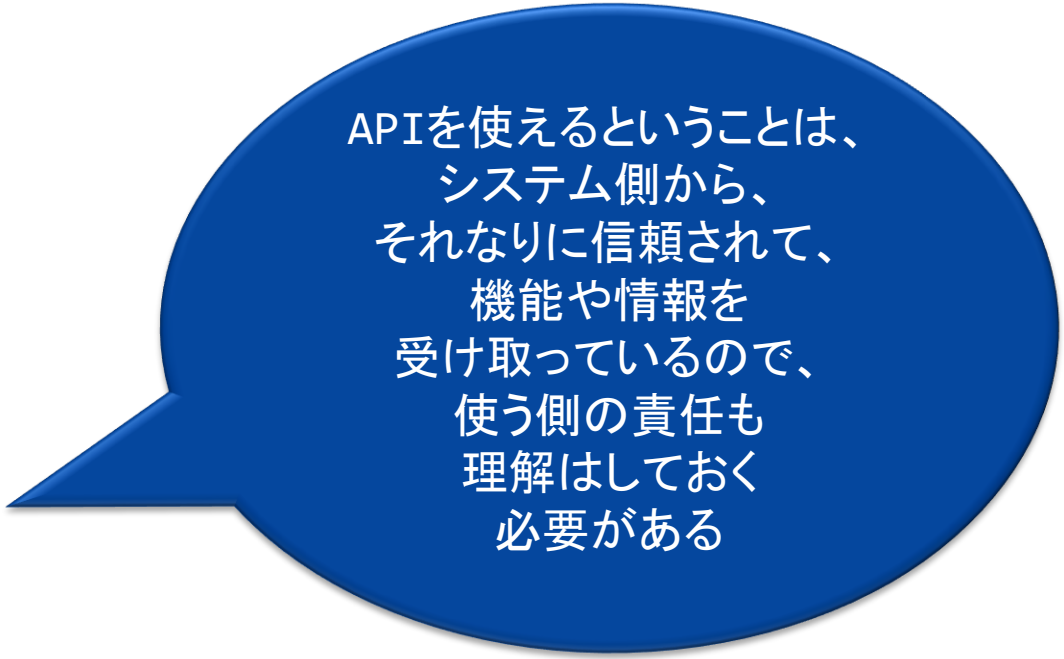
認証関連の情報を
盗られると
大抵のシステムは
大損害を受けます

TLSの証明書検証などを
無効にする意味？

通信経路上で
認証情報やメッセージ本文中の
重要な情報などを含めて
盗み見られても構わないという宣言です

API利用者の責任





```
name = input('Input Your Name: ')
print('Thanks, {0}!!'.format(name))
```

DevNet Express [Tokyo]

March 15, 2017 • Tokyo, Japan

cisco DevNet

developer.cisco.com