In-Process Clinical Intelligence (IPCI)紹介資料

群馬大学医学部附属病院システム統合センター
鳥飼　幸太

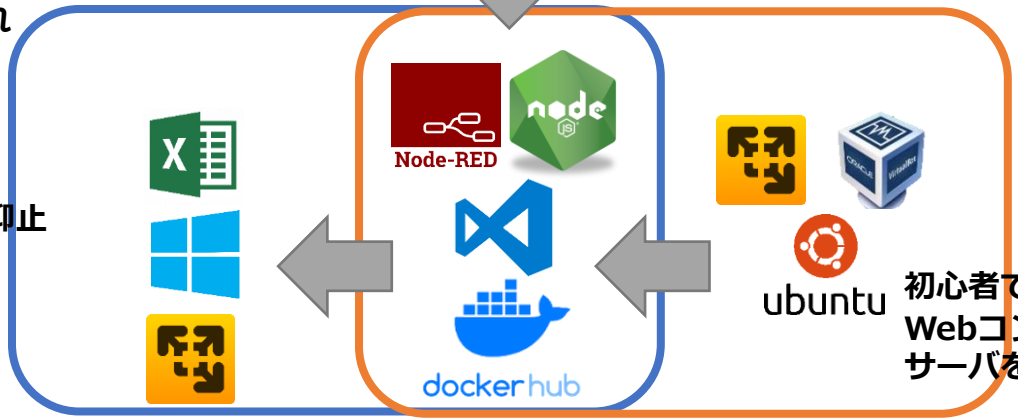
IPCIの実体は
「Ubuntu仮想マシン」です

考察：永続化対象と利用ユーザ維持

Androidは現時点では必ずしも成功した
開発プラットフォームとは言えない

Linux上で「ツールとして」Windowsを使う例はまだ少ないが、永続化視点では意味がある
LTS以降の「載せ替え」が課題→dockerまで抽象化？

データよりもExcel関数/UI/VBA資産が
Windowsで大きな位置を占めている？
→仮想化して永続化の流れ



永続的なバージョンの宣言
OS変更時のユーザー離れを抑止

初心者でも理解しやすいGUIの獲得
Webコンピューティングでは多数の
サーバを必要とする→ライセンス費問題

Windowsは常に
端末内でのワンストップを目指す
仮想化の脅威

pip/npmなどのコマンドインストーラが
再び浸透してきた

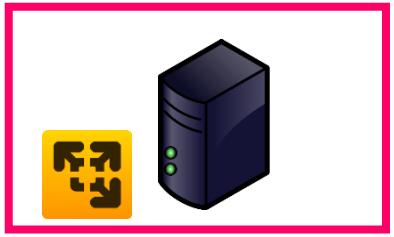
Macはハードウェアと一体の
デバイスとして領域を拡充
「使いやすいUnix」または
「ツール」または
「HTMLアプリの土台」

ECMAScriptはHTML/OSSと相まって
「次世代POSIX」の地位を確立

クラウドによる「囲い込み」
スケーリング用途での拡大
医療情報に適するか要検討

IPCIの実体＝医療情報コンセプトに基づいて設計・実装された仮想マシン

仮想マシン（IPCI）

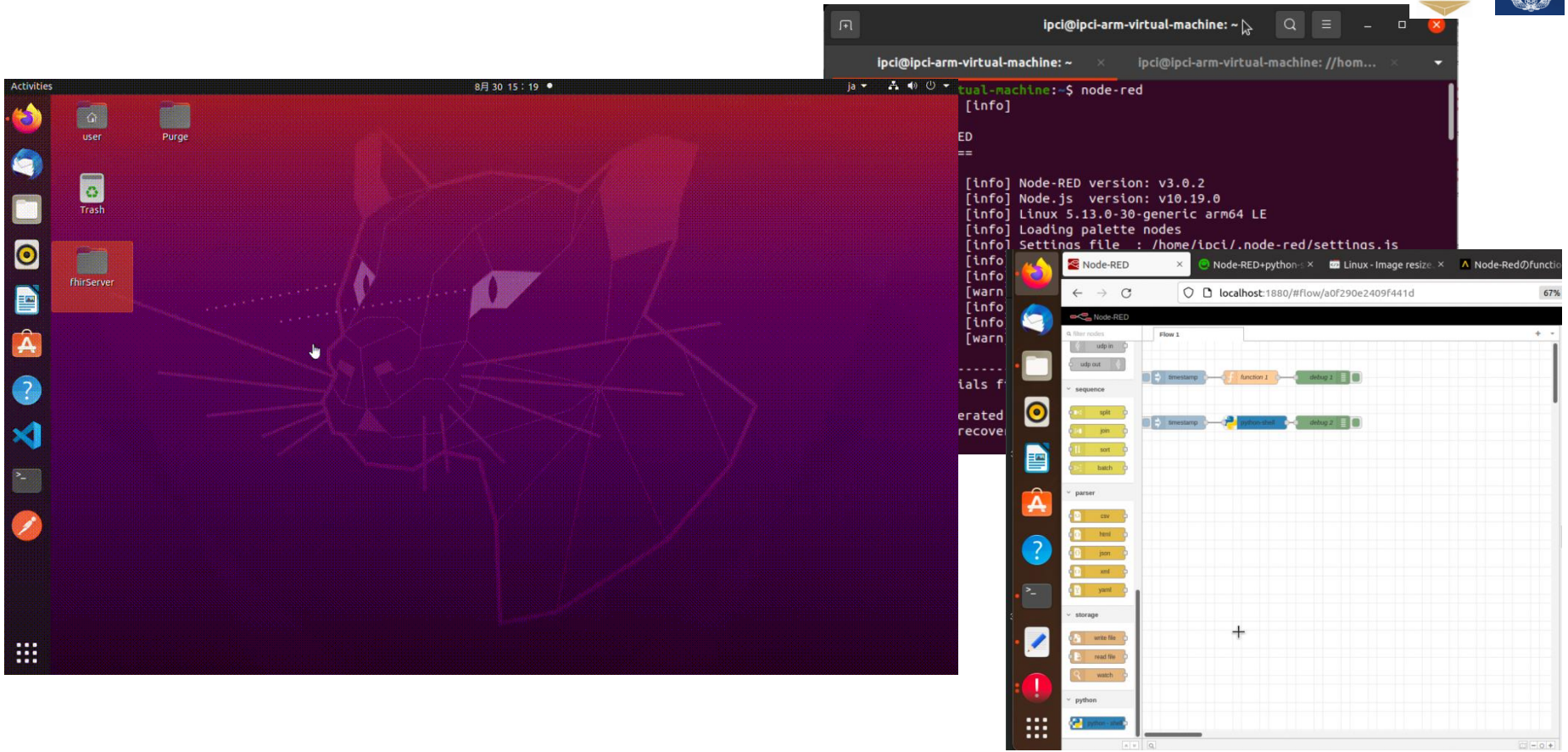


物理的実体（PCやサーバ）

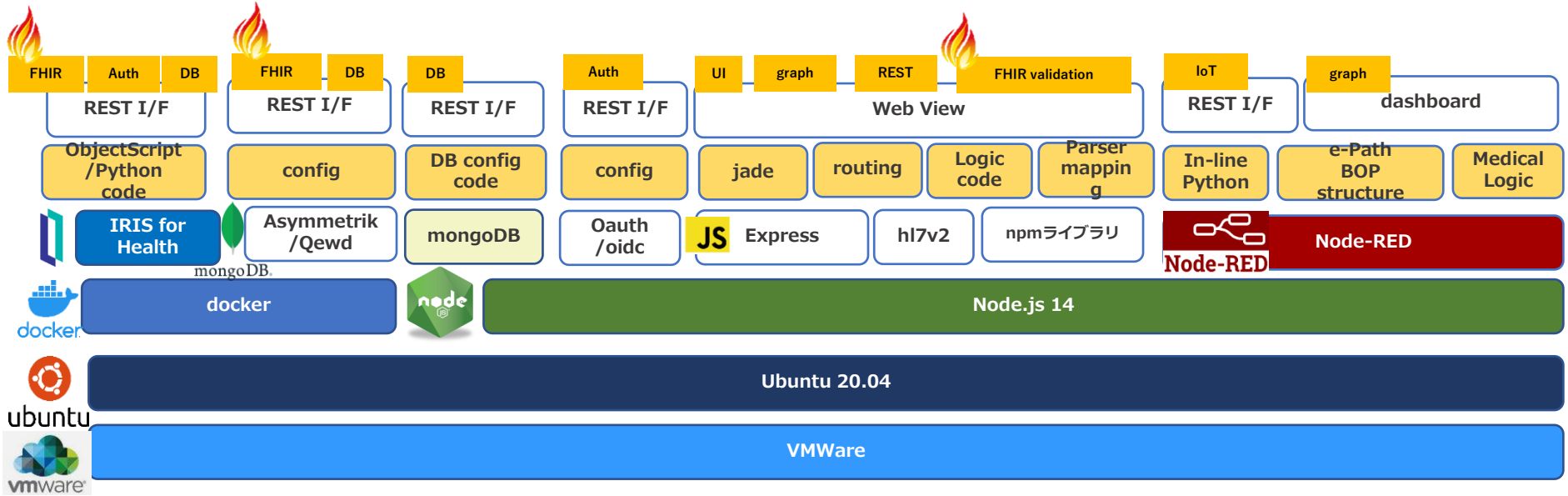
今回実装したものは、
VMware：仮想化プラットフォーム
Ubuntu：OS
Node.js：スクリプティングベース
Docker：コンパイラベース
として、
Node-RED：医療ロジック・Webインターフェース
上に医療アルゴリズムを永続的に蓄積・利用するサーバ

前回チュートリアルでは、要素の一つであるNode.jsサーバの作り方を紹介した
今回チュートリアルでは、HL7 FHIRの送受信・変換サーバとしての使い方を紹介する

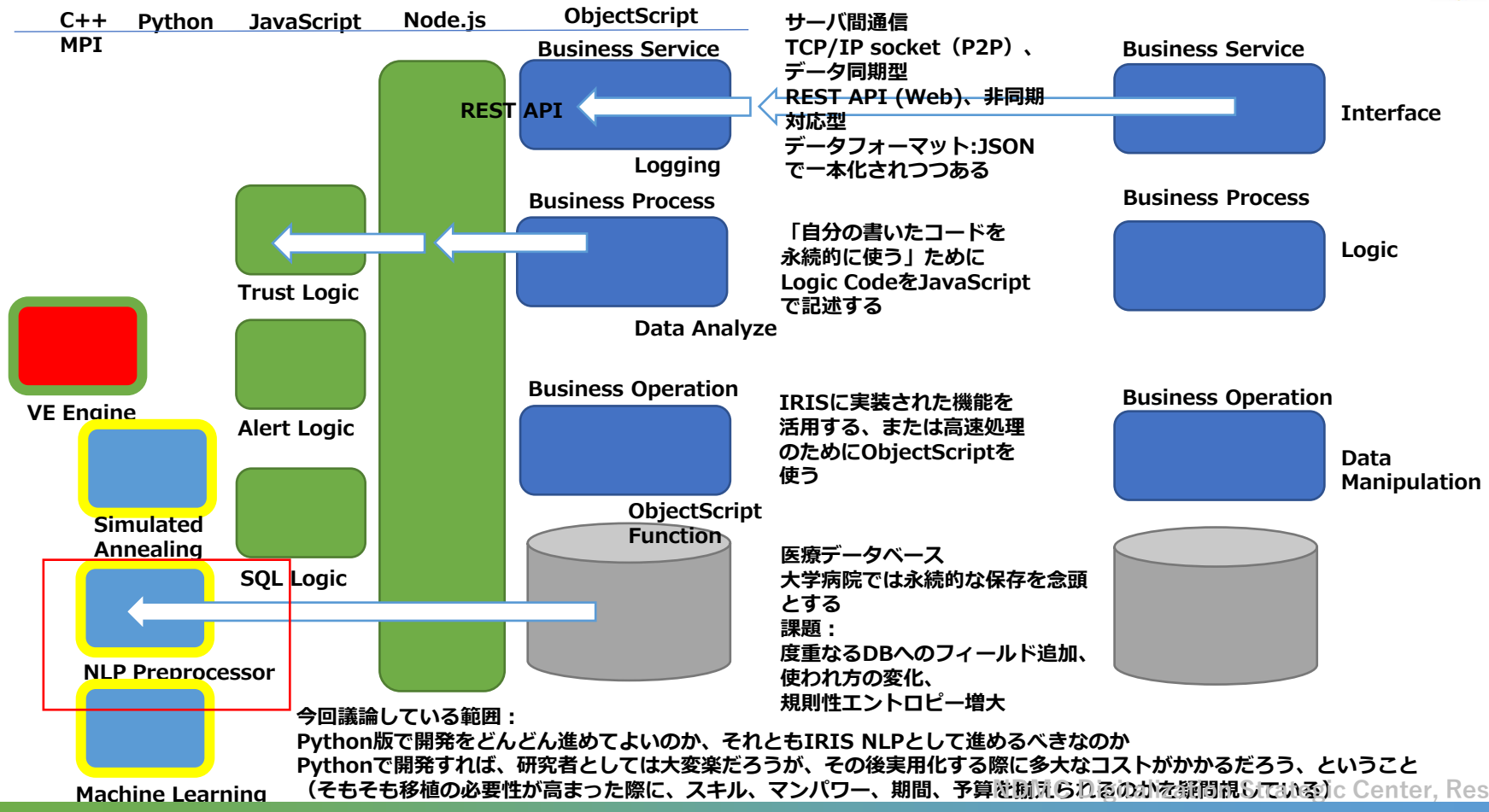
IPCI-FHIRサーバ機能とPatientリソースの起動



IPCIアーキテクチャ

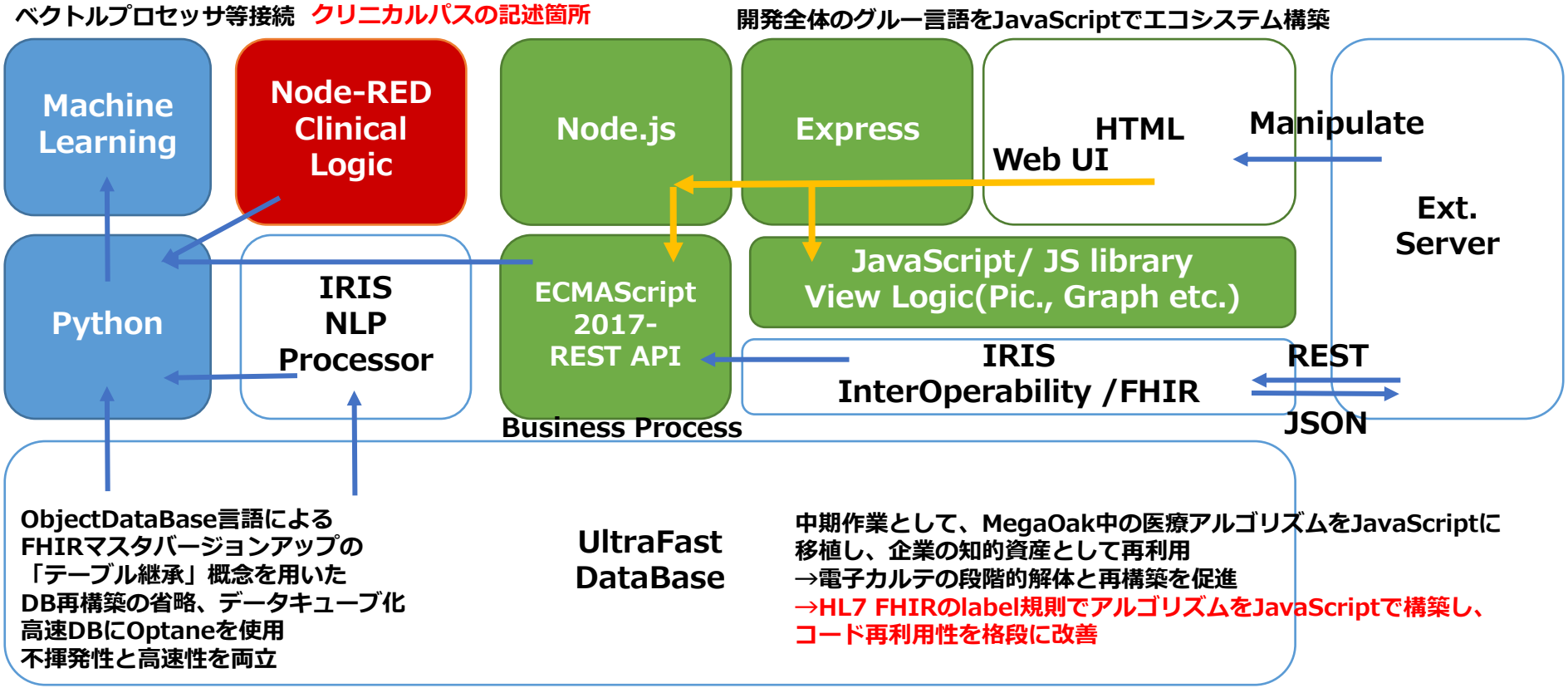


「継続的に開発できる」病院情報システムとは何か



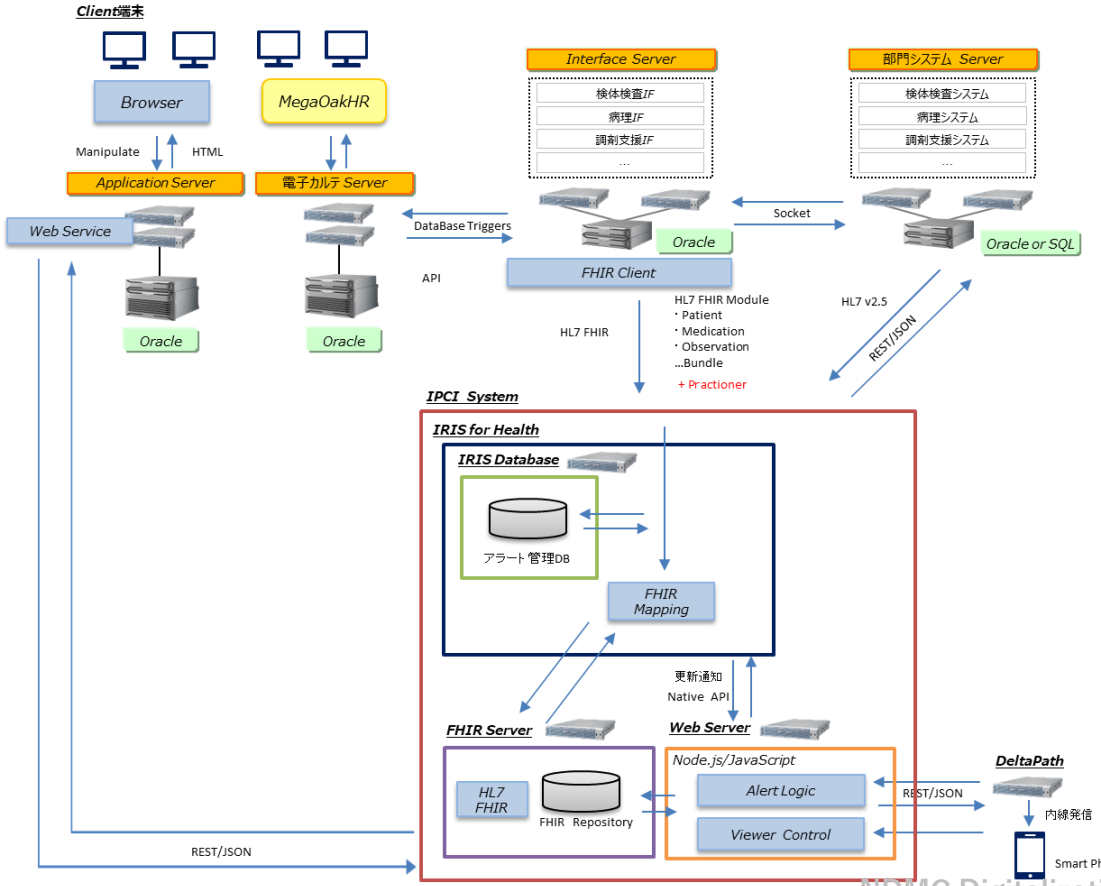
今回議論している範囲：
Python版で開発をどんどん進めてよいのか、それともIRIS NLPとして進めるべきなのか
Pythonで開発すれば、研究者としては大変楽だろうが、その後実用化する際に多大なコストがかかるだろう、ということ
（そもそも移植の必要性が高まった際に、スキル、マンパワー、期間、予算が揃えられるのかを疑問視している）

Medical Informatics/ DX System Framework
病院システムにおける In-Process Clinical Intelligence (IPCI)



IPCIと群大病院情報システム(HIS)との接続

2022.9-群馬大学医学部附属病院内で稼働中

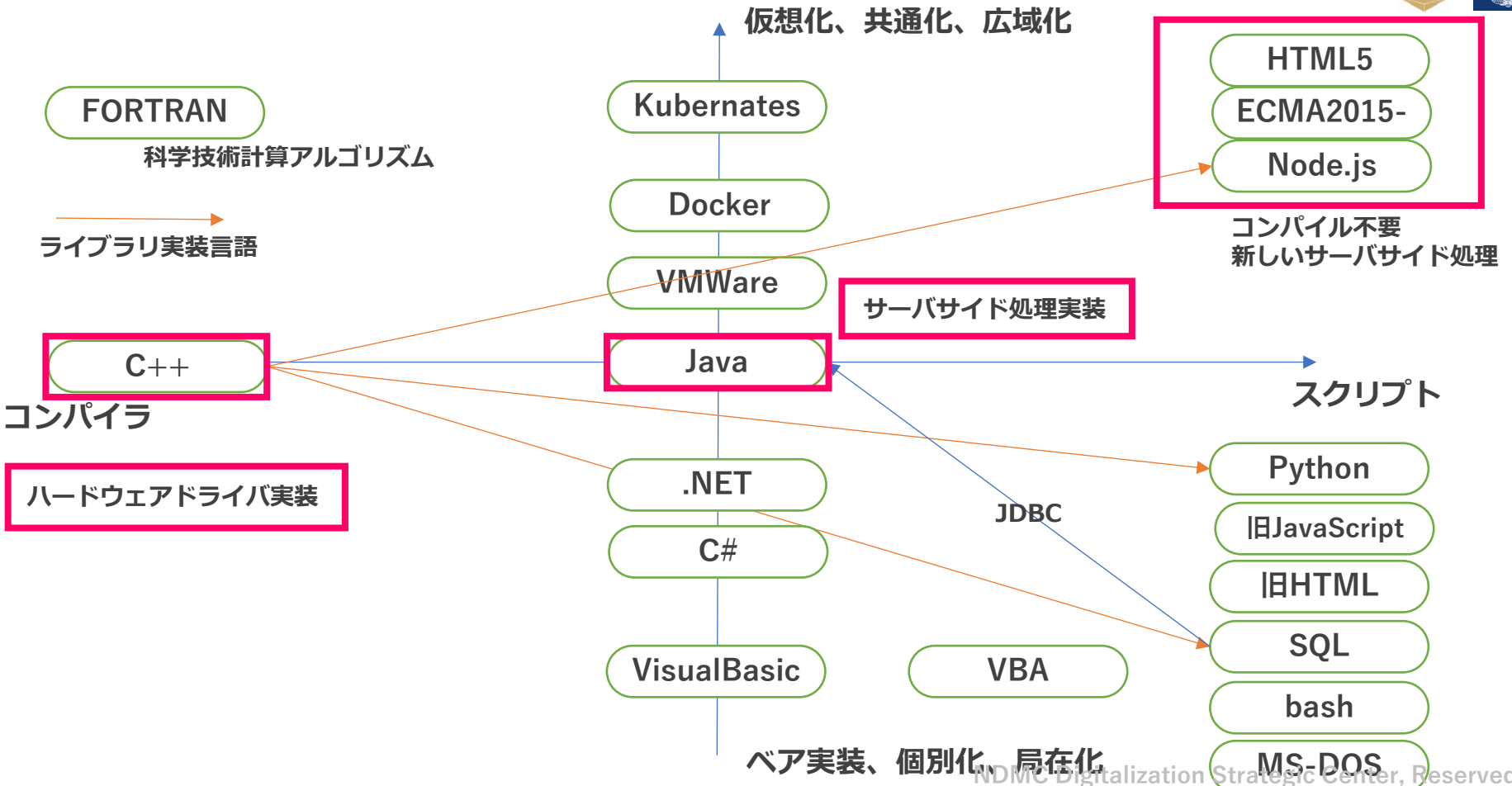


IPCIの特徴

リファクタ性を重視したプラットフォーム


- 「環境構築」自体をオートメーションする→スキルの高いエンジニアの作業結果を複製して恩恵を受ける
- (Linuxのルート環境レベルで独立した)構造として独立することで、機能実装する際の相互関連性を減らす
- OS依存のプログラム（＝マシン語に近いコンパイラを使う）を使わないようにすること
- マイクロコード思想：単純なプログラムやサービスは、再利用性が高くできる可能性がある
- 「コード作成者自身でないスタッフが、コードを容易に理解でき、迅速に活用できること」
- 文法ができるだけ簡素であること
- ライブラリが使いやすく、利用すると便利になる→エコシステムが成立している

採用言語の検討：仮想化と永続化の可能性から



Web/RESTに即したフレームワーク

Webサーバ言語　フレームワーク

クライアント
デバイス言語

オブジェクト指向言語

高速演算　C++　Java

PHP　Laravel

Ruby

Swift

HTML

Kotlin

プロトタイプベースオブジェクト指向言語

JavaScript　Node.js　Express　JavaScript

Java

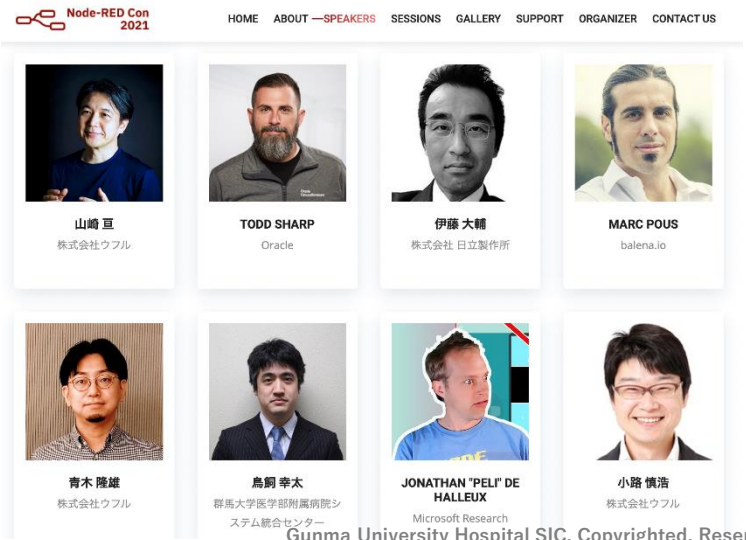
Javaが早期にVM概念を確立していたにも関わらずデバイス言語にならなかったのは
「学習コストの多さ」であると考えられる：
オブジェクト指向言語は細かな定義が行え、コードは十分高速だが、リファクタコストが高い

プロトタイプベースは「RAD的」な使い方と「厳格」な使い方の両方を可能とする

医療ワークフロー永続性とIoTに適したコーディング

- **普遍性の高いアルゴリズムをこそ永続化しようと試みられるべき**
- **条件判断、分岐、中止、繰り返し、変更を記述するのに適した方式が望ましい**
- **コード肥大に伴うコード（クラスやライブラリ）を保守する手間が少ないことが望ましい**
  - **コンパイラがいらない**

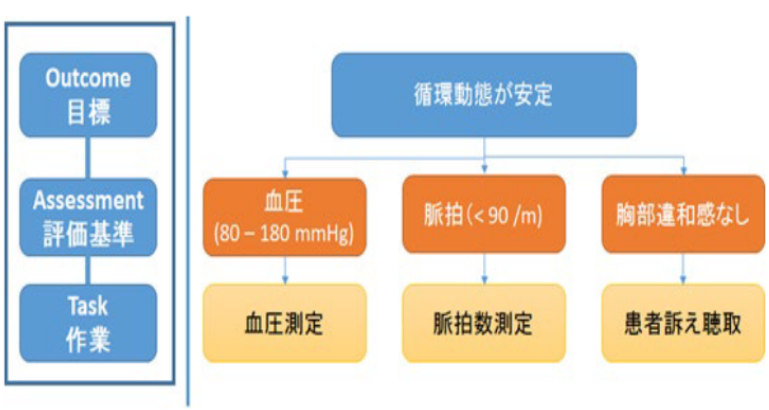
- **IoTが医療行為の起点になるシグナルを発するコードが望ましい**
- **ドライバ等を通じた接続性をサポートしている**

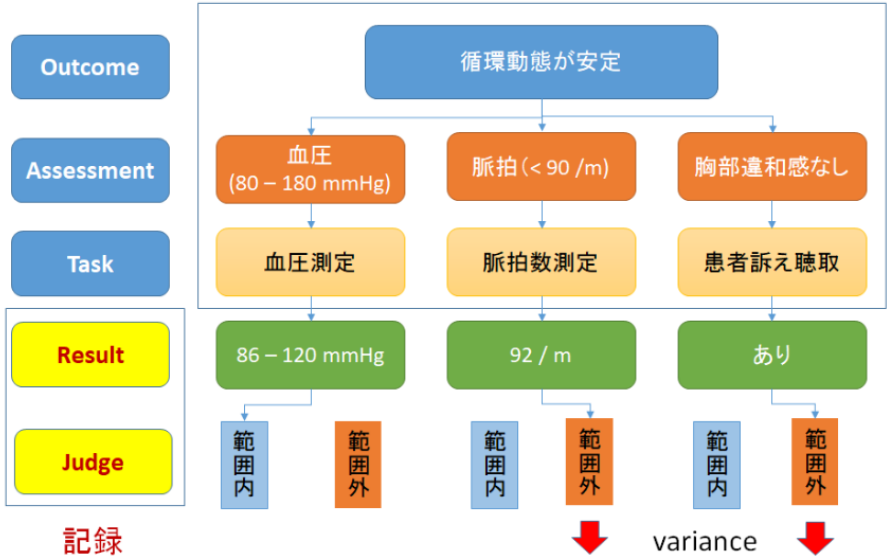


e-Pathが提唱するアウトカムユニット構造

REST APIを

アウトカムユニットの構造と記録の例

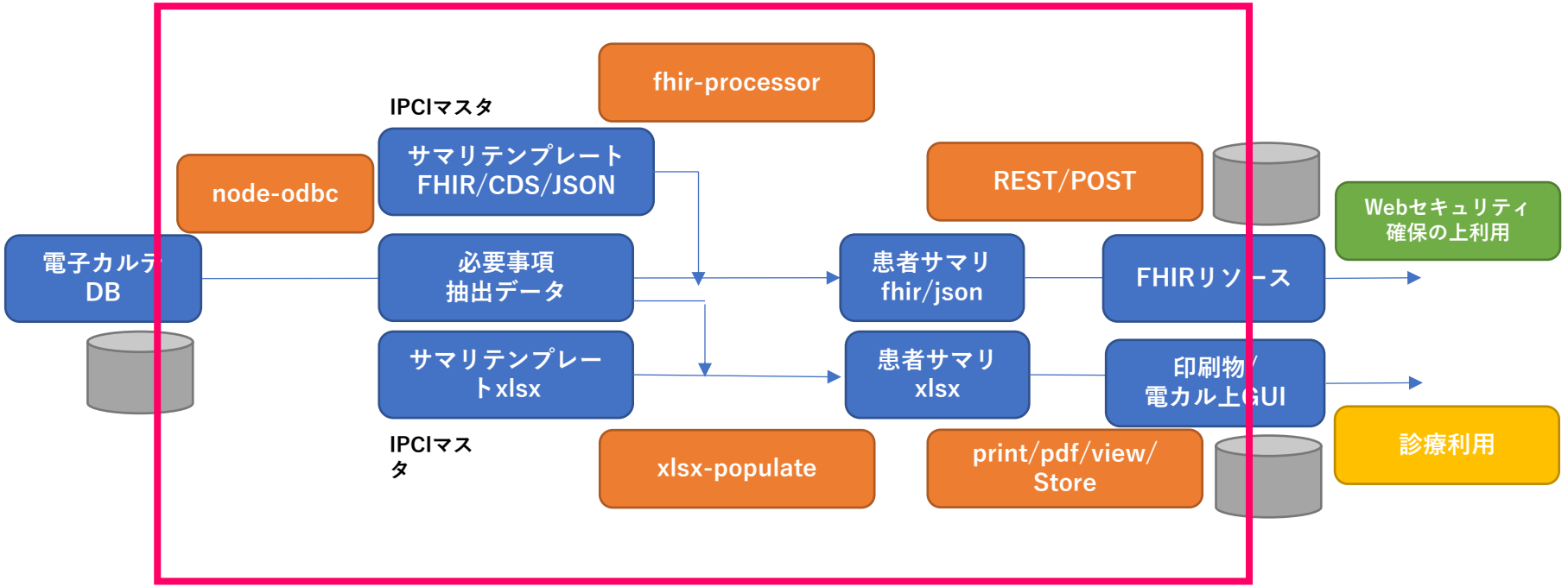




https://e-path.jp/img/prjt/outcum_unit.pdf

診療録サマリを生成するプロセスをIPCI機能で表現する

- Node-REDのフローに表現する流れ



プロトタイプオブジェクト指向の活用

Node-RED

function1

各ノード自身で独立して動かせる
切り離したまま置いていてもエラーにならない
内部的に固有のIDを持つ
ノード自身は内部的にNode.jsのJSONデータとなっている（非常に重要）
→テキストデータによってノードの移植が可能

msgオブジェクトの発生

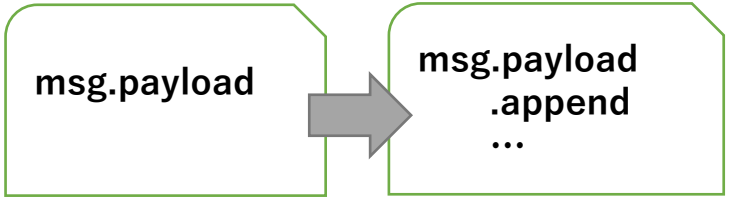
→ start　function1　function2

JSの機能でプロパティを動的に付与

```
msg.append = msg.payload;
return msg;
```

チームコーディング上の課題：
functionの記述によってはmsgオブジェクトの内容をクリアできるので、msgのオブジェクト名と保持/削除ルールをわかりやすくする必要あり

msgオブジェクト全体

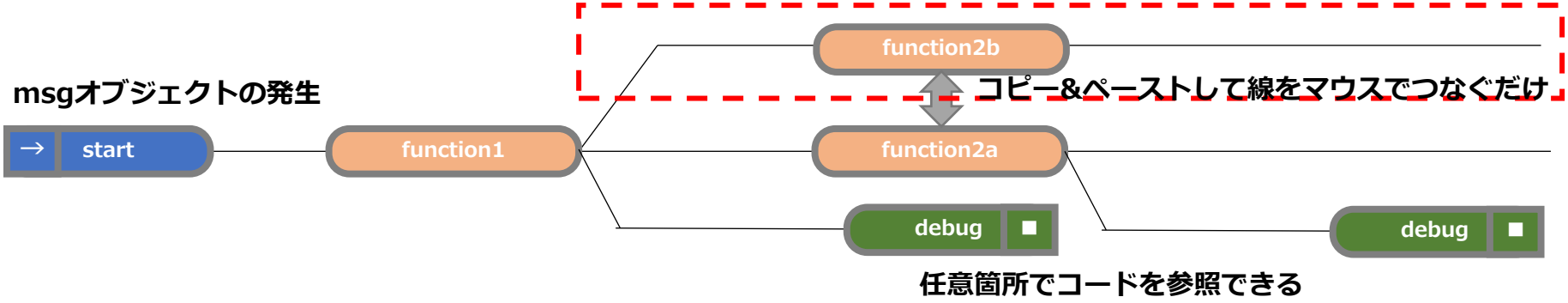


多数の同じデータ処理を行う場合：
Node-REDでシグナル分割するsplit/join機能を使う
msg.aaa.lengthプロパティでの繰り返し
（処理速度が気にならなければどちらで記述してよい）

コードブロックの入れ替えや変更が容易

Node-RED

新しい機能を試したい場合、ほぼGUI上のみで並列ノードを作成して比較できる
（コード記述型だと関連する処理ファイルの移動など含め工数が増える作業）

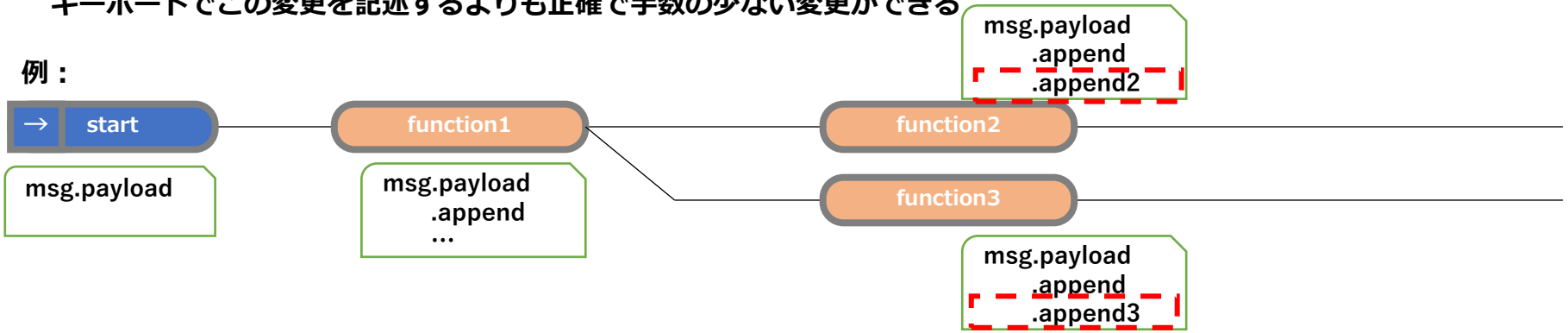


msgオブジェクトを通じたモジュール化が容易

Node-RED

フロープログラムのUIでは、マウスによりコードの実行順や分岐自身を取り扱うと、
キーボードでこの変更を記述するよりも正確で手数の少ない変更ができる

例：

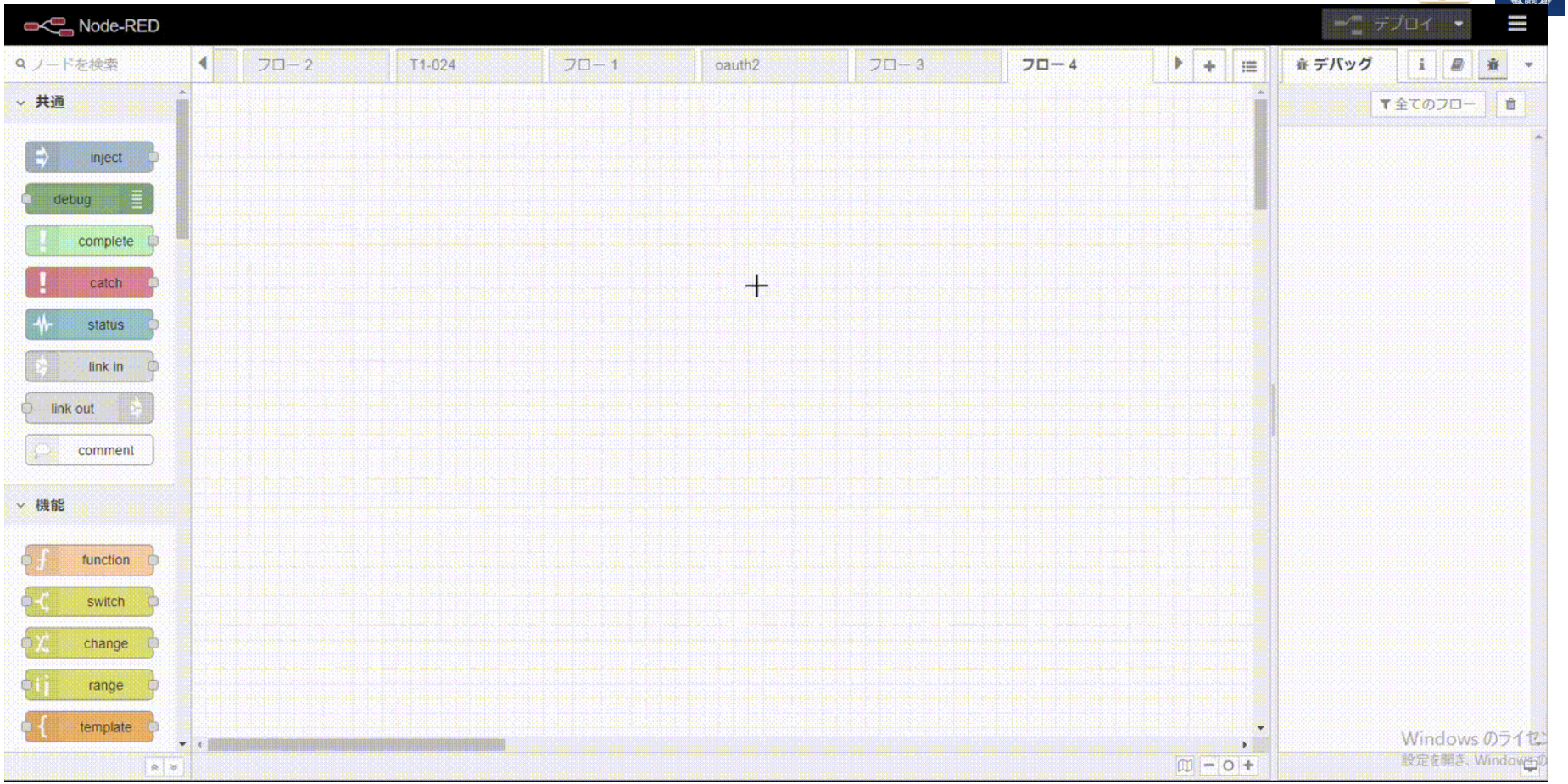


分岐後は独立したオブジェクトとして振舞う

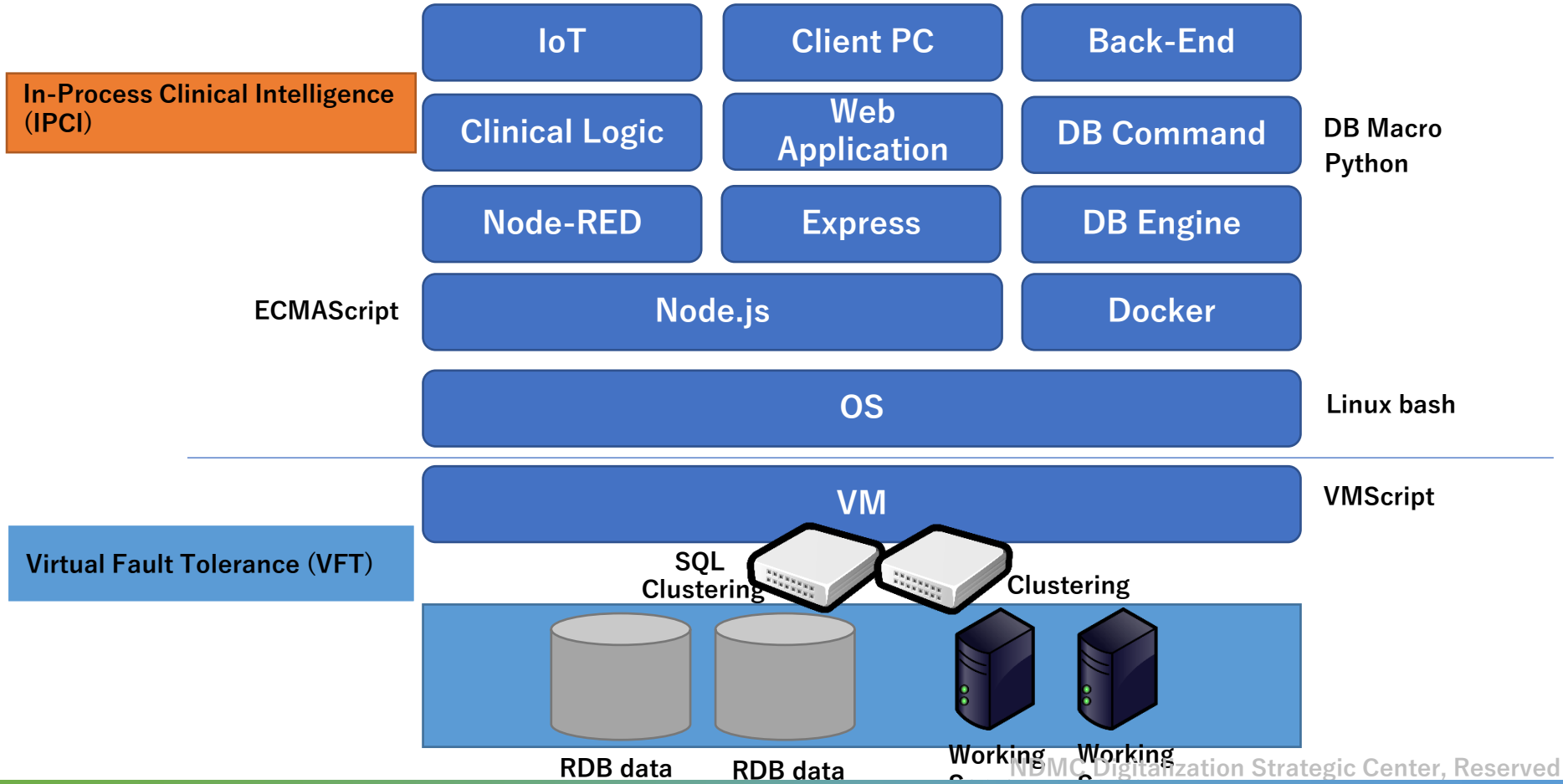
医療ロジック自身が、演算途中のデータからさらに処理を分岐させたい場面が多い
例：検査結果の疾患解釈は診療科によって、また注目する疾患によって「組み合わせ」や「閾値」が異なる
これらを複数解釈させようとすると、フロープログラムの「処理できるデータをマウスで複製できる」利点が活きる

IPCIチュートリアル

Node-REDによる医療ロジックフローのコピー&ペースト



IPCIをコアとしたスケーリング構成（大規模サーバへの応用）



触ってみましょう

作業の流れ

1 VMWareの準備

2 IPCI仮想マシンデータのダウンロード

3 VM起動

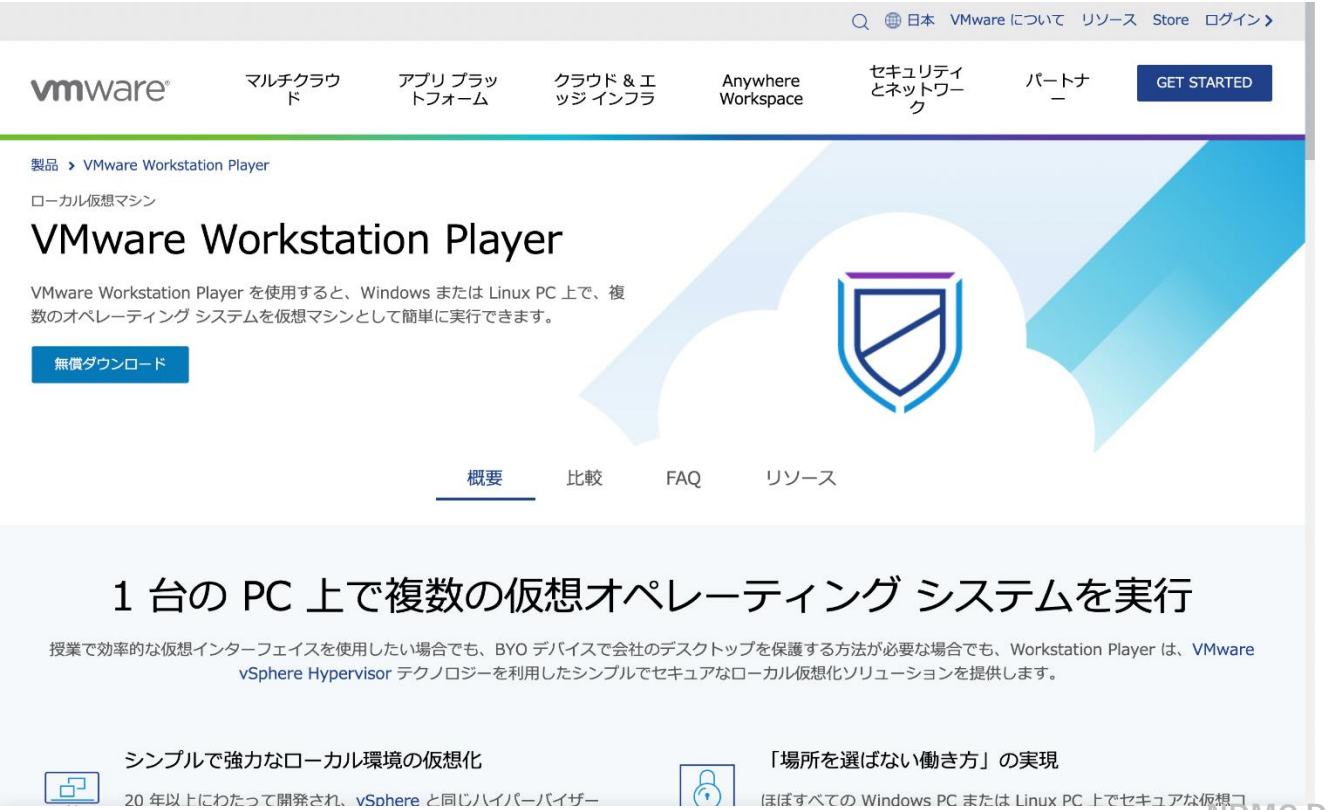
4 Node-RED起動

5 Node-REDフロー作成

6 FHIRサーバ利用

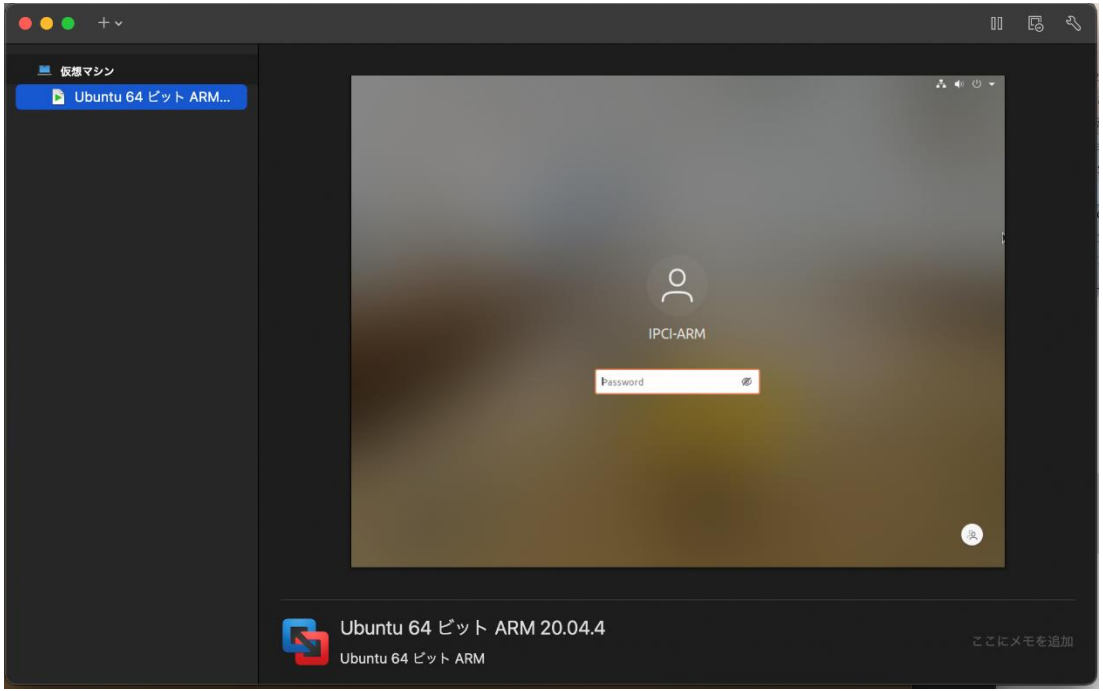
VMWareの準備

https://www.vmware.com/jp/products/workstation-player.html



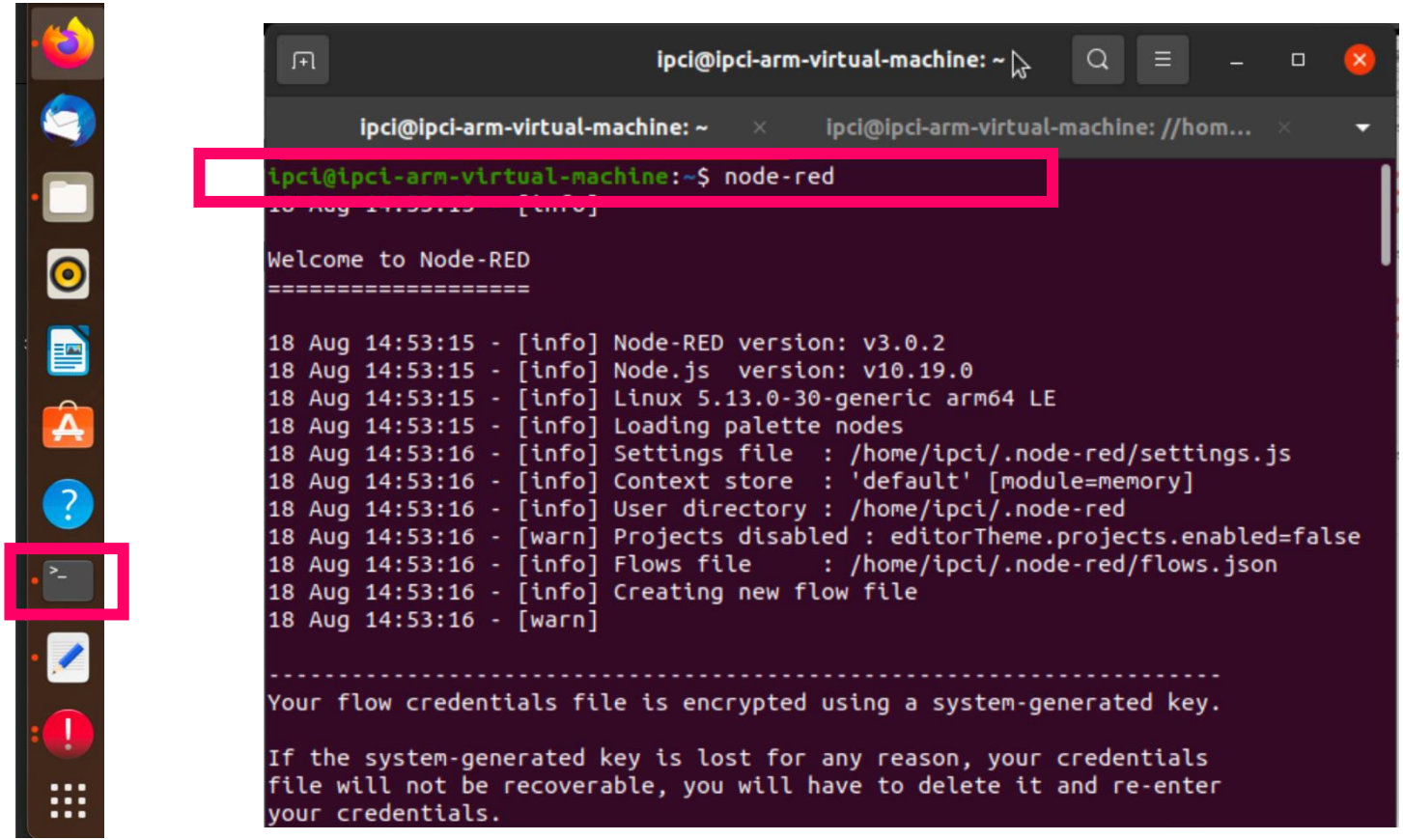
- **Intel CPU 2コア以上**
- **メモリ2GB以上（4GB以上推奨）**
- **ディスク20-30GB**

IPCIサーバ起動法



- **VMWare WorkstationPlayerで起動**

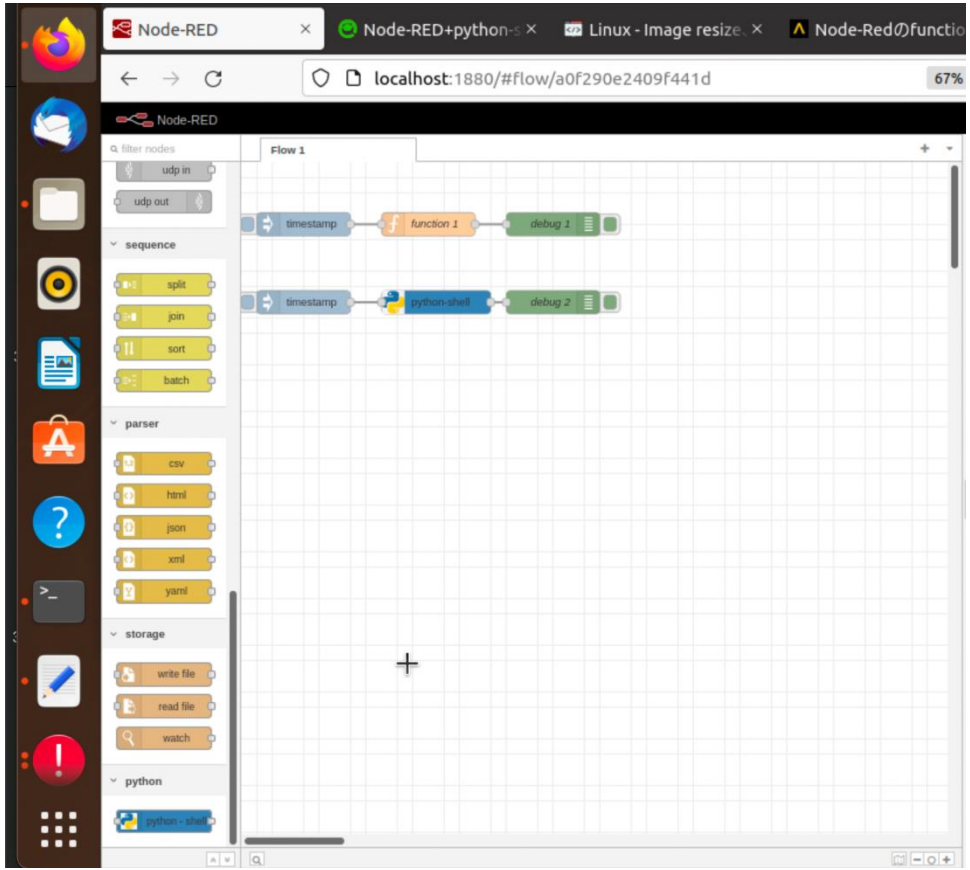
ターミナル（コマンド入力の窓）立ち上げ



起動コマンドは

node-red↩

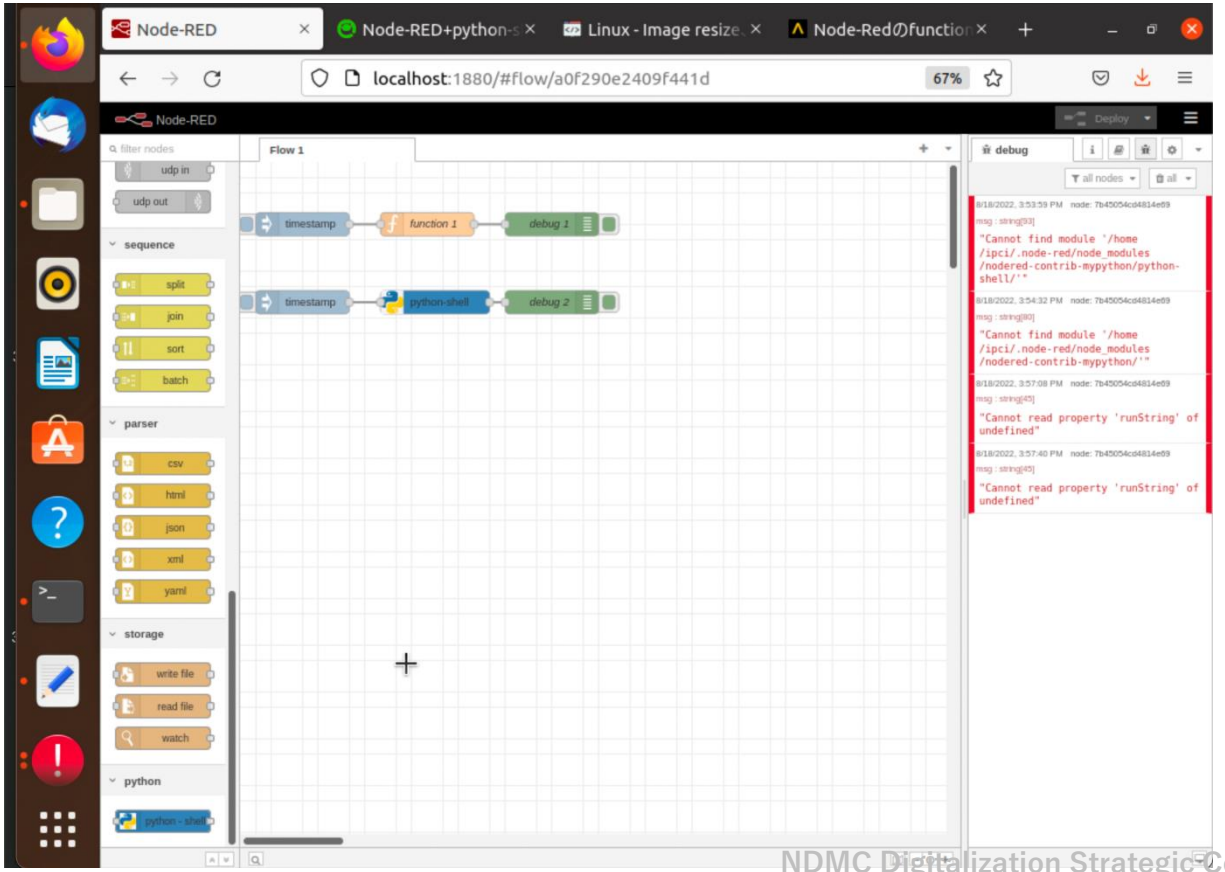
Node-RED起動法



- **Webブラウザ(ここではFireFox)を起動し、URLに”localhost:1880”と入力**

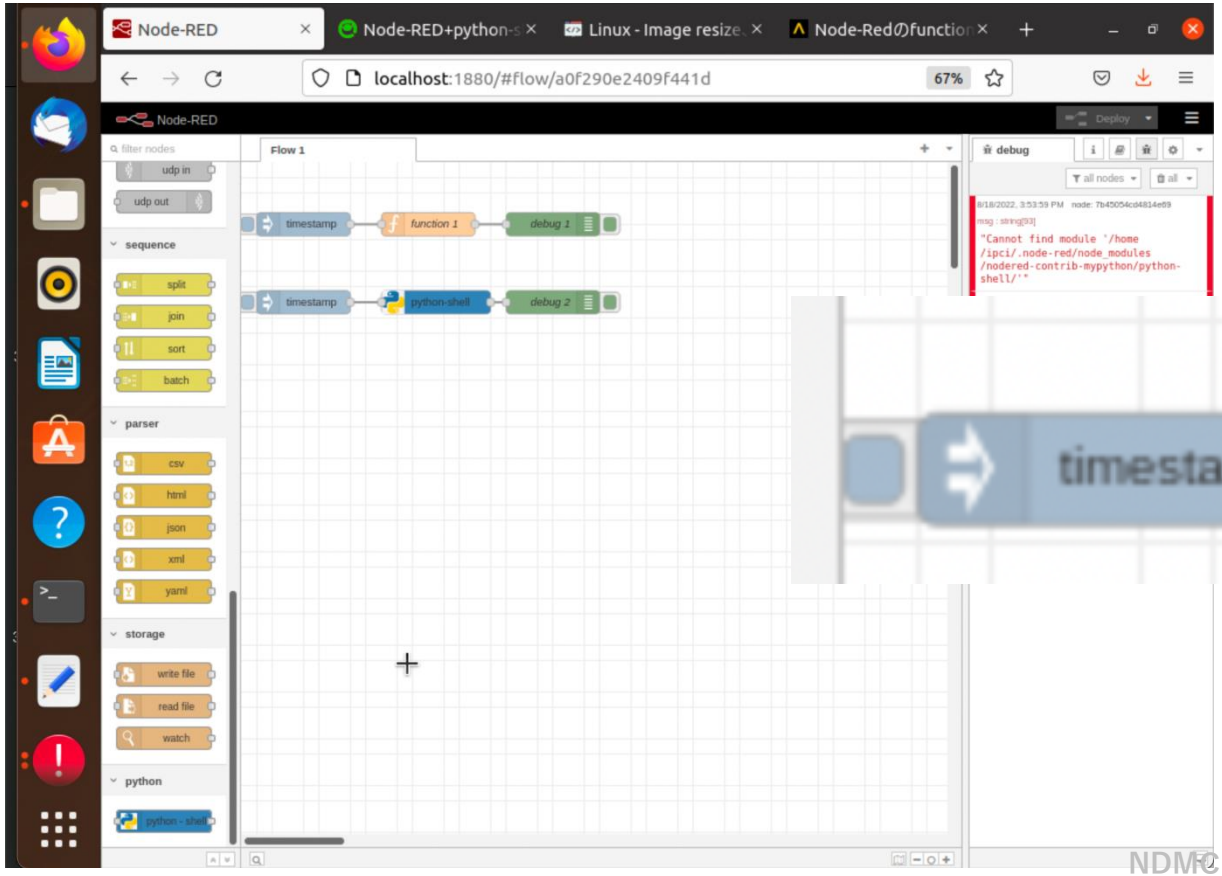
Node-REDコントロールの利用法

- **左側のタブからノードをドラッグ&ドロップする**



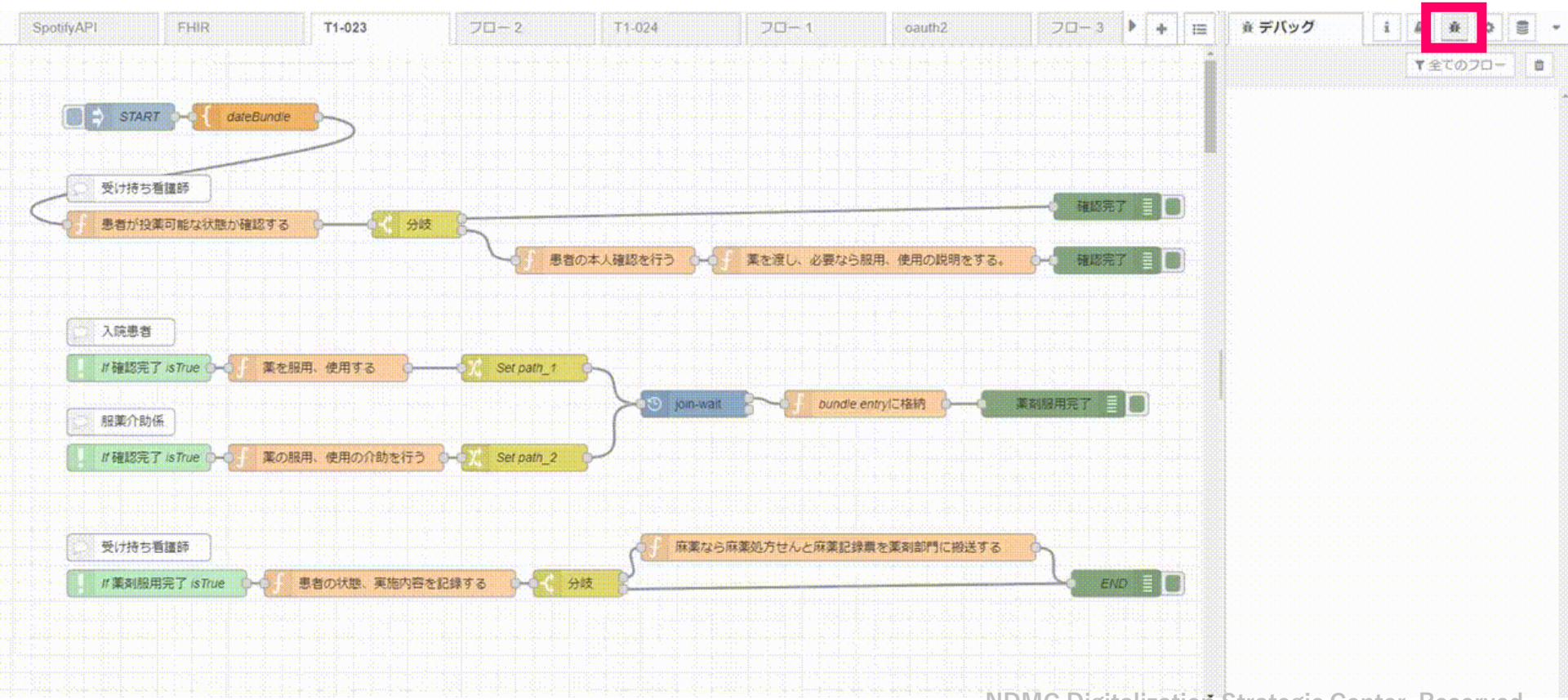
フローの接続

- **マウスでコネクタ部分[・]をドラッグすると配線が発生する**



Node-REDコントロールの利用法

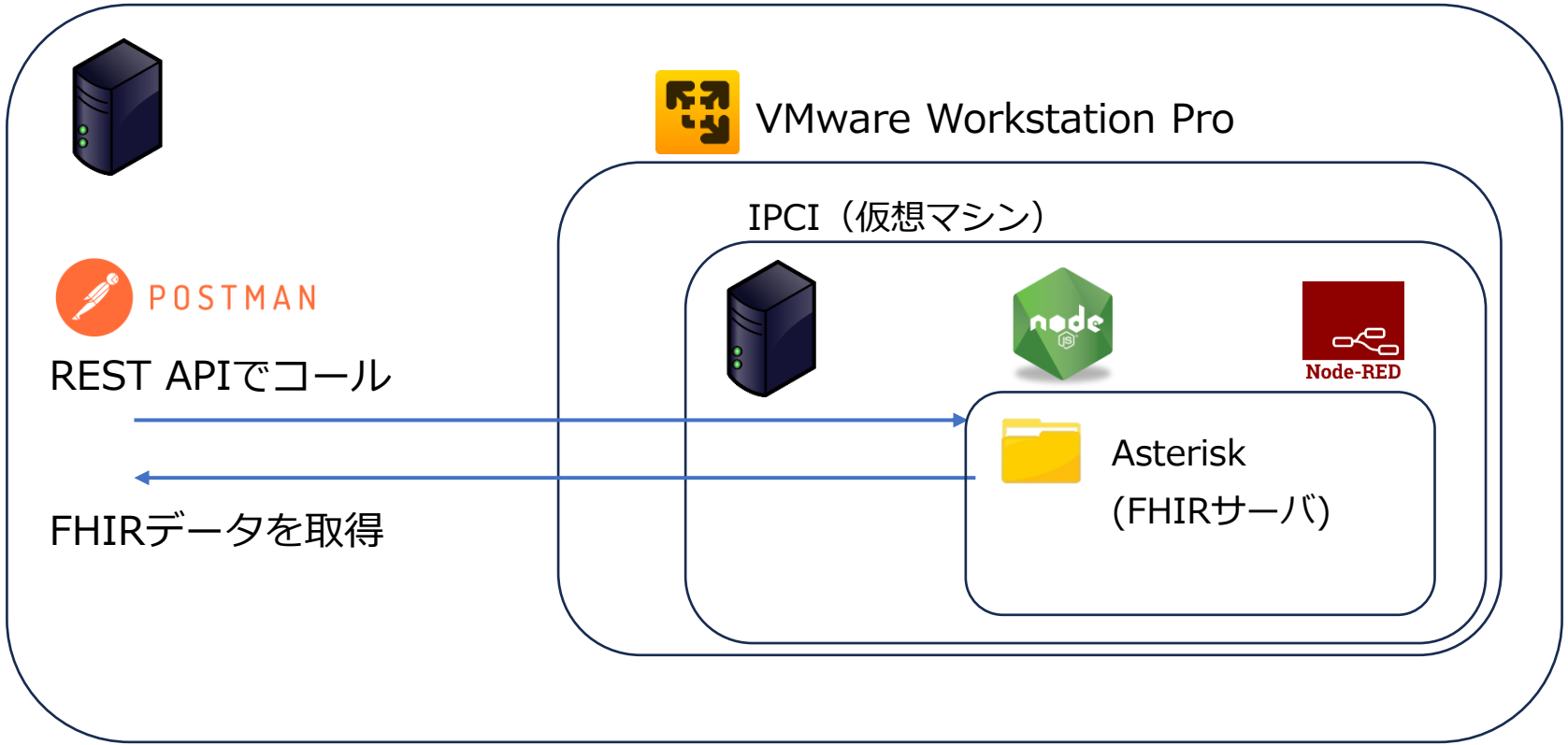
- **“start”の左側をクリックするとフローが実行され、debugボタン押下で出力が表示される**



FHIRサーバとして使用する

セットアップ

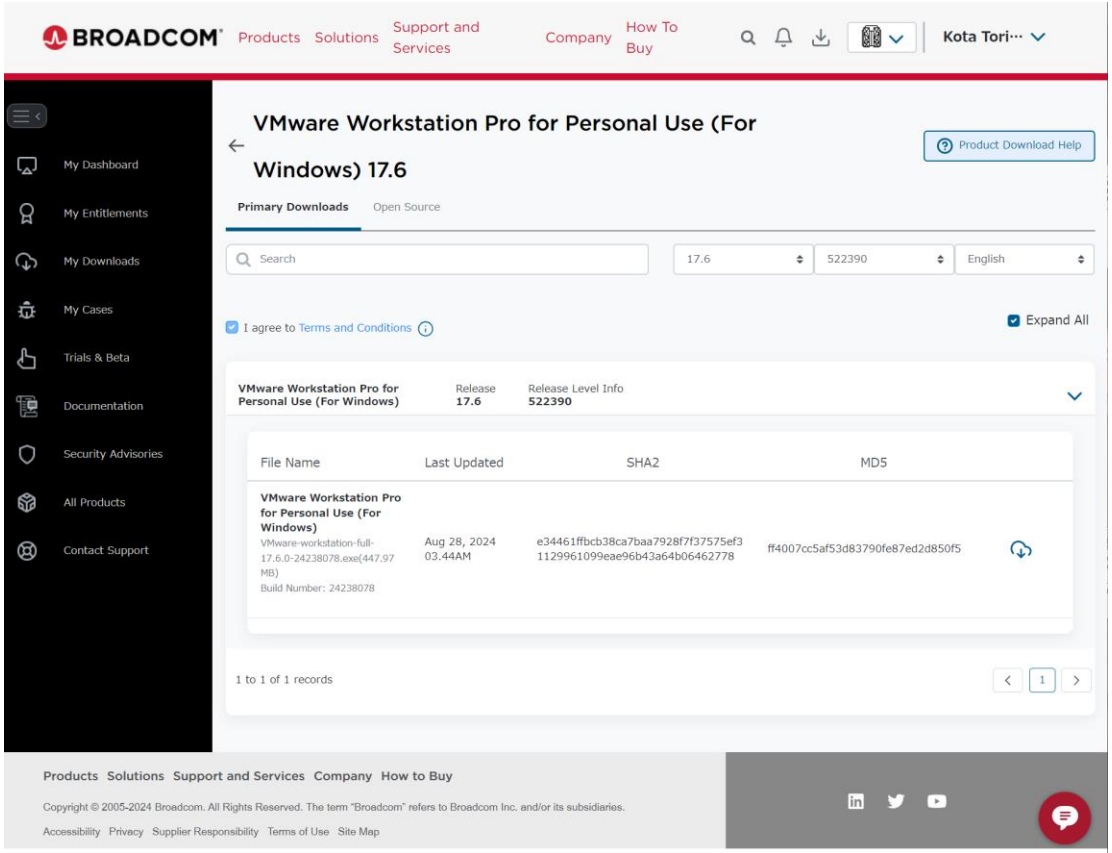
個人PC（各自ご準備ください）



Node.jsでは各プログラムが一つのフォルダの下で動作完結する

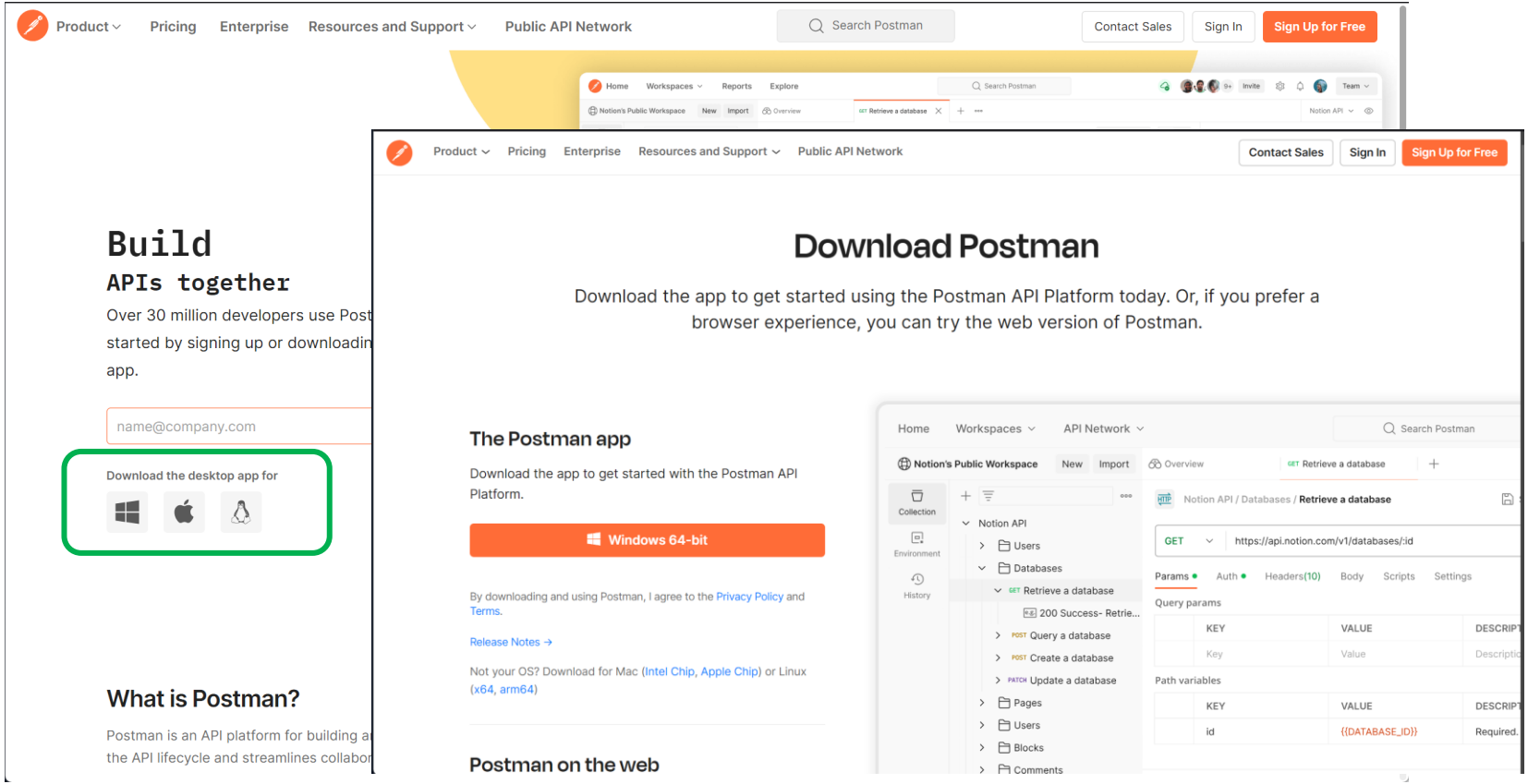
VMware workstation proのインストール

Broadcomのwebページでアカウント作成→Pro 17.6(10/3時点)をダウンロード



Postmanのダウンロード

PostmanのwebページでOSに合わせたソフトウェアをダウンロード



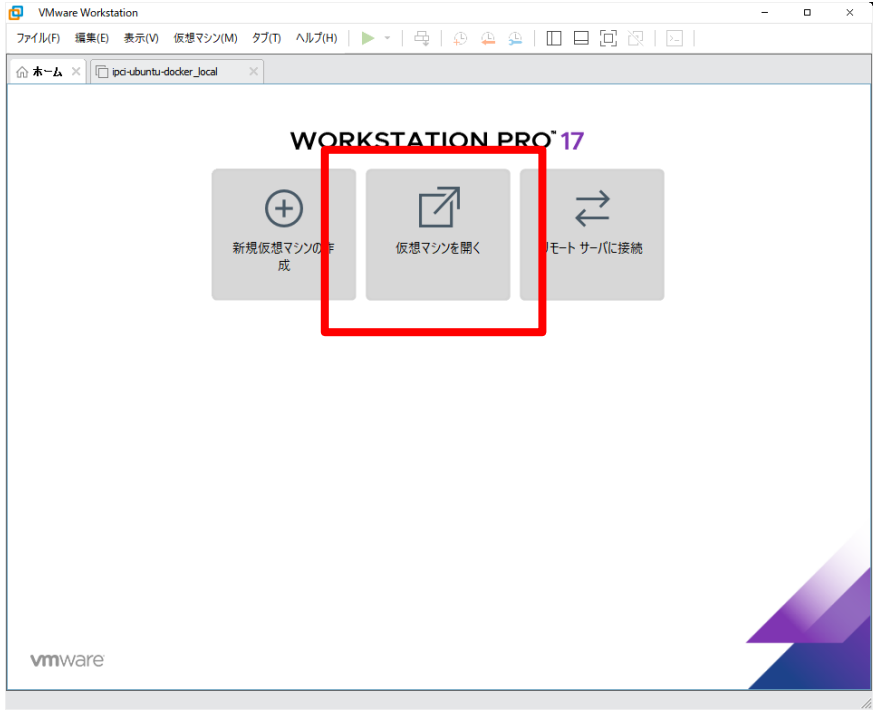
環境準備のためのダウンロード

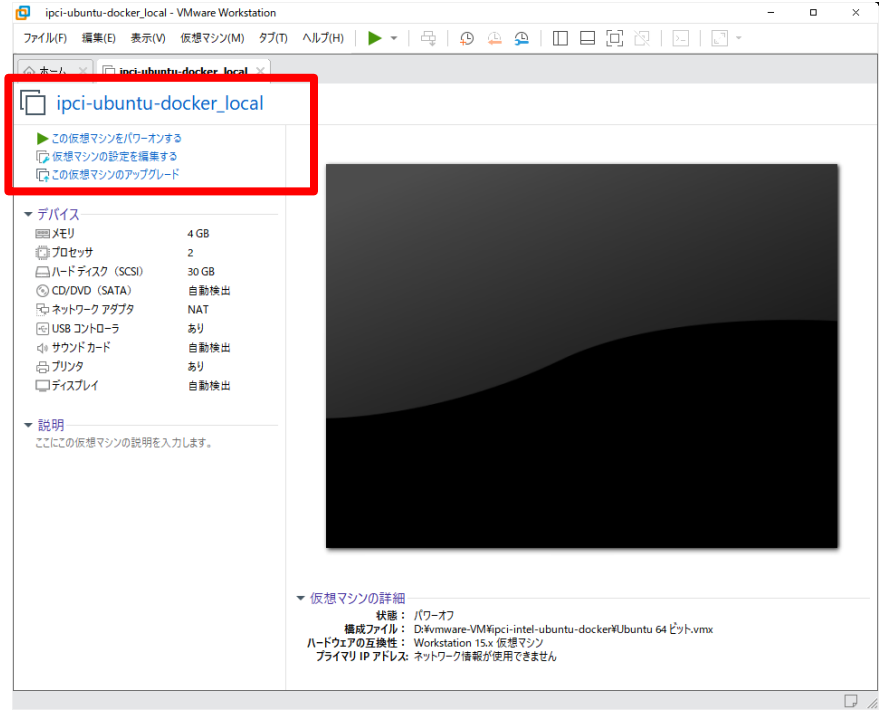
- M学会のチュートリアル専用ページからIPCIの仮想マシンイメージとハンズオン用ファイルをダウンロードします。

https://mta2023handson.z11.web.core.windows.net/files/IPCI.zip

IPCIの仮想マシンの起動

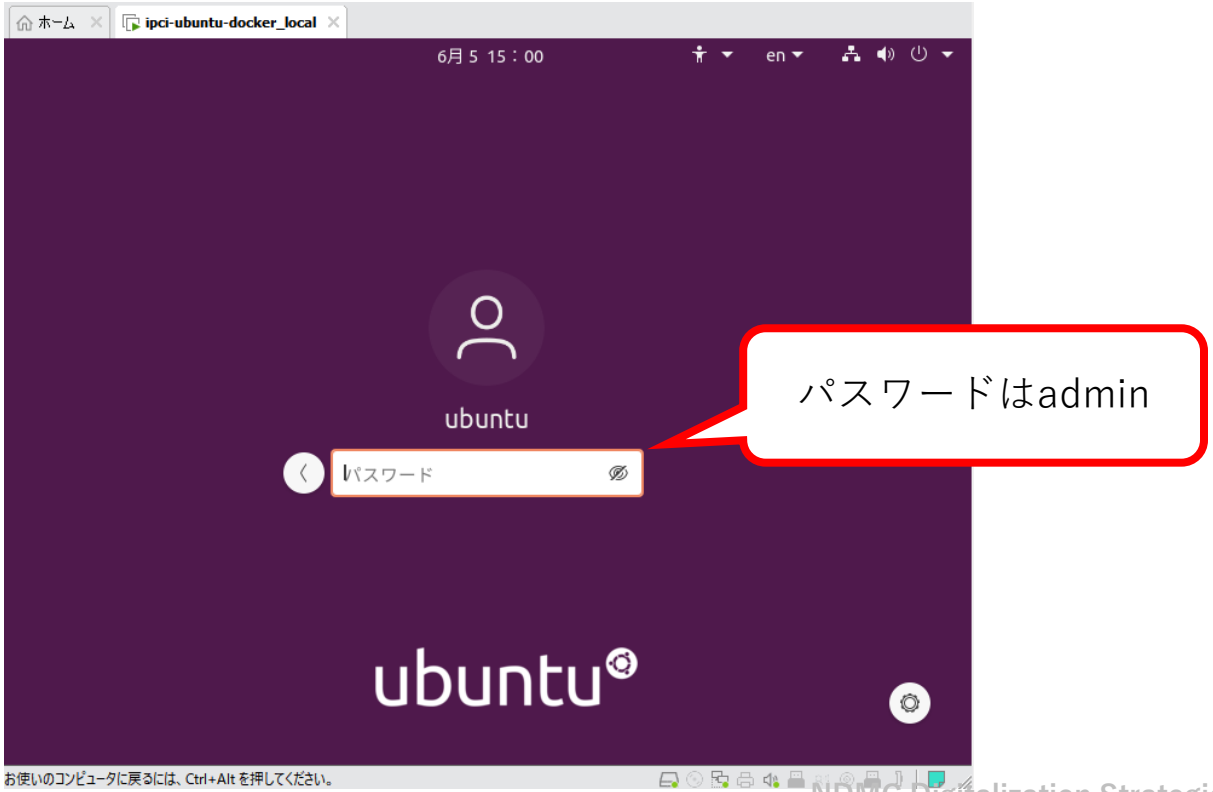
- IPCIの仮想マシンイメージをVMware Workstation Playerから開いて起動します。



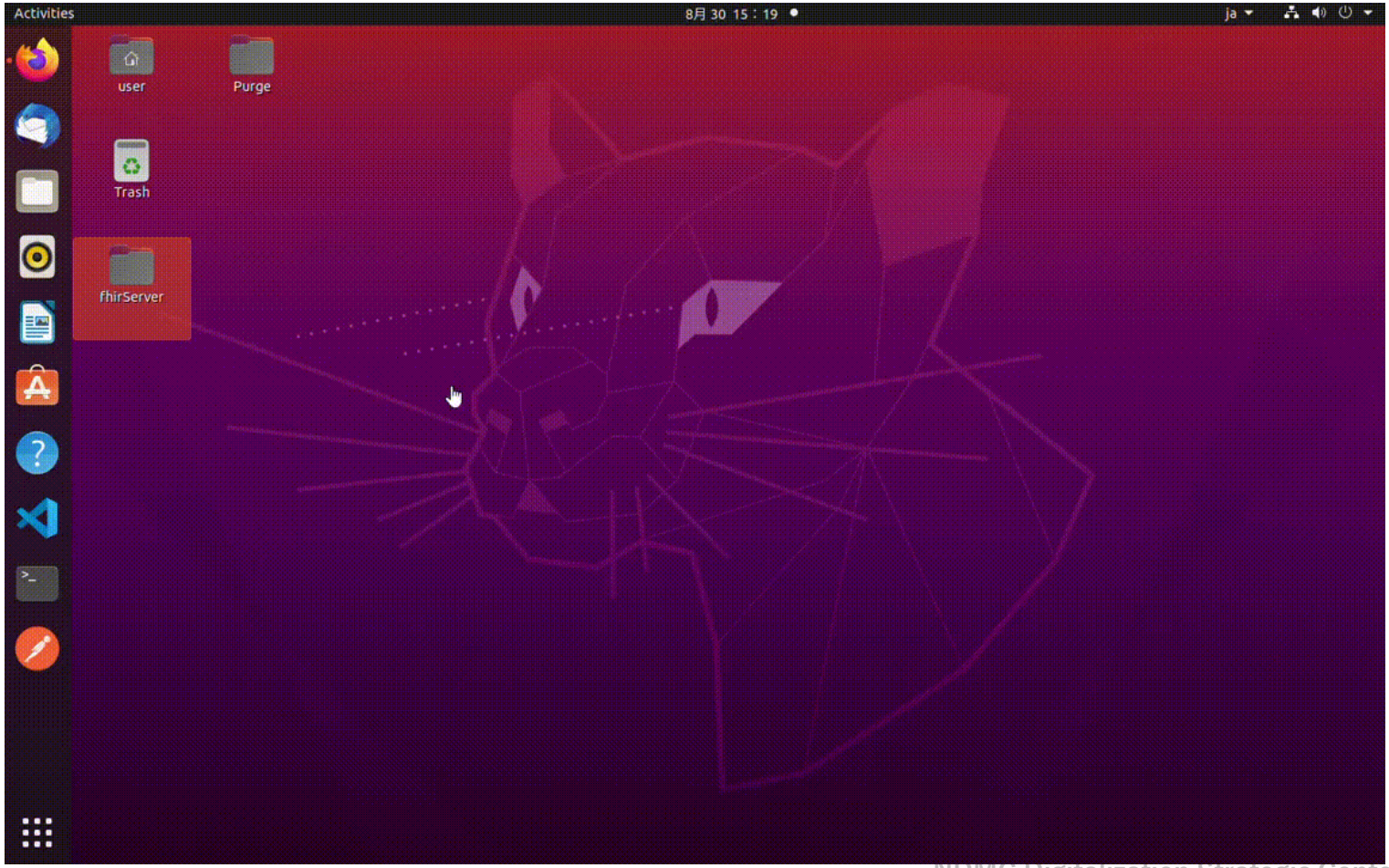


IPCIの仮想マシンのログイン

- IPCIの仮想マシンへログインします。



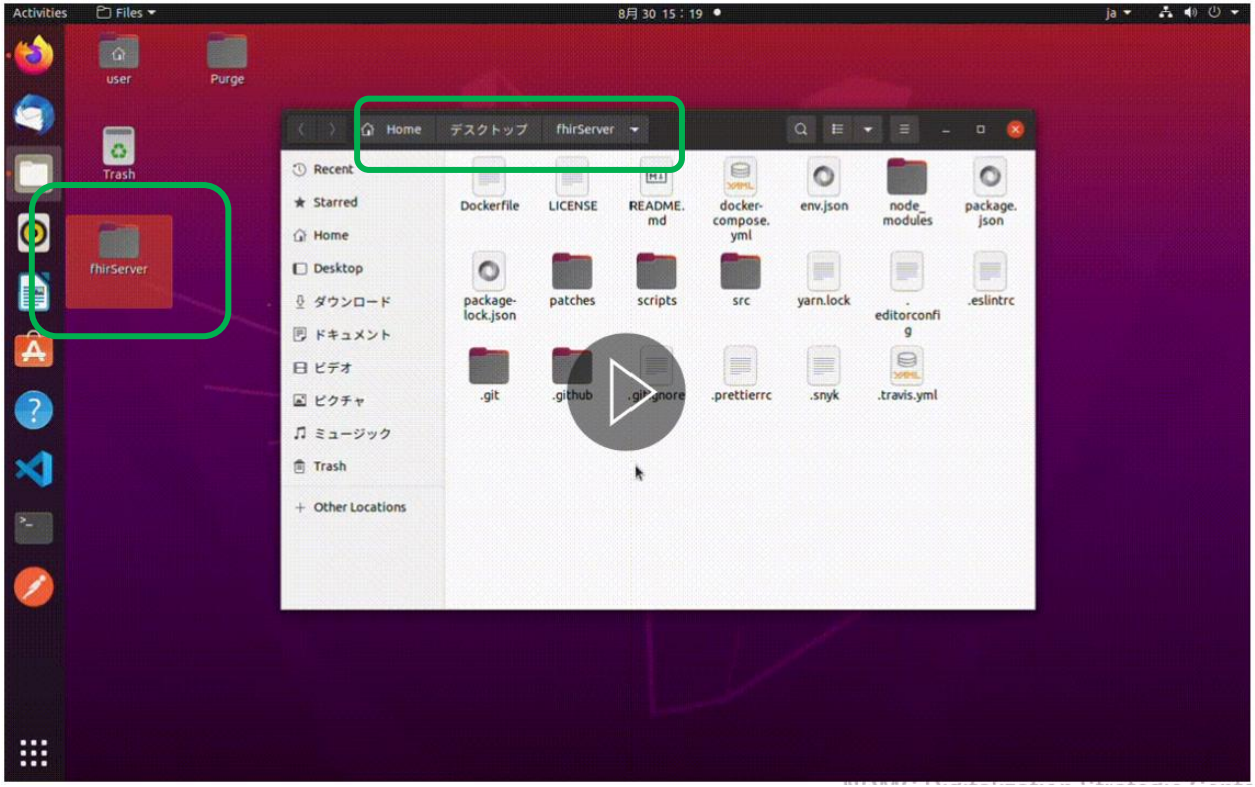
IPCI-FHIRサーバ機能とPatientリソースの起動



FHIRサーバ起動(1)

Asterisk
(FHIRサーバ)

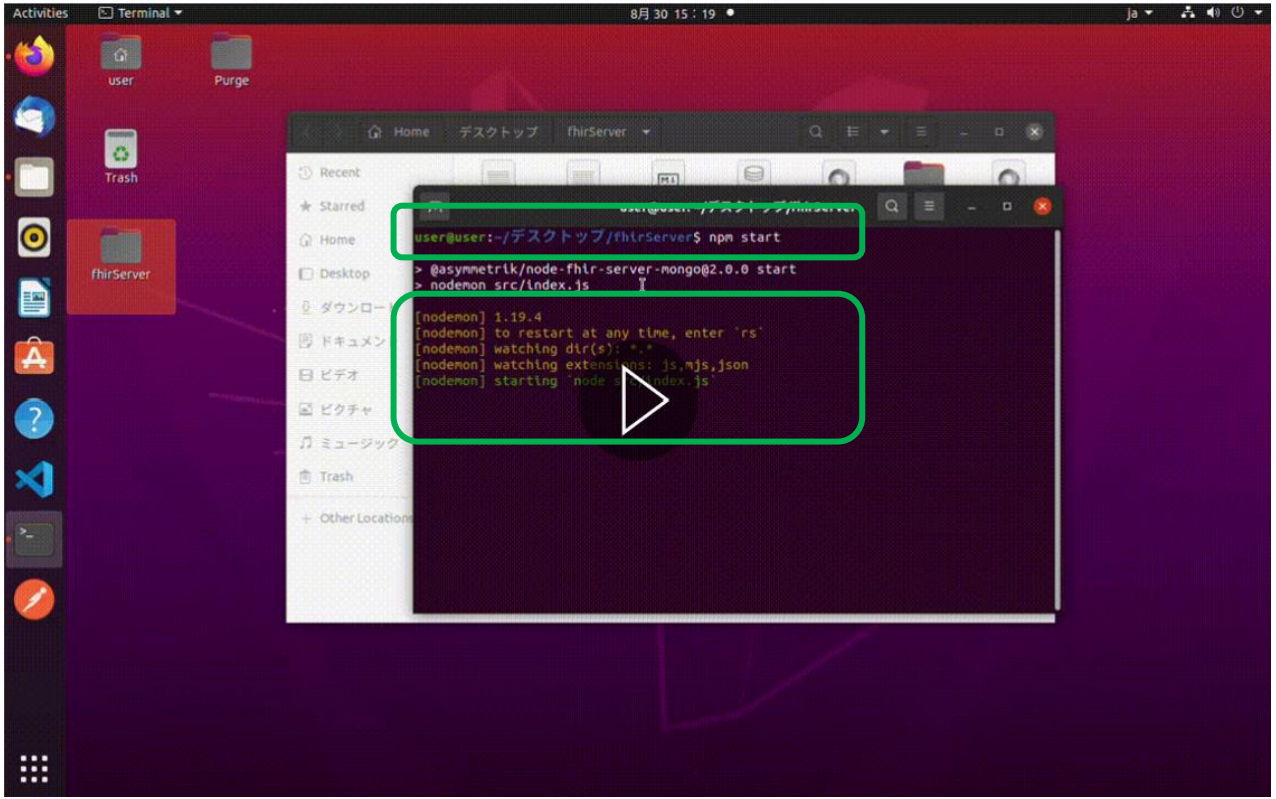
デスクトップにある「fhirServer」を開く
ウィンドウ上で右クリック→”Open in Terminal”→コンソールを起動



FHIRサーバ起動(2)

Asterisk
(FHIRサーバ)

npm start と入力→asymmetrik（FHIRサーバ）が起動する



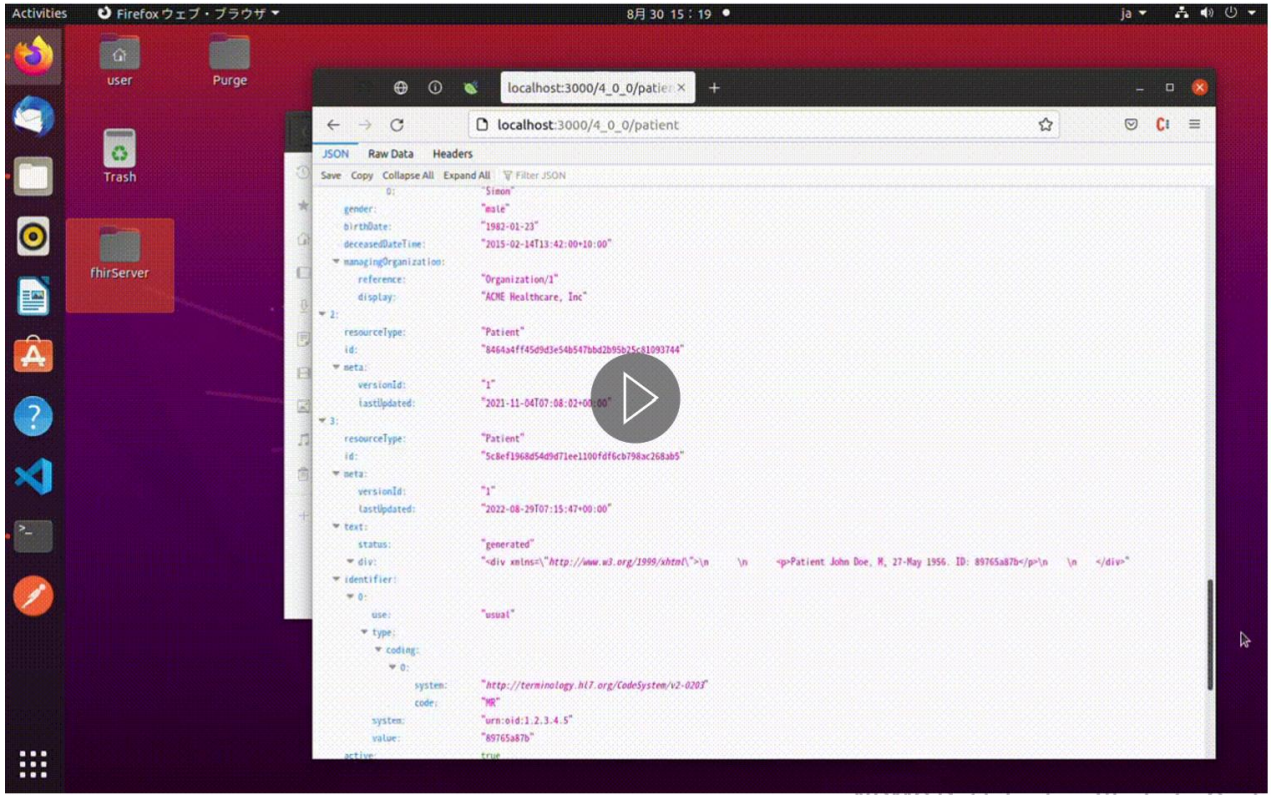
npm start：expressフレームワークを使用したwebサーバ起動の一般的方法

FHIRサーバの動作確認

Asterisk
(FHIRサーバ)

Firefoxを起動

アドレスバーに”localhost:3000/4_0_0/patient/”→FHIR/JSONが表示される



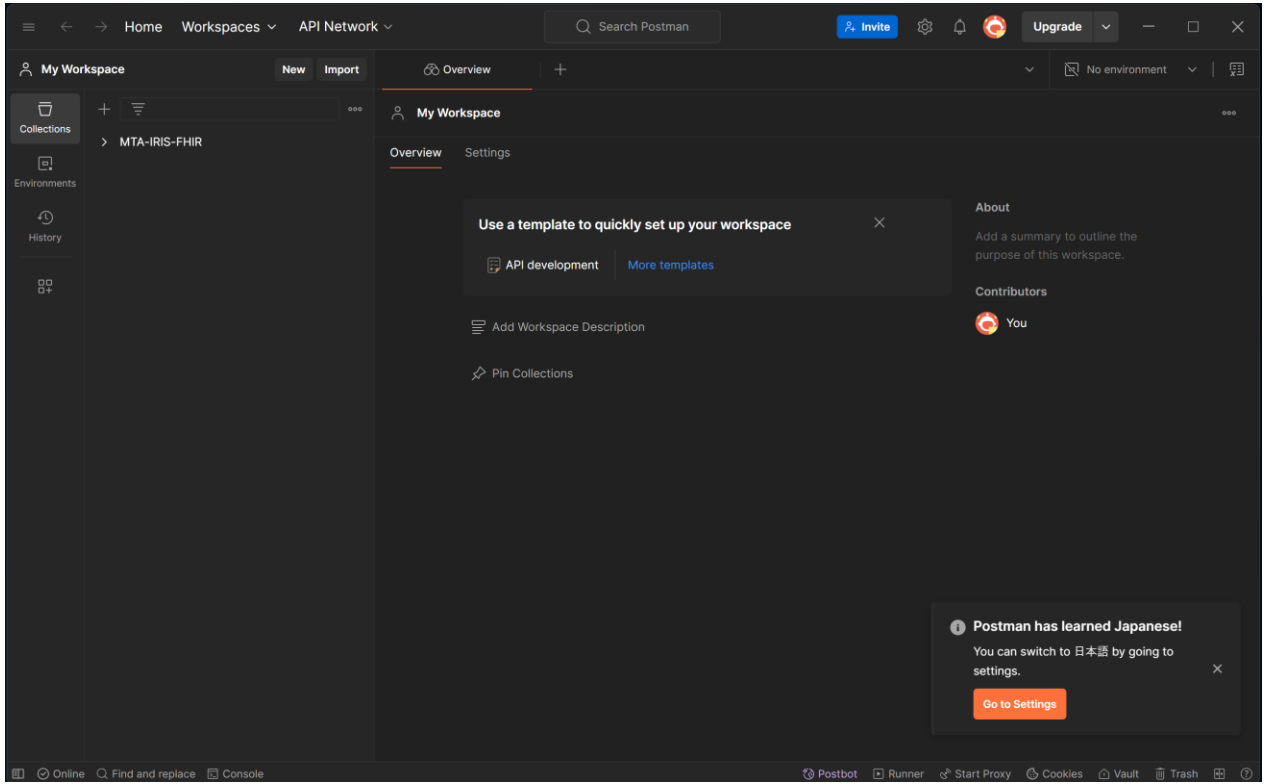
ベースPCからFHIR/Webサーバとして使用する

Postmanを起動

GETタブを起動

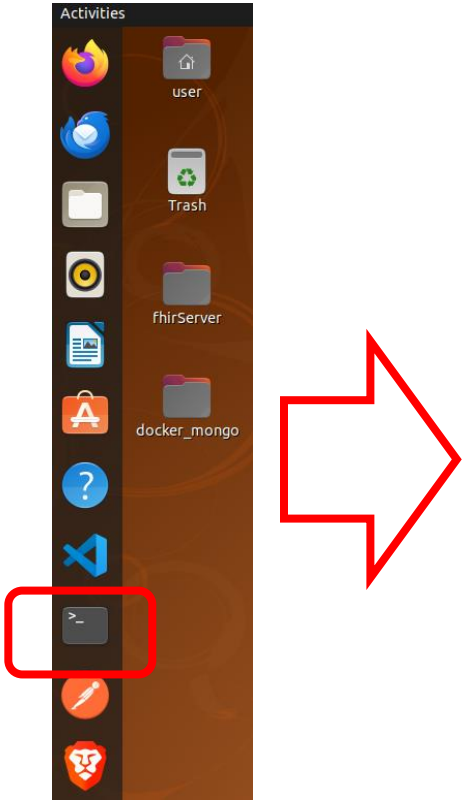
Asterisk

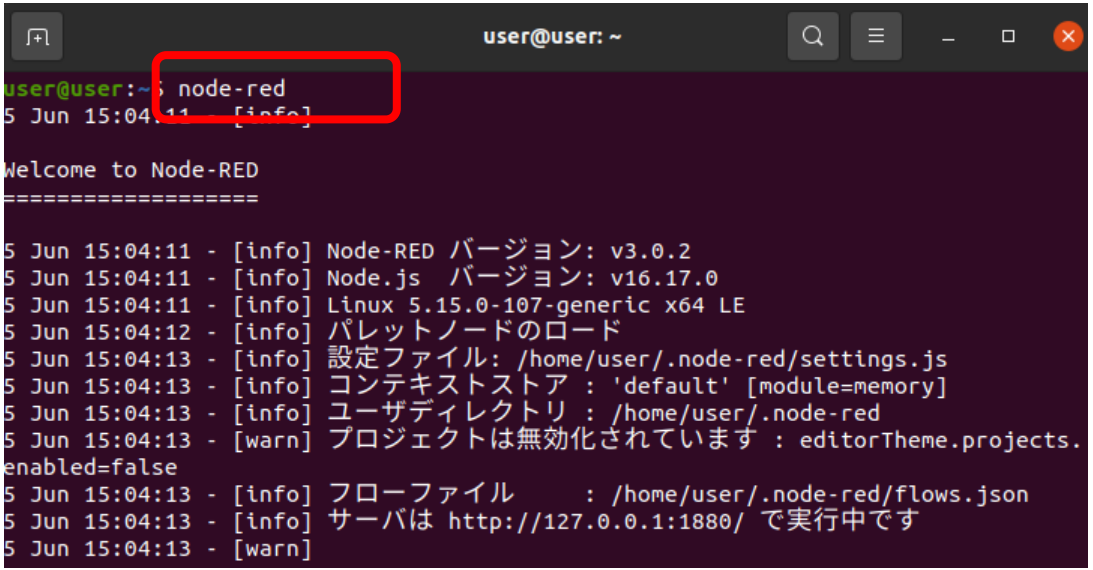
(FHIRサーバ)



Node-Redの起動

- ターミナルを開いてNode-Redを起動します。

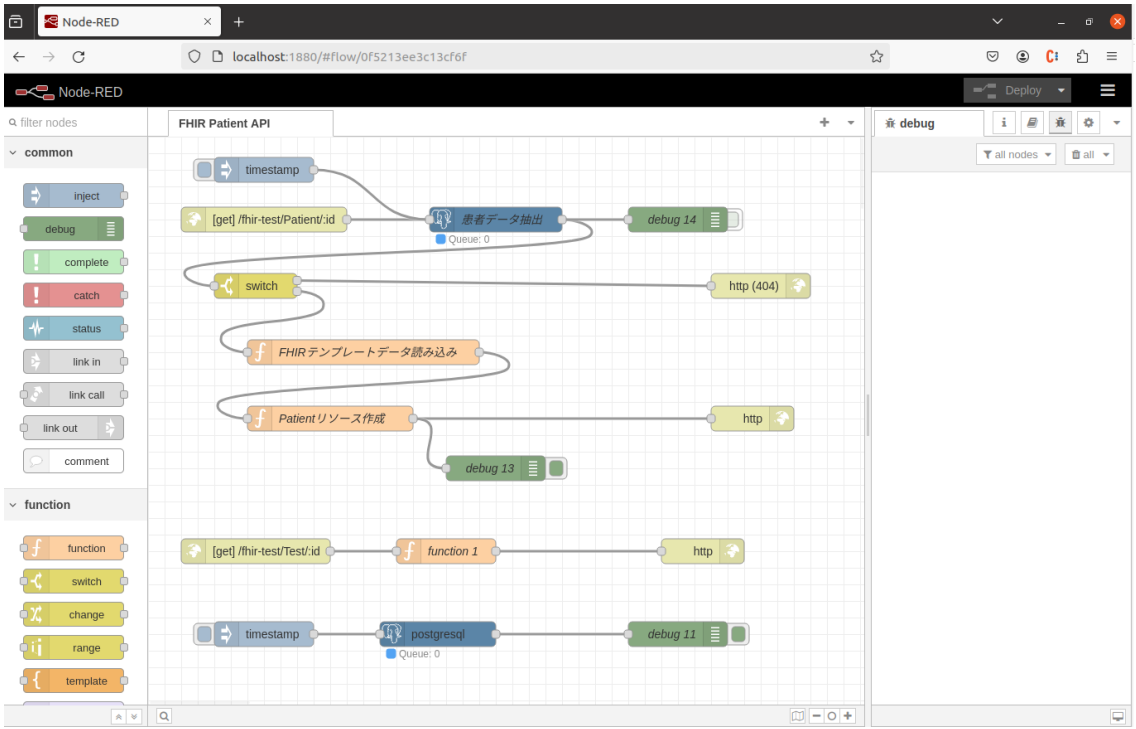




Node-Redの表示

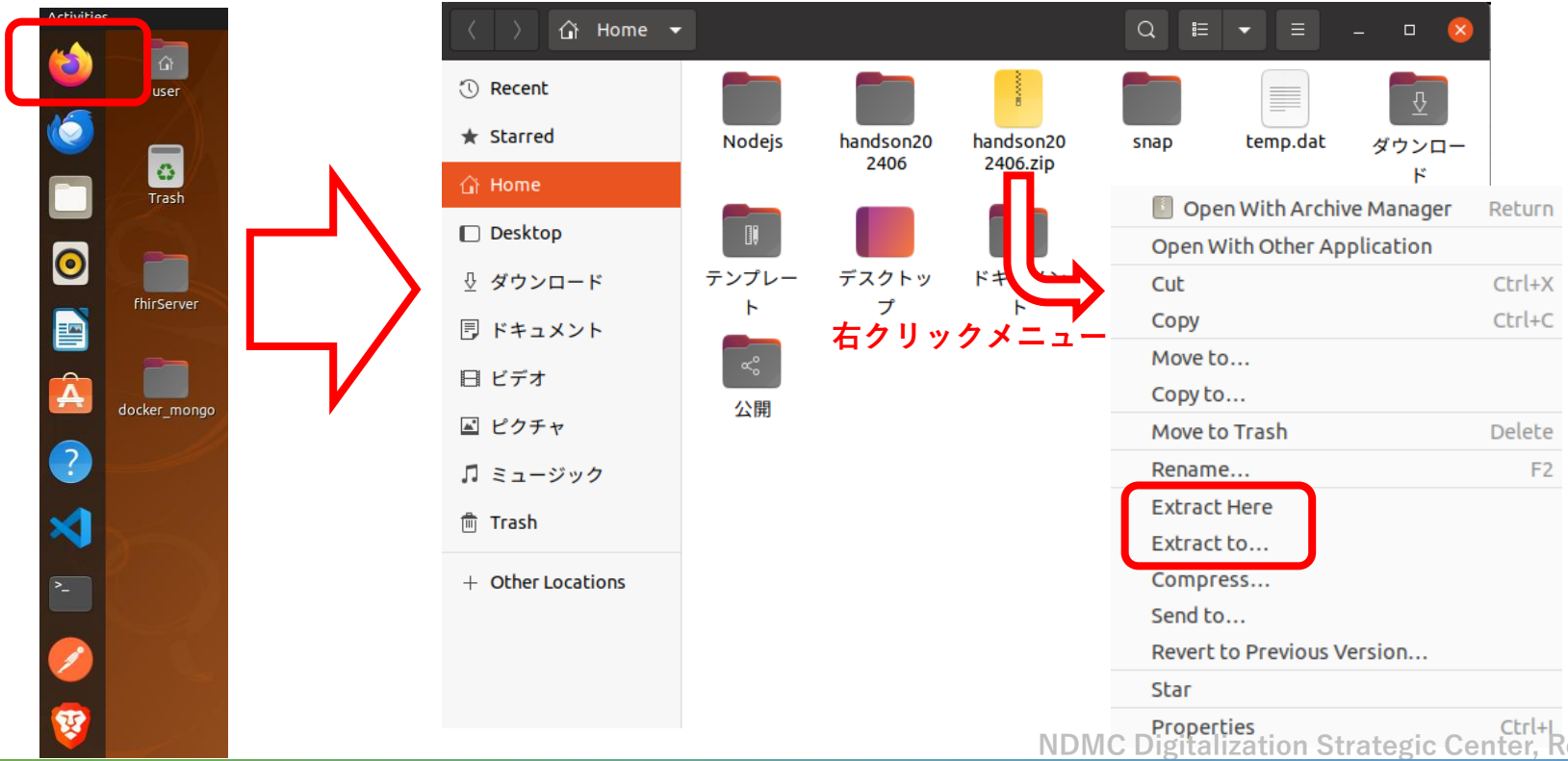
- WebブラウザでNode-RedのURLを開きます。

http://localhost:1880/



ハンズオン用ファイルのコピー

- ホームフォルダにダウンロードしたzipファイルをコピーして展開します。



テスト用データベースの構築

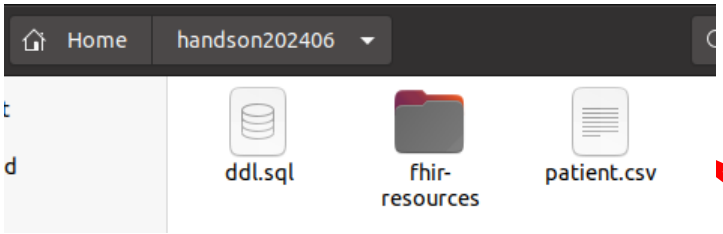
- 新しくターミナルを開いて以下のコマンドを実行します。

```
sudo apt update
sudo apt install postgresql postgresql-contrib

sudo -u postgres createuser -d -P "user“


sudo -u postgres createdb -O "user" emrdb
```

userのパスワードを設定する
プロンプトが表示されるので
adminと入力する。

テスト用データベースの初期データ登録

- ハンズオン用のデータが入ったフォルダにあるddl.sql(データベースの構築スクリプト)を実行します。

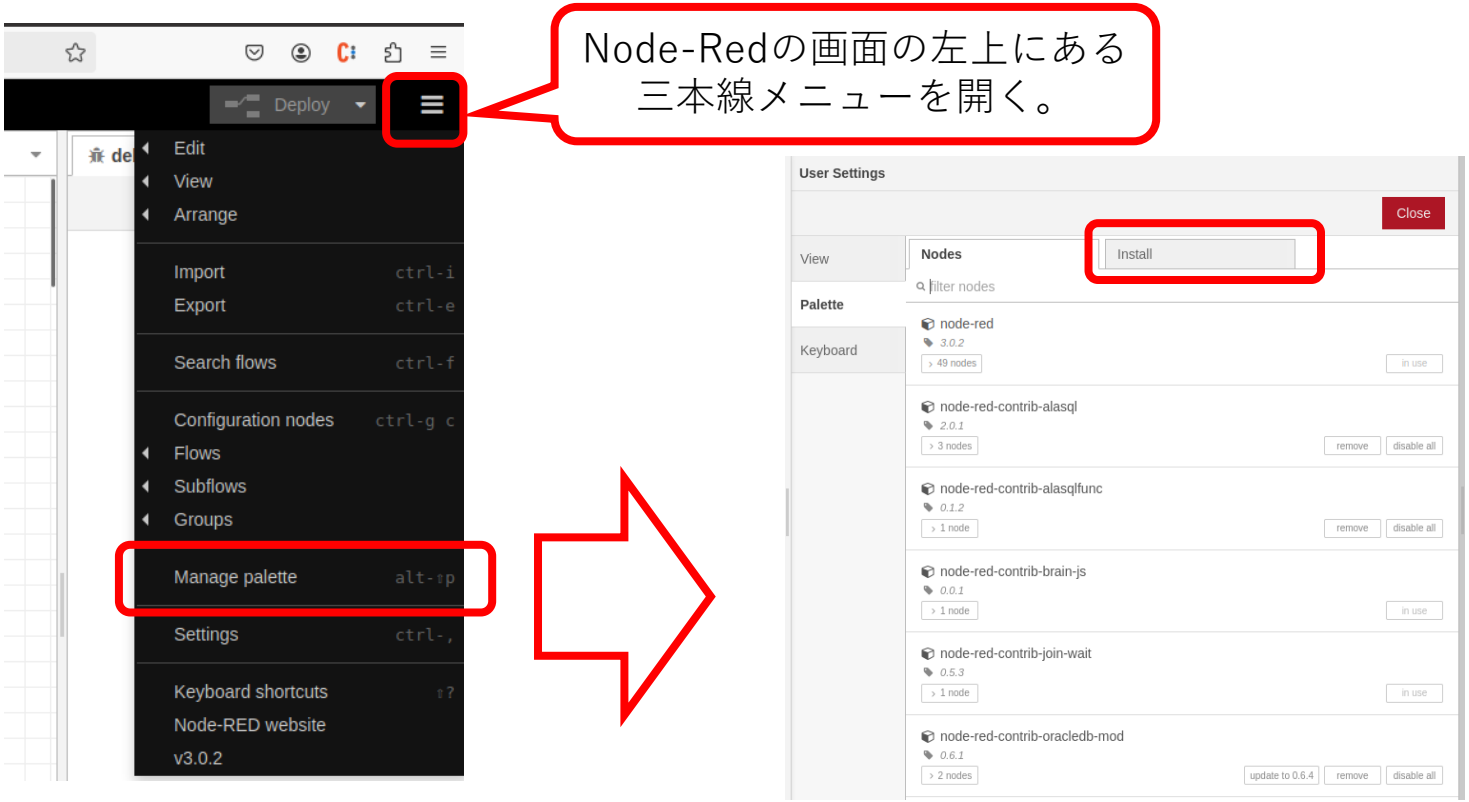


このファイルを開くとテスト用のデータが入っていますので編集してみてください。編集はVisual Studio Codeがおすすめです。

```
cd handson202406

psql -d emrdb

psql -f ddl.sql -d emrdb
```

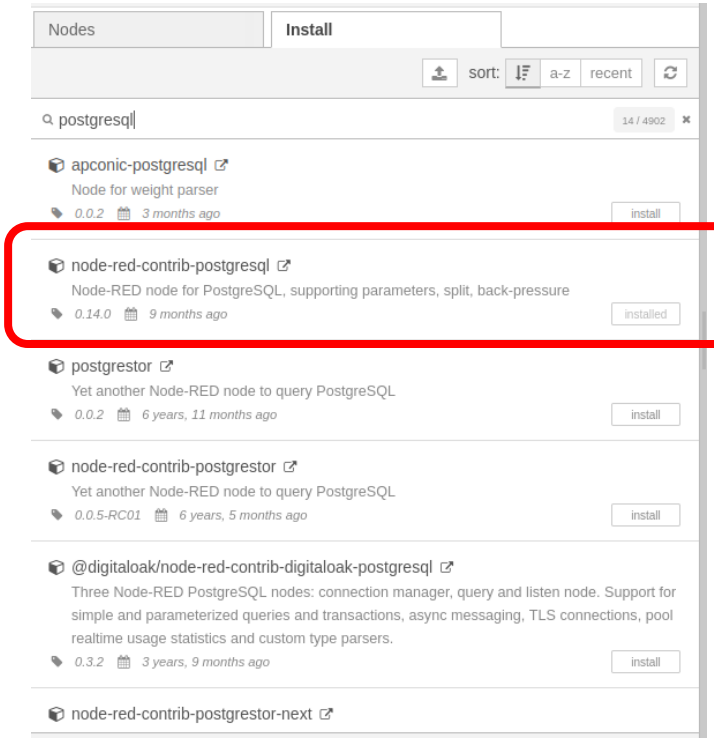
Node-RedのDB接続プラグインインストール

- Node-Redの管理画面にあるManage Palletを開きます。



Node-RedのDB接続プラグインインストール

- InstallタブからDBプラグインを検索してインストールします。



node-red-contrib-postgresql
をインストール

インストールが完了したらNode-Redの再起動(Node-Red実行中のターミナルでCtrl-C押下で停止後に起動コマンド実行)します。

本日の流れ

- **HL7 FHIRとIPCIについて（20分）**
  - 医療情報ユーザーから見たFHIRアプリケーションの利点
  - FHIRとはどんな規格か、REST, JSONについて
  - M学会公式サービスセットIPCIについて

- **ハンズオン ～IPCIを使用したFHIRサーバを作ろう！～（90分）**
  - 環境準備・確認（IPCIの起動確認、各種ライブラリのインストール）
  - ハンズオン1：Node-Redを使用したWebサービスを作ろう！
  - ハンズオン2：Node-Redを使用してデータベースにアクセスしてみよう！
  - ハンズオン3：データベースから患者情報を取得してFHIRのPatientリソースとして返すWebサービスを作ろう！

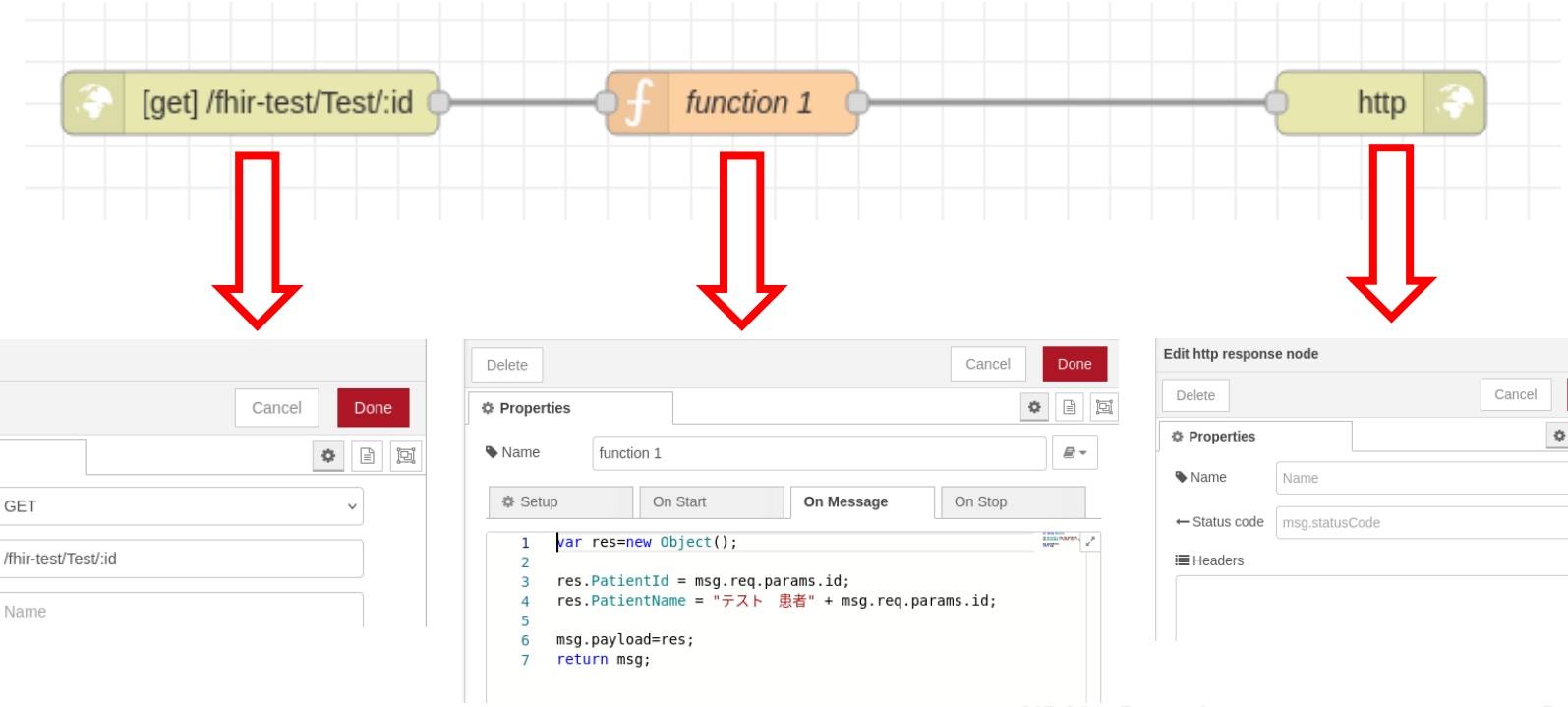
- 質疑、学会案内（10分）

新規フローの作成

- Node-Redの画面から新規フローを作成します。

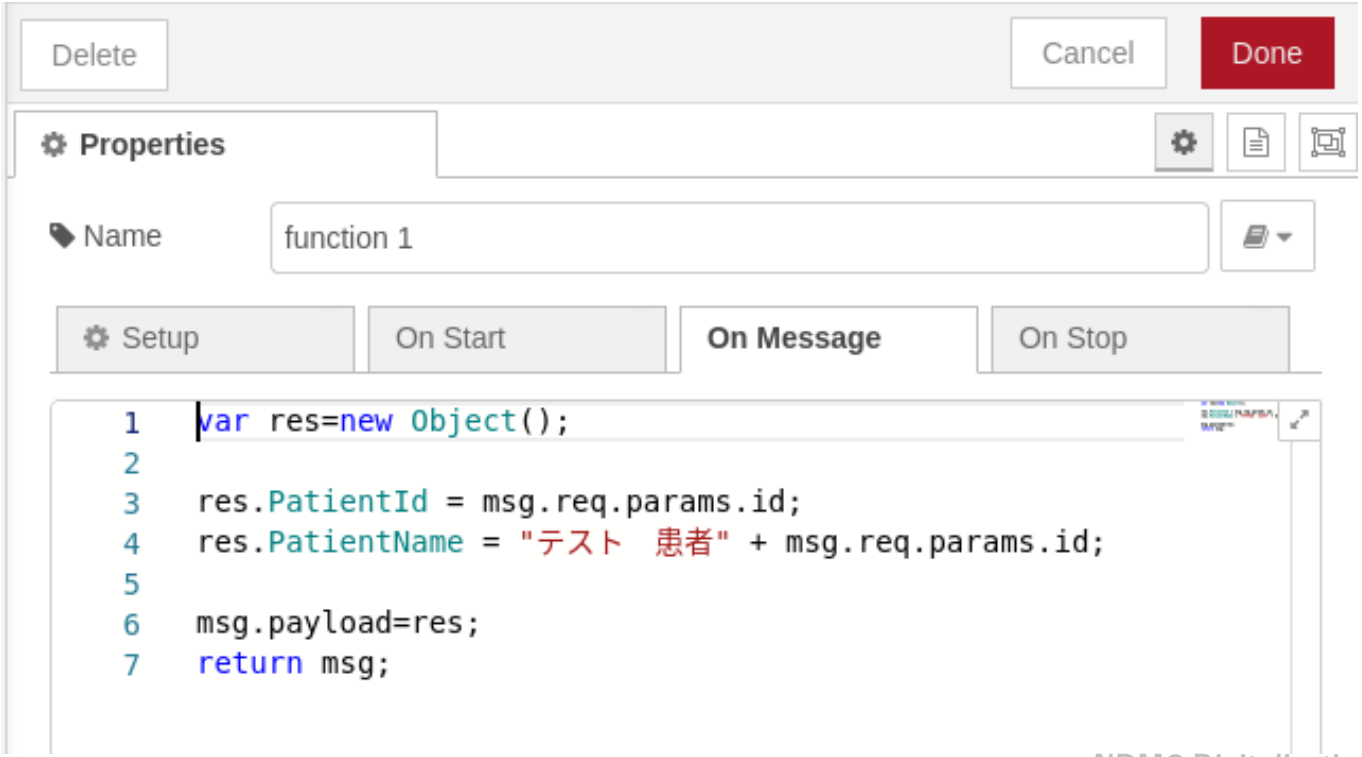
コンポーネントの配置

- WebのGETリクエストを受付でレスポンスを返す単純なREST APIを作成します。



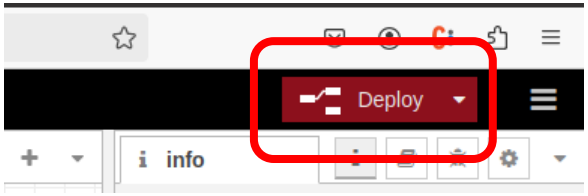
functionコードの実装

- 以下のようなコードを書いてみましょう。



フローのデプロイ

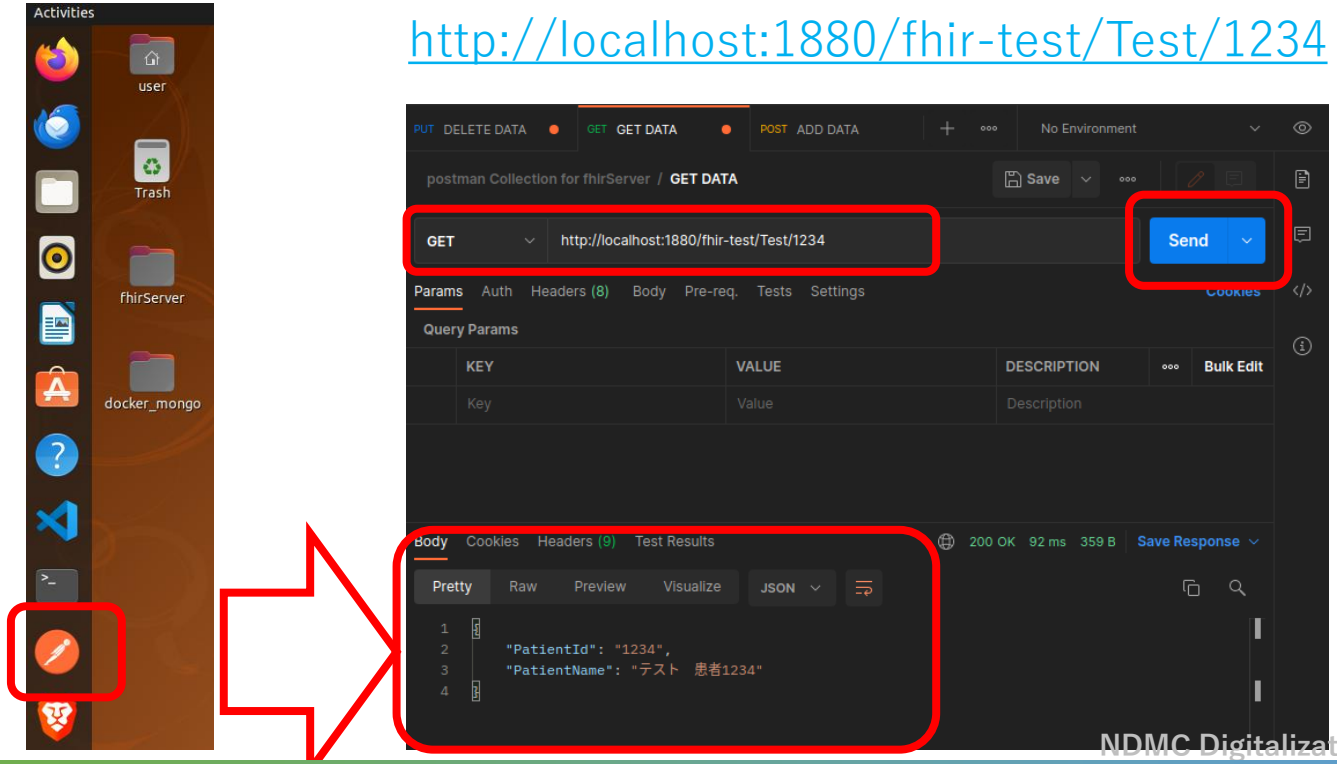
- デプロイボタンをクリックしてフローを保存し実行可能な状態にします。



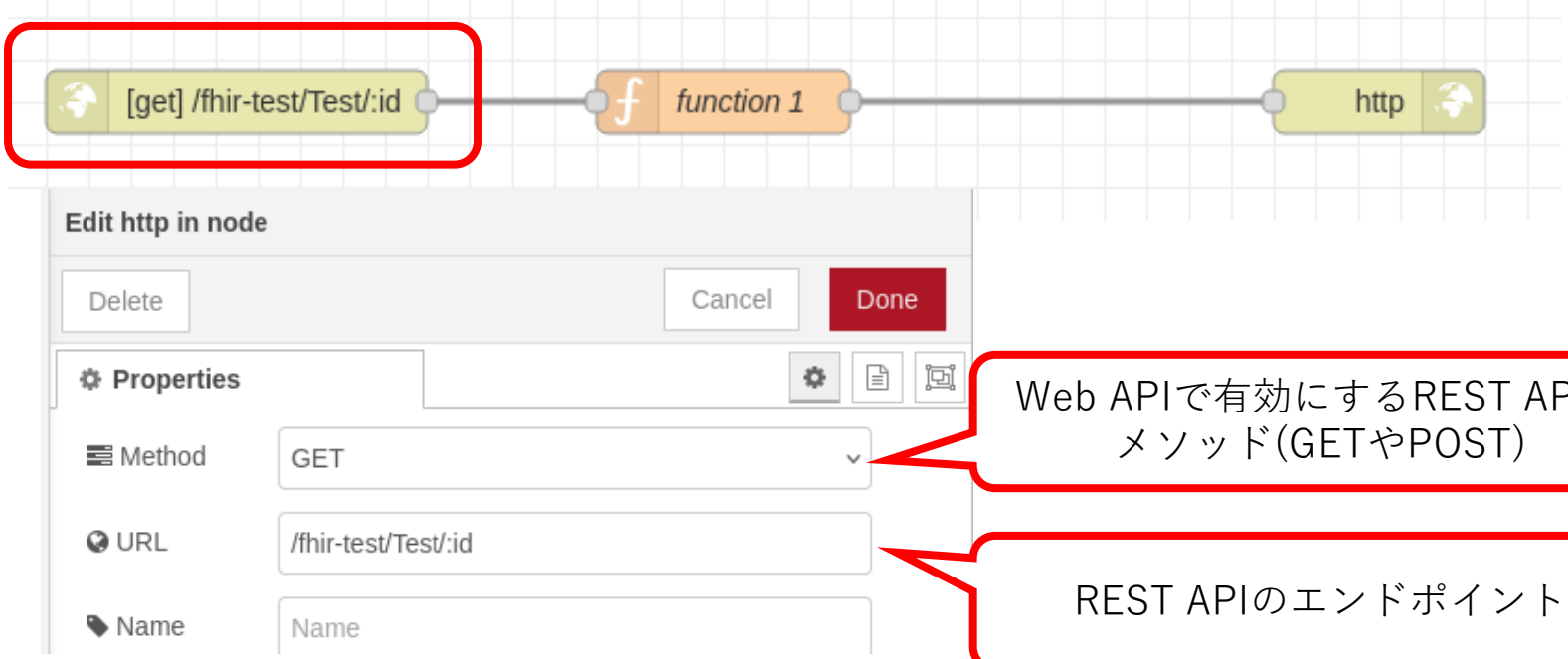
フローの実行

- postmanを開いて以下のURLをGETで実行するとfunction1で実装したレスポンスが返ってくることを確認します。

http://localhost:1880/fhir-test/Test/1234



フローの解説 1

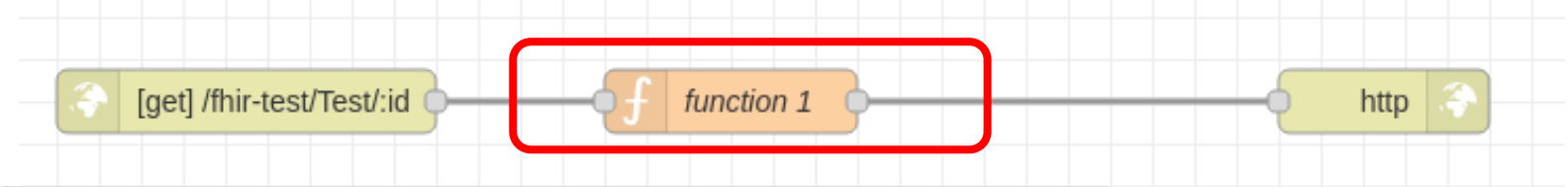


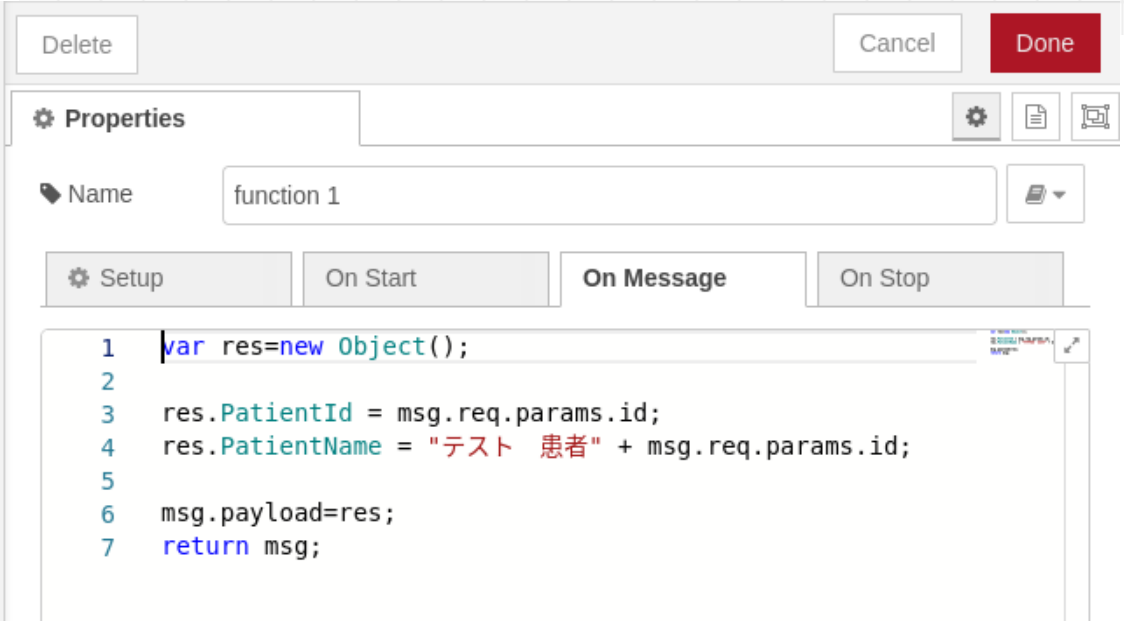
Web APIで有効にするREST APIのメソッド(GETやPOST)

REST APIのエンドポイント

今回のURLにある「:id」はリクエストパラメータになります。

フローの解説 2



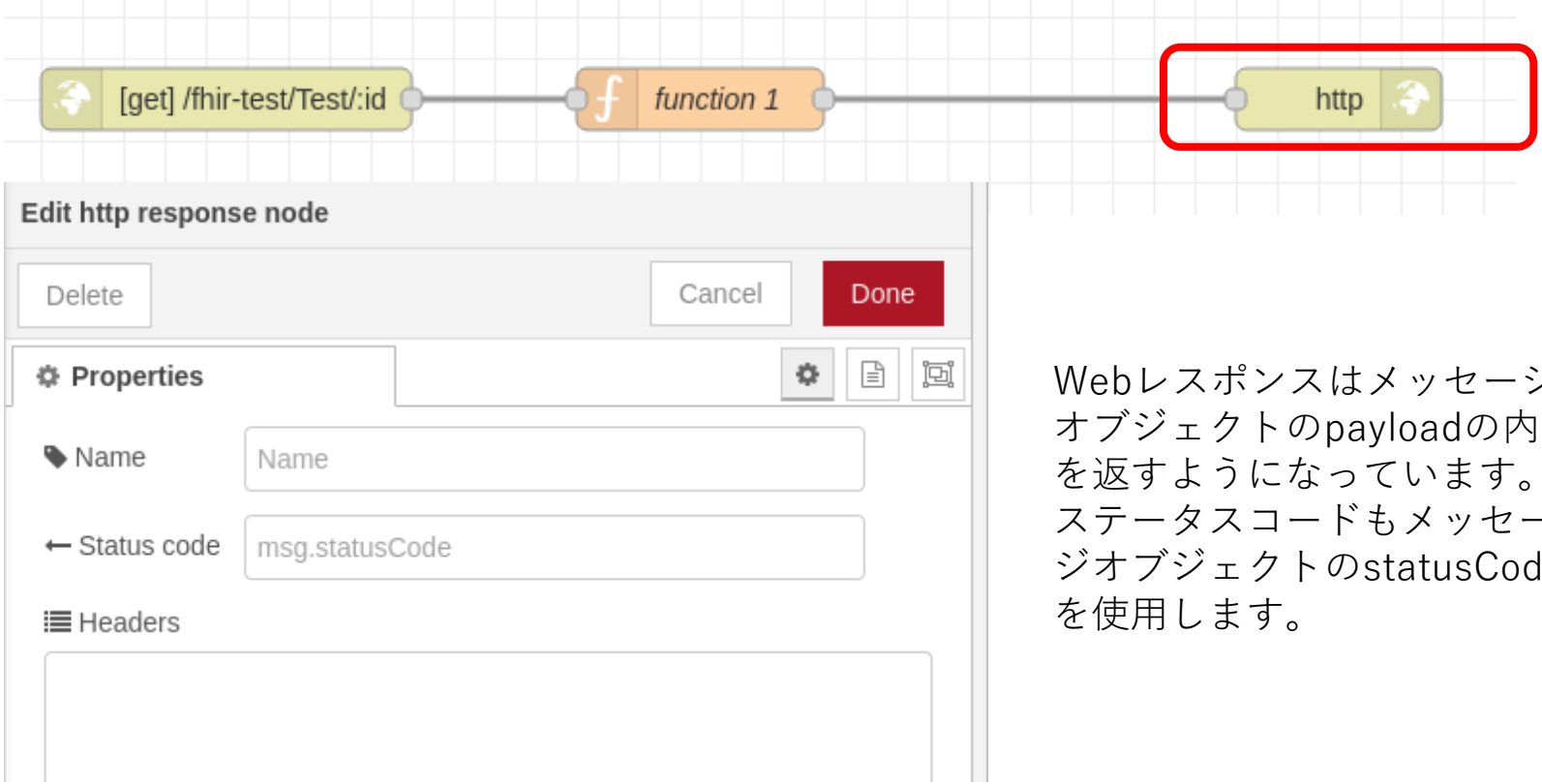


Node-Redはメッセージオブジェクト(msg)をコンポーネント間で受け渡していくことで処理を定義します。

前のコンポーネントであるWebリクエスト受付ではWebリクエスト情報がmsg.reqに保存されています。

処理結果は通常payloadに保存しておきreturnでメッセージオブジェクトごと関数の戻り値とします。

フローの解説 3



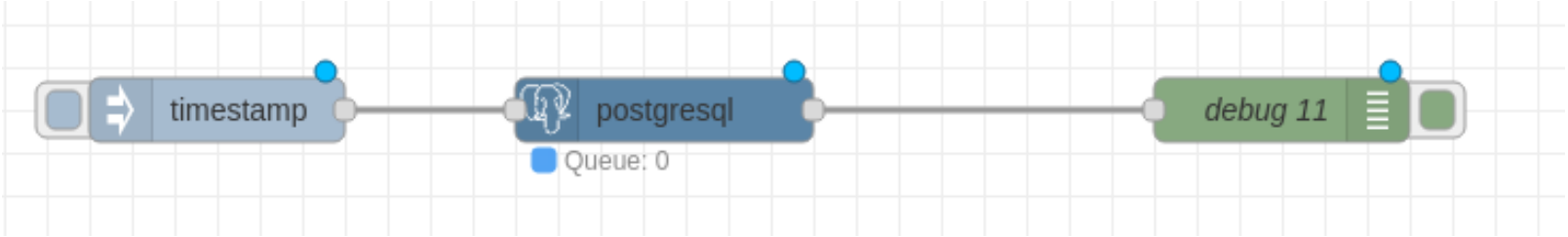
Webレスポンスはメッセージオブジェクトのpayloadの内容を返すようになっています。ステータスコードもメッセージオブジェクトのstatusCodeを使用します。

新規フローの作成

- Node-Redの画面から新規フローを作成します。

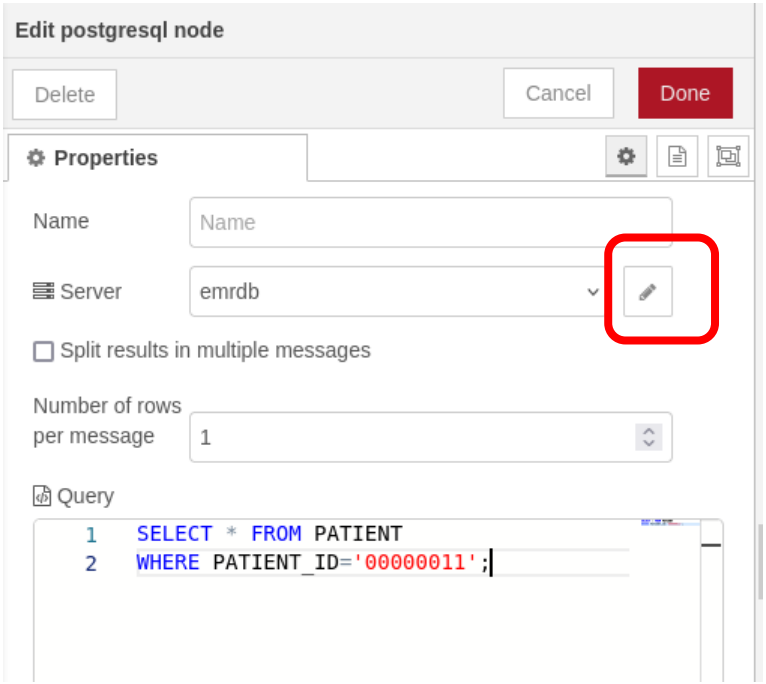
コンポーネントの配置

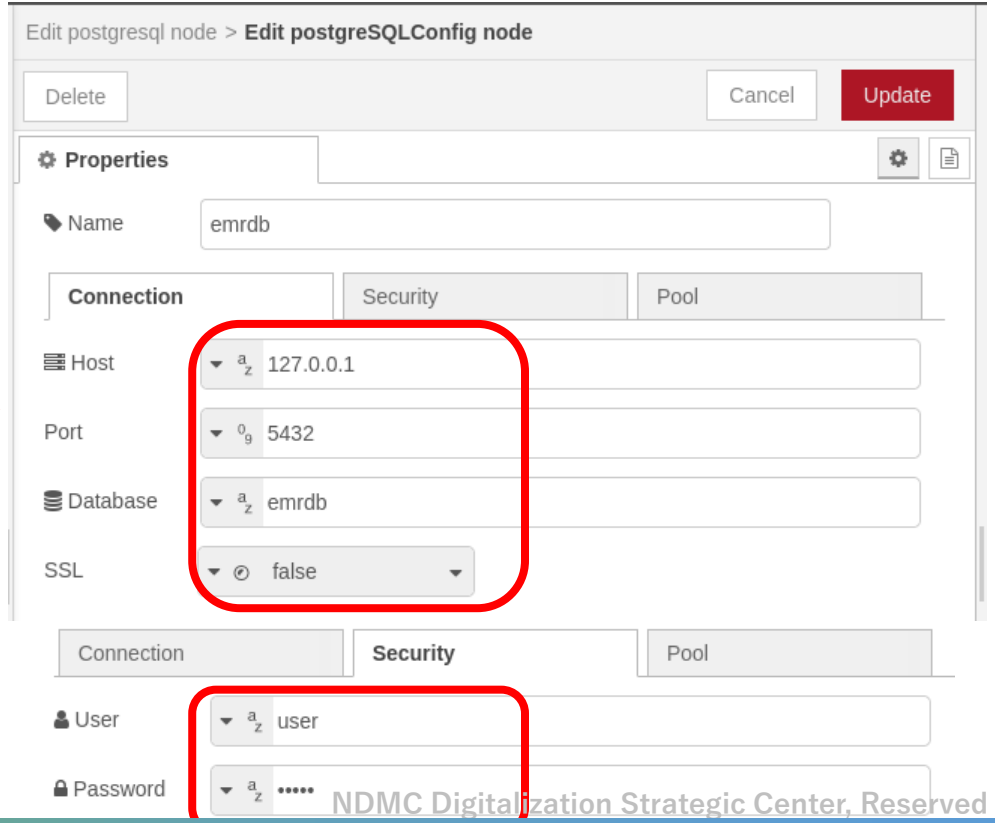
- データベースに接続してデータを取得するフローを作成します。



データベース接続設定

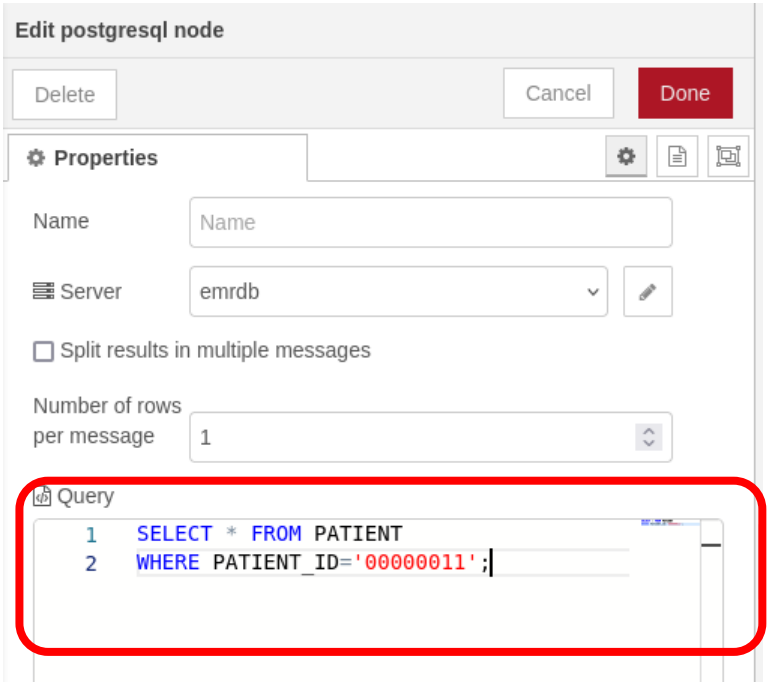
- postgresql Nodeの設定を行います。





SQLの記載

- postgresql Nodeの設定で実行するSQLを設定します。

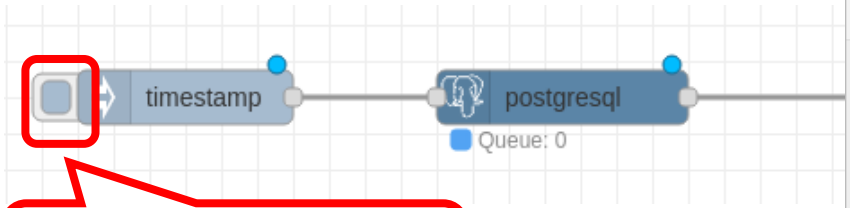


Query欄に実行するSQLを記載します。今回は固定の条件ですがパラメータも使用可能です。

取得したデータはArray型でメッセージオブジェクトのpayloadに格納されます。

デプロイ＆実行

- デプロイ後Injectノードの右端をクリックするとフローが実行できます。
実行結果はデバッグウィンドウから確認できます。

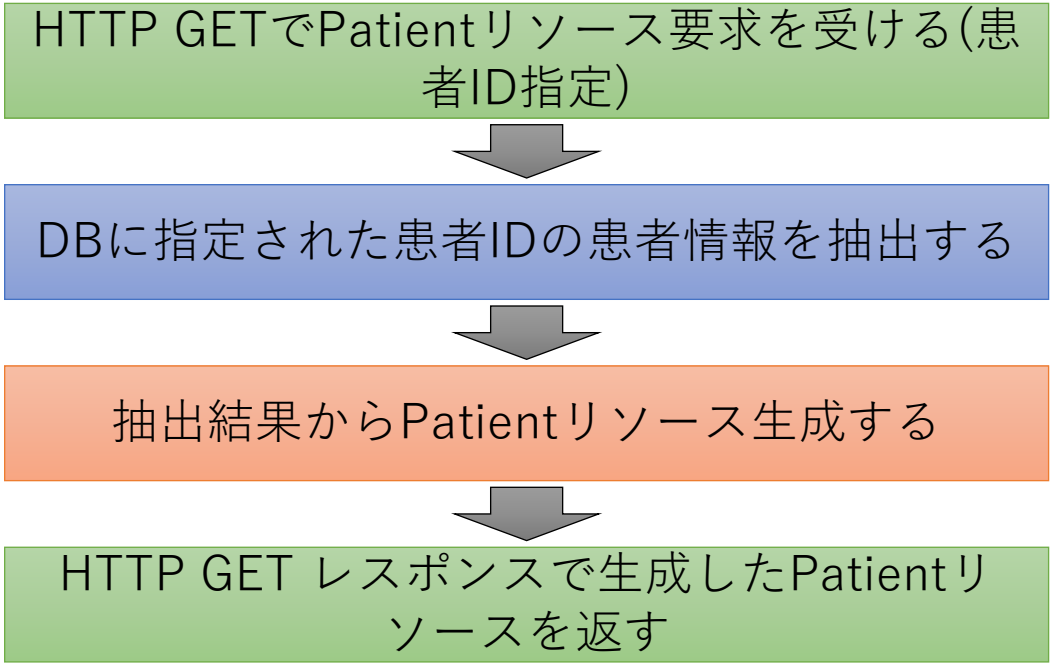


ここをクリックすると
実行できます。

デバッグウィンドウは
ここをクリックすると
表示します。

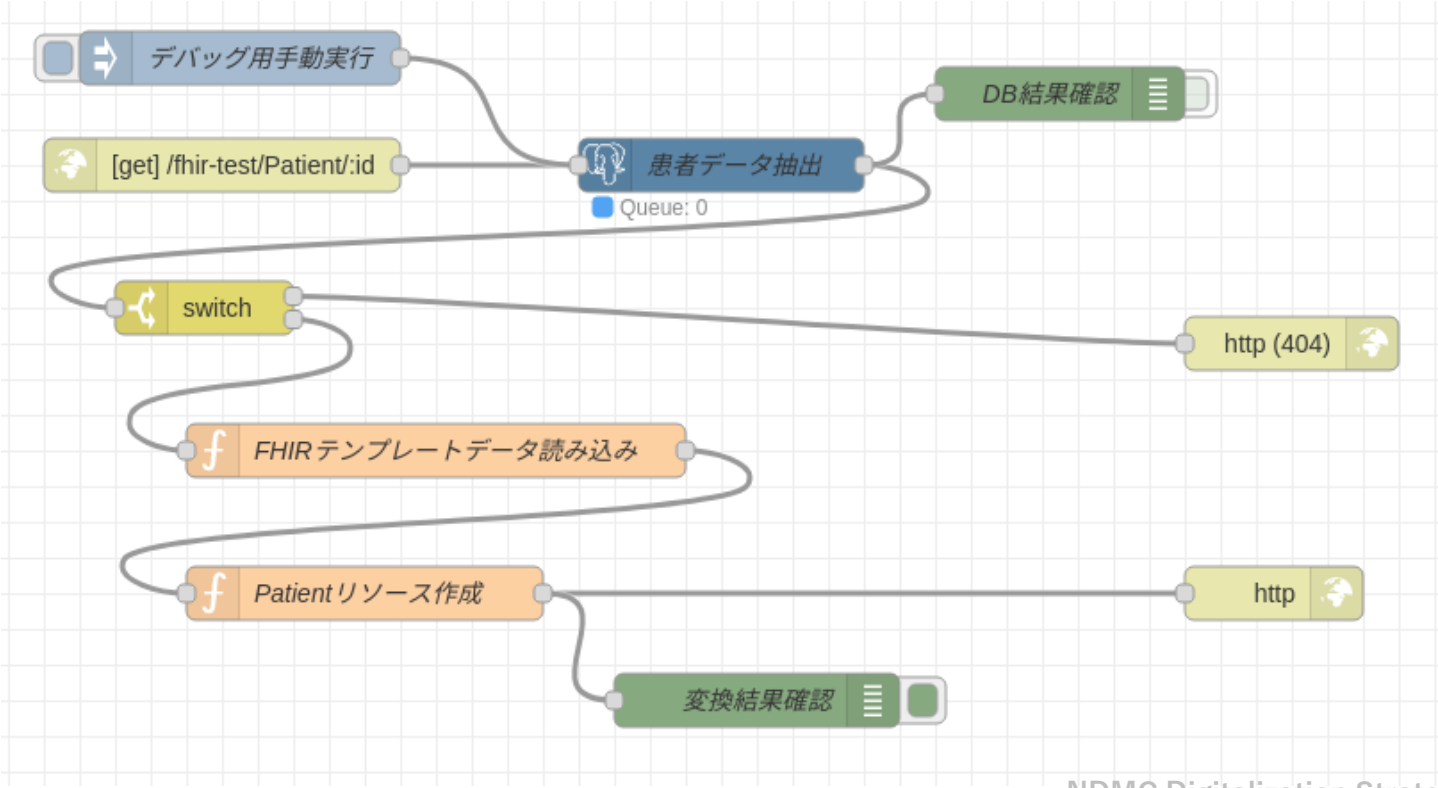
処理の流れの確認

- 基本的にはハンズオン1とハンズオン2の組み合わせとなりますがこれにFHIRリソースへの変換が入ります。大まかな処理の流れは次のようになります。



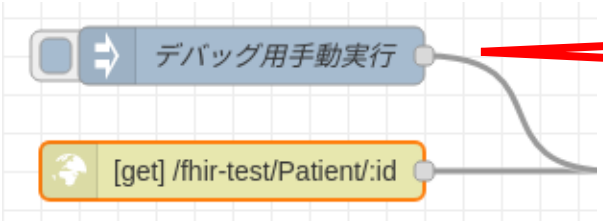
コンポーネントの配置

- 処理概要に沿って各種コンポーネントを配置します。

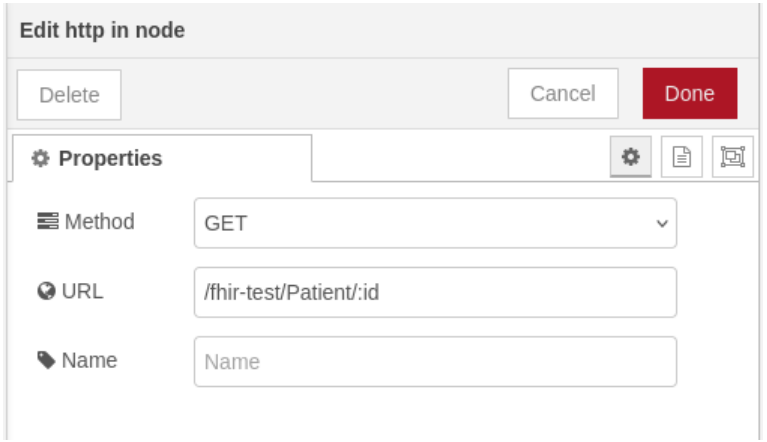


REST APIによるリクエスト受付

- REST APIのエンドポイントとパラメータを定義します。

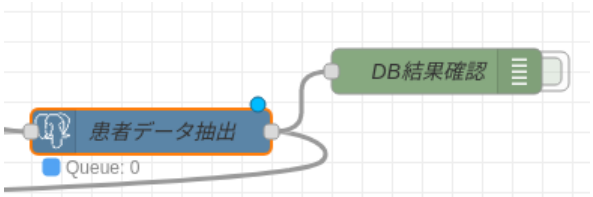


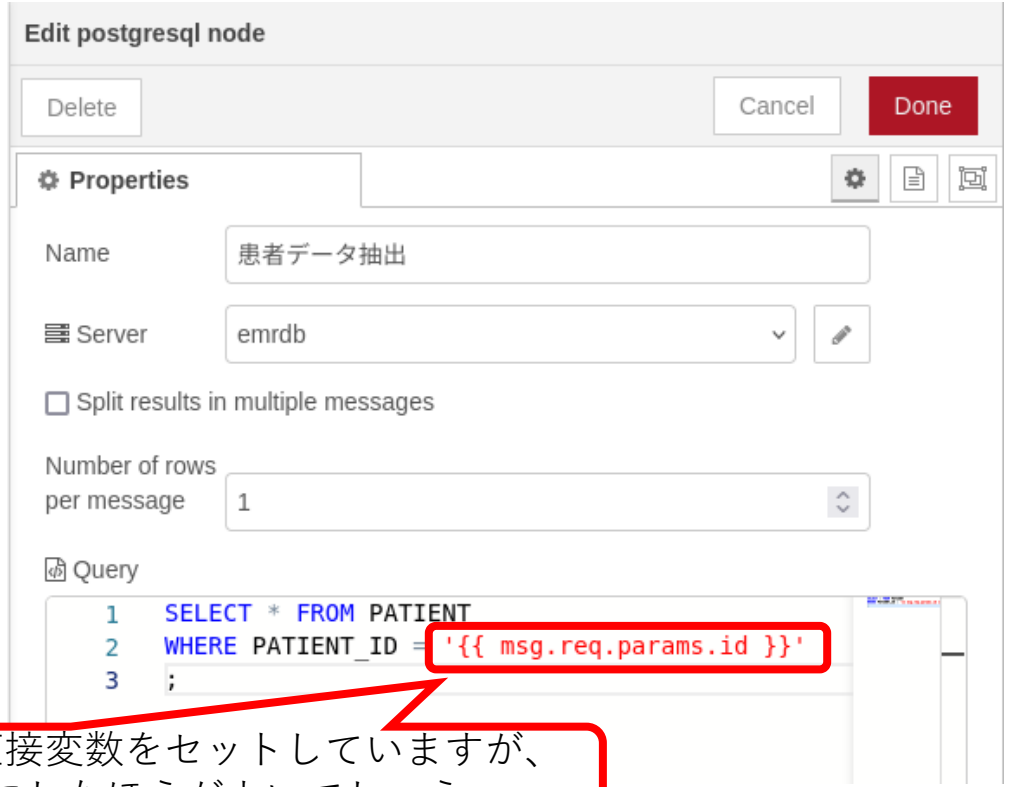
これはデバッグ用の手動実行です。
本来の実行と同様のパラメータを
セットできるようにしてあります。



DBからのデータ抽出

- 受け取った患者IDのパラメータを使用してSQLを実行します。

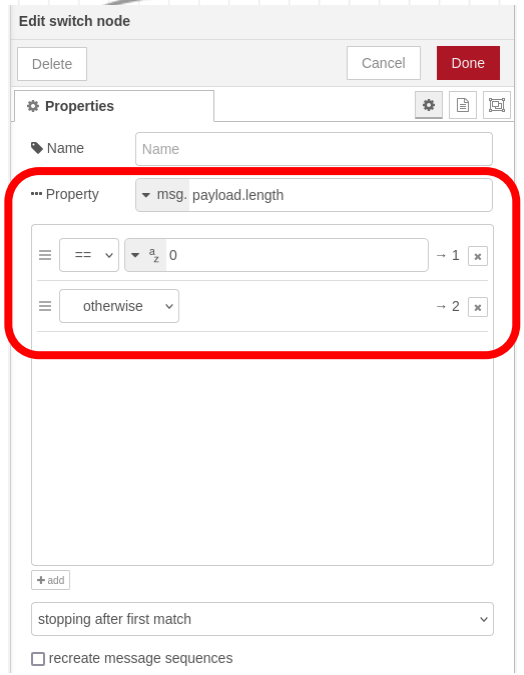


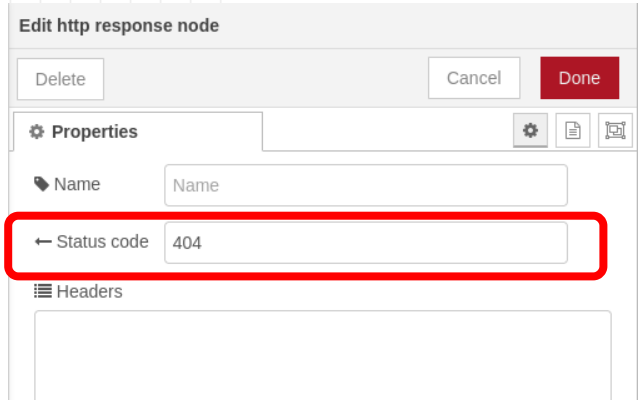


今回はサンプルということもあり直接変数をセットしていますが、
本来はパラメタライズドクエリにしたほうがよいでしょう。

DB結果からの条件分岐

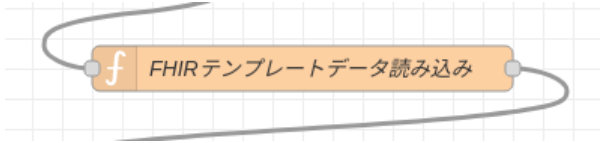
- 患者情報がなかったときは404が返るようにします。

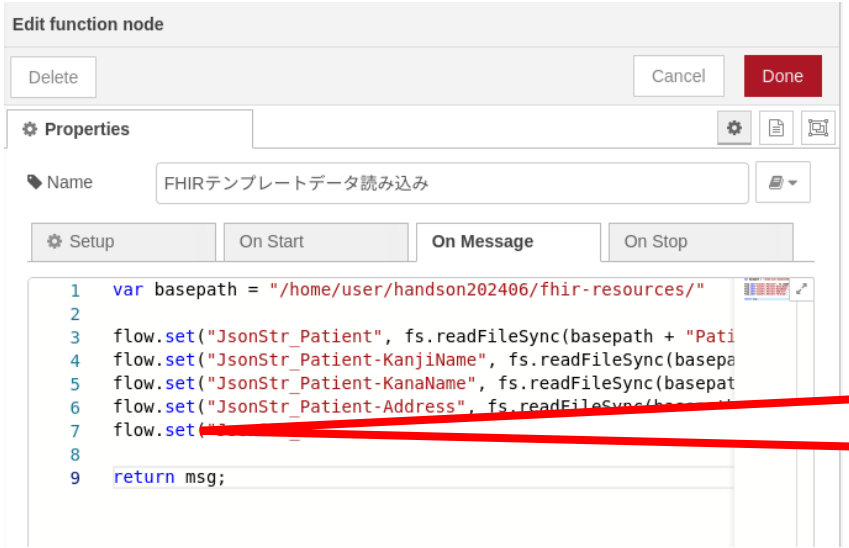


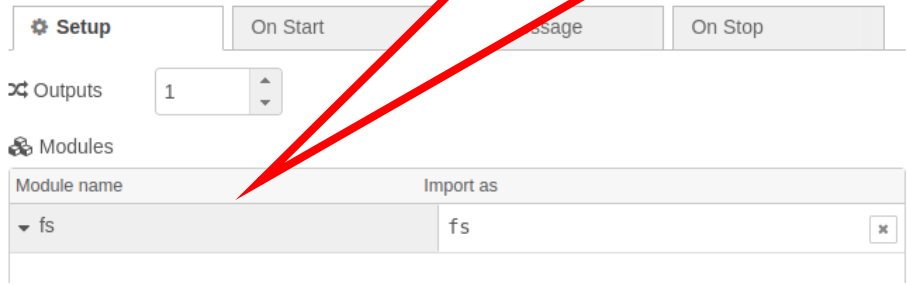


FHIRデータのテンプレート読込

- ハンズオンフォルダにFHIRリソースのテンプレートファイルが含まれていますので、それを読み込みます。





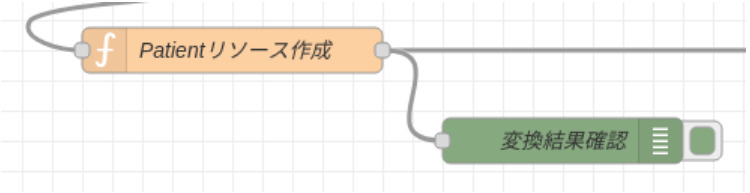


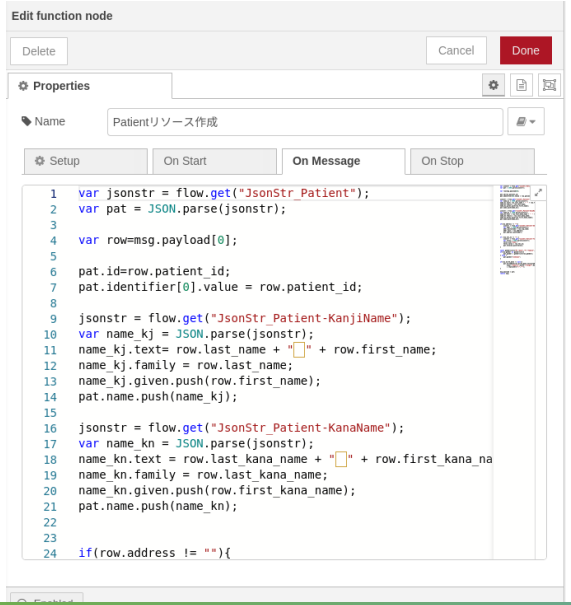
モジュールのインポートは
Setupタブで定義します。

「flow」はフロー内で使用可能な変数を格納できるオブジェクトです。
メッセージオブジェクトとの使い分けに注意しましょう。

FHIRデータへのマッピング

- FHIRリソーステンプレートにDBから抽出したデータをセットしていきます。





flow変数に格納されたPatientリソースのJSON文字列を取得

↓

JSON文字列からJSONオブジェクトへ変換

↓

DBから取得したデータをJSONオブジェクトにセット

↓

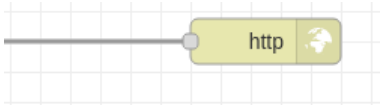
子要素もそれぞれJSON文字列から変換したJSONオブジェクトをひな型としてデータをセット

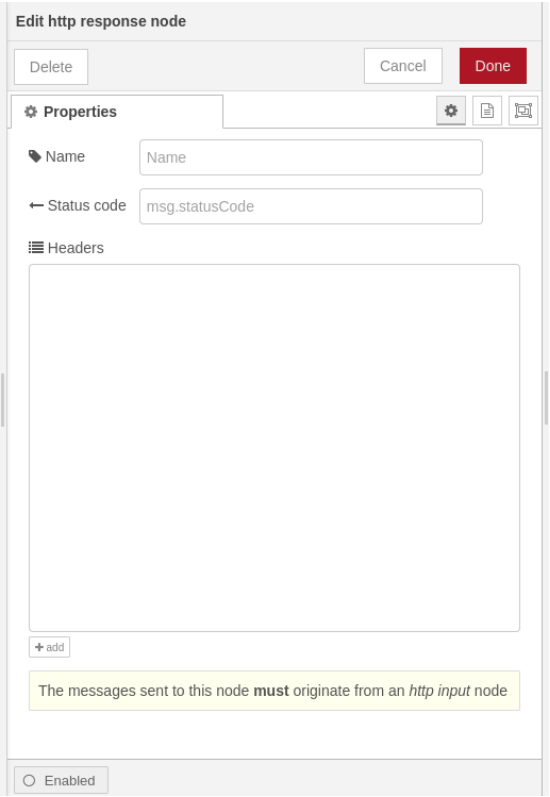
↓

メッセージオブジェクトのpayloadにPatientをセット

REST APIのレスポンス

- メッセージオブジェクトの内容をレスポンスとして返します。

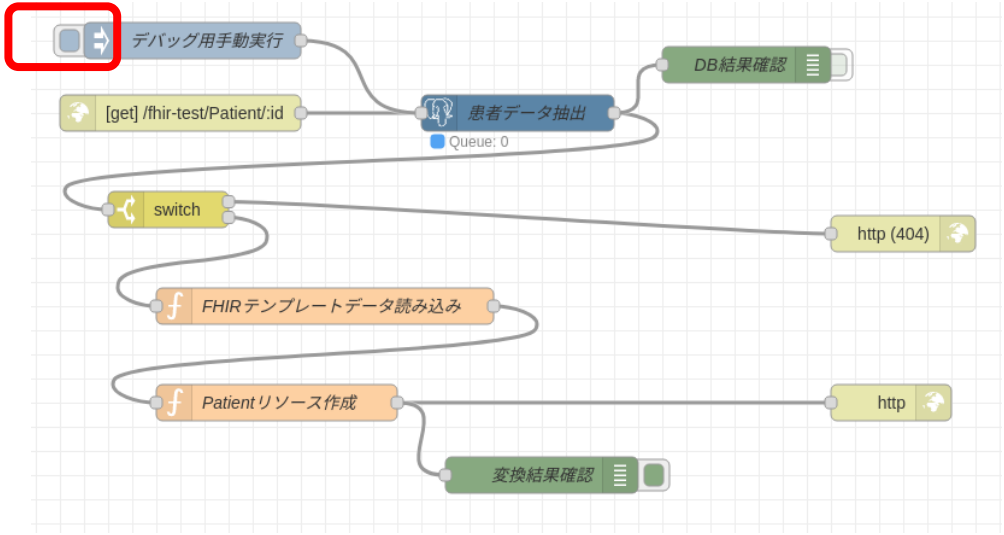




何も設定しなければ正常終了として
メッセージオブジェクトのpayloadの
内容がそのままJSON文字列化されて
返却されます。

フローのデプロイ＆デバッグ実行

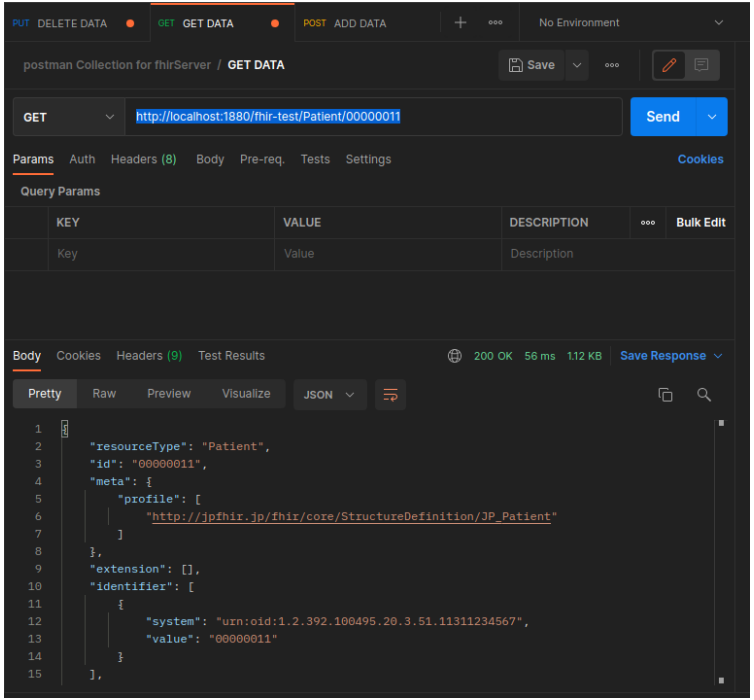
- フローをデプロイした後動作確認のためデバッグ実行します。



実際にREST APIを実行

- postmanを使用して実際にREST APIとして実行してみます。

  http://localhost:1880/fhir-test/Patient/00000011



UMLモデル要素の
Node-REDフローでの表現法

Node-REDにおける業務フロー(UMLモデル)実装性の確認

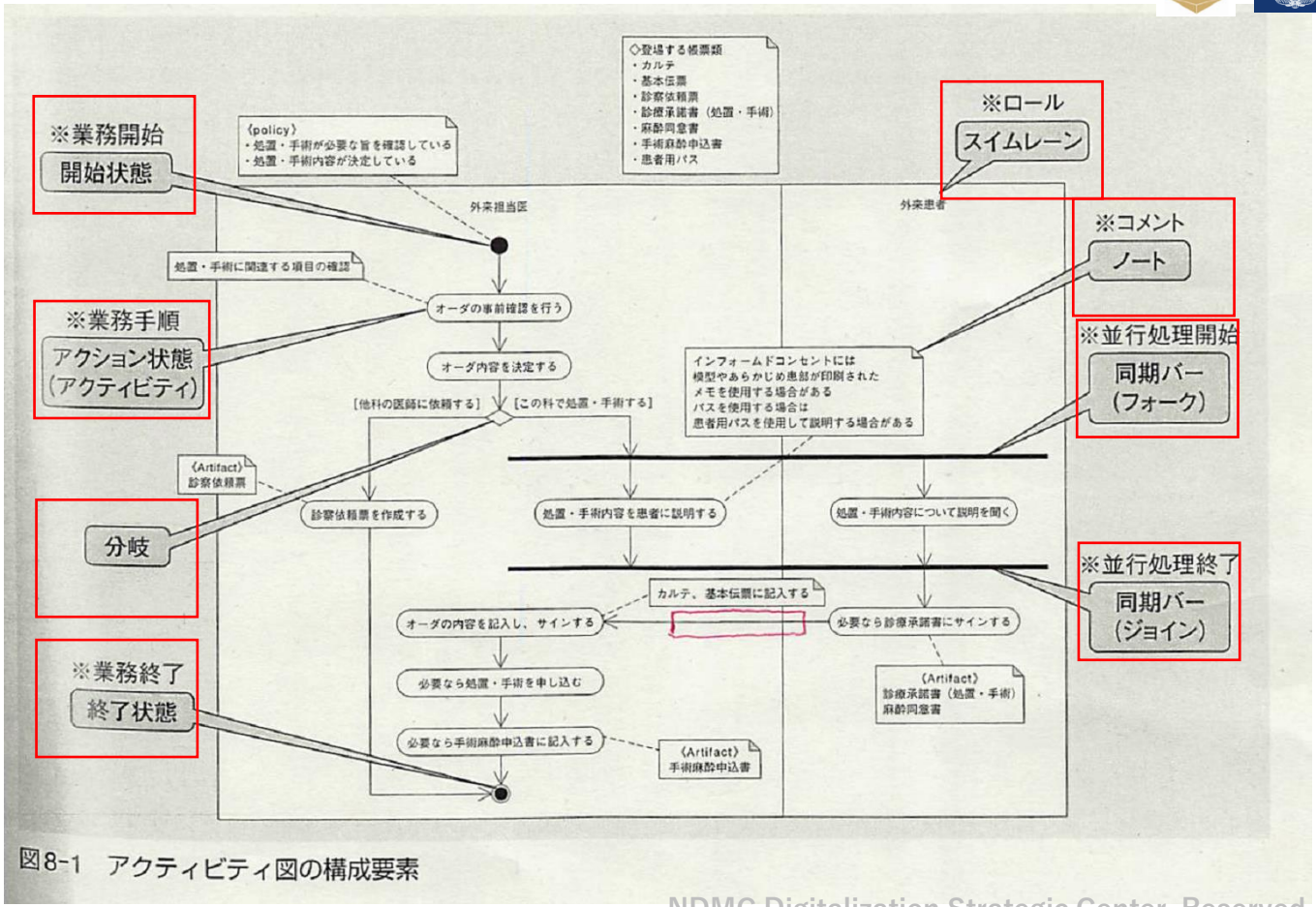


図8-1　アクティビティ図の構成要素

業務フロー記述UMLで必要とされる機能とNode-RED要素の対応調査

モデル要素の対応付け

（1）開始状態
開始状態は処理の開始を表す．アクティビティ図上では黒で塗りつぶした丸で記述する．

図8-2　開始状態

（2）終了状態
終了状態は処理の終了を表す．アクティビティ図上では黒と白の二重丸で記述する．

図8-3　終了状態

（3）アクション状態（アクティビティ）
アクション状態は何かの処理を行っている状態を表し，アクティビティとも呼ばれる．アクティビティ図上では角の丸い長方形で記述する．

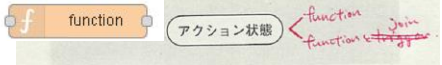


図8-4　アクション状態

（4）遷移
遷移はある処理の状態から別の処理の状態へ遷移することを表す．アクティビティ図上では遷移する方向への矢印で記述する．

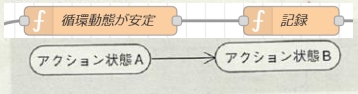


図8-5　遷移

（5）分岐
分岐は何らかの条件によって変化する処理の流れを表す．アクティビティ図上では遷移の矢印に，［ ］を付けた分岐条件で表記する．なお，分岐のポイントはひし形で記述する．

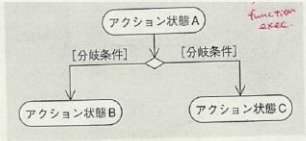


図8-6　分岐

（6）同期バー
同期バーは複数の処理が並行して行われる流れを表す．並行処理の開始を表す同期バーを「フォーク」，並行処理の終了を表す同期バーを「ジョイン」と呼ぶ．アクティビティ図上では太線で記述する．

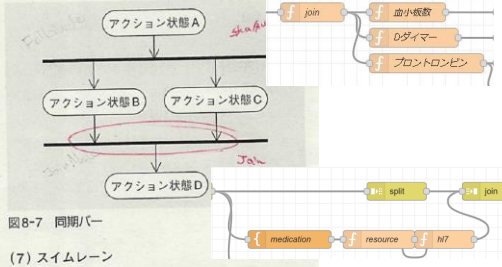
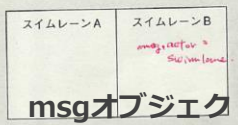
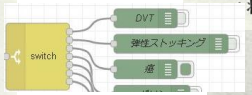


図8-7　同期バー

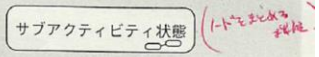
（7）スイムレーン
スイムレーンはアクション状態を実行する担当者を表す．アクティビティ図上では大きな長方形で記述する．スイムレーンにアクション状態を配置することで，その担当者が明確に表現できる．



msgオブジェクト内で分離可能



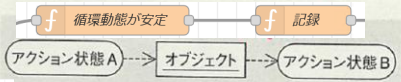
（8）サブアクティビティ状態
サブアクティビティ状態はその中にアクティビティ図が内包されていることを表す．アクティビティ図上では角の丸い長方形の右下にサブアクティビティ状態を表すアイコンを付記して記述する．



functionの集約可能

図8-9　サブアクティビティ状態

（9）オブジェクトフロー状態
オブジェクトフロー状態はアクション状態間で何らかのオブジェクト（情報等）の受け渡しが行われることを表す．アクティビティ図上ではアクション状態間にオブジェクトを表す長方形を配置し，破線の矢印でそれが受け渡される方向を記述する．



8-10　オブジェクトフロー状態

.0）ノート
ノートはモデルに対するコメントを表す．UMLの全てのアクティビティ図で使用できる．

図8-11　ノート

（11）割り込み可能アクティビティ領域（UMLバージョン2.0）
割り込み可能アクティビティ領域は，その領域内のアクティビティで特定のイベントが発生した場合，通常のフローとは別の処理（アクティビティ）へシグナルを送ることができる仕組みである．特定の状況を感知する範囲を，角の丸い破線の長方形で囲んで表現する．領域内部からのシグナルは，ジグザグの実線に矢印を付け，それを内部から外部に出して表す．

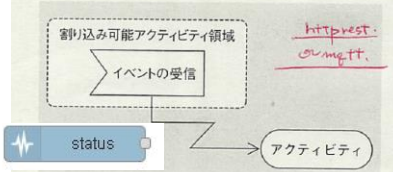


図8-12　割り込み可能アクティビティ領域

（12）コネクタ（UMLバージョン2.0）
コネクタは複雑なアクティビティ図をシンプルに整理するための要素である．コネクタは２つ１組で使用される．一方のコネクタに対して接続したフローを，もう一方のコネクタから再開できる．コネクタは丸の中にコネクタ名を記入して表す．コネクタ名は，主に「A」などのアルファベット１文字が使われる．

図8-13　コネクタ

必要機能に対応する要素があることを確認できた

Node-REDを用いた病院業務フローモデル

- https://gist.github.com/ryumtym/b091d78c7035328919fe2283e2b965b8

要素の解説

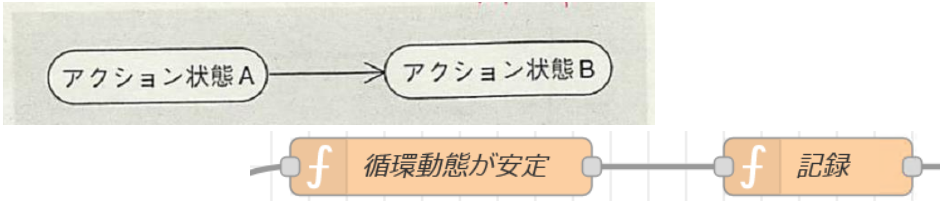
UML要素とノードの対応

(1)開始状態
開始状態は処理の開始を表す。
アクティビティ図上では黒で塗りつぶした丸で記述する。

(2)開始状態
終了状態は処理の終了を表す。
アクティビティ図上では白と黒の二重丸で記述する。

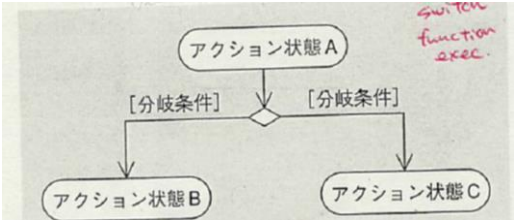
UML要素とノードの対応

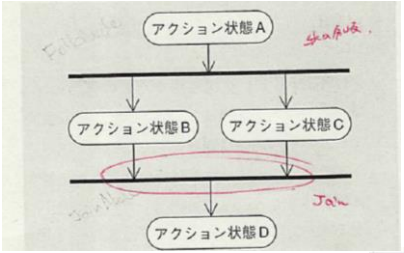
(3)アクション状態(アクティビティ)
アクション状態は何かの処理を行っている状態を表し、アクティビティとも呼ばれる。アクティビティ頭上では角の丸い長方形で記述する。

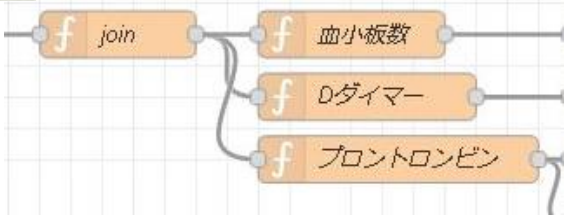


(4)遷移
遷移はある処理の状態から別の処理の状態へ遷移することを表す。アクティビティ頭上では遷移する方向の矢印で記述する。

UML要素とノードの対応



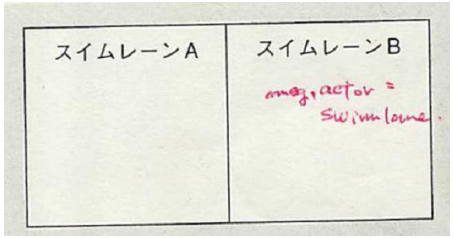


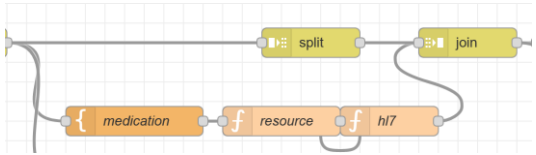


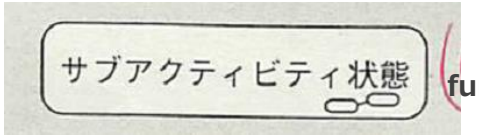
(5)分岐
分岐は何らかの条件によって変化する処理の流れを表す。アクティビティ図上では遷移の矢印に、[]をつけた分岐条件で表記する。なお、分岐のポイントはひし形で記述する。

(6)同期バー
同期バーは複数の処理が進行して行われる流れを表す。並行処理の開始を表す同期バーを「ジョイン」と呼ぶ。アクティビティ頭上では太枠で記述する。

外部ノードの利用







functionの集約可能

(7) スイムレーン

スイムレーンはアクション状態を実行する担当者を表す．アクティビティ図上では大きな長方形で記述する．スイムレーンにアクション状態を配置することで，その担当者が明確に表現できる．

(8) サブアクティビティ状態

サブアクティビティ状態はその中にアクティビティ図が内包されていることを表す．アクティビティ図上では角の丸い長方形の右下にサブアクティビティ状態を表すアイコンを付記して記述する．

FHIRサーバとしての利用

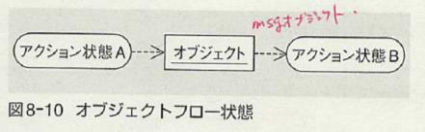
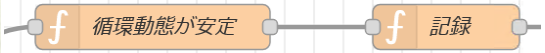
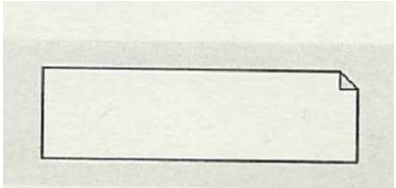


図8-10 オブジェクトフロー状態



(9) オブジェクトフロー状態

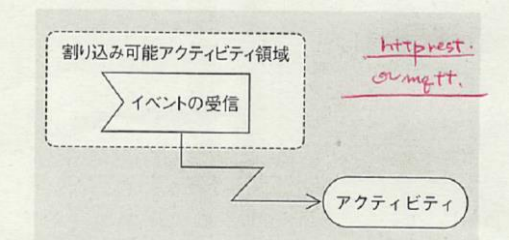
オブジェクトフロー状態はアクション状態間で何らかのオブジェクト（情報等）の受け渡しが行われることを表す．アクティビティ図上ではアクション状態間にオブジェクトを表す長方形を配置し，破線の矢印でそれが受け渡される方向を記述する．



(10) ノート

ノートはモデルに対するコメントを表す．UMLの全てのアクティビティ図で使用できる．

Node-RED起動法



(11) 割り込み可能アクティビティ領域（UML バージョン 2.0）

割り込み可能アクティビティ領域は，その領域内のアクティビティで特定のイベントが発生した場合，通常のフローとは別の処理（アクティビティ）へシグナルを送ることができる仕組みである．特定の状況を感知する範囲を，角の丸い破線の長方形で囲んで表現する．領域内部からのシグナルは，ジグザグの実線に矢印を付け，それを内部から外部に出して表す．

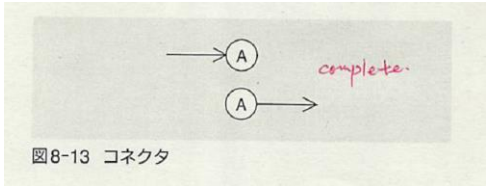


図8-13 コネクタ

(12) コネクタ（UML バージョン 2.0）

コネクタは複雑なアクティビティ図をシンプルに整理するための要素である．コネクタは２つ１組で使用される．一方のコネクタに対して接続したフローを，もう一方のコネクタから再開できる．コネクタは丸の中にコネクタ名を記入して表す．コネクタ名は，主に「A」などのアルファベット１文字が使われる．

Node-REDフロー内での
msgオブジェクトの取り扱い

msgオブジェクト概要

- フローの実行に伴ってmsgオブジェクトが生成される
- msgオブジェクトは平たく言えば、JavaScriptの変数と同じ取り扱いができる

- M言語風に言えば「label：value」の形式

- Propertyの追加

- Propertyの編集

- 配列の利用

- 連想配列の利用

Node-RED OSSフローの利用

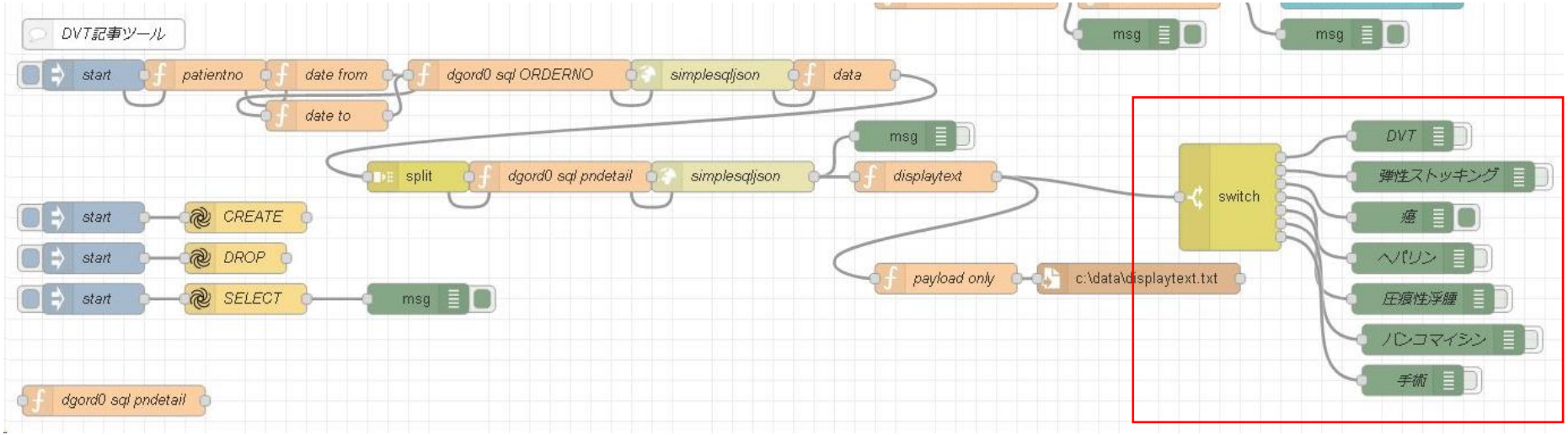
Node-RED起動法

- ターミナルよりnode-redとタイプ

Node-REDフロー組み立て例

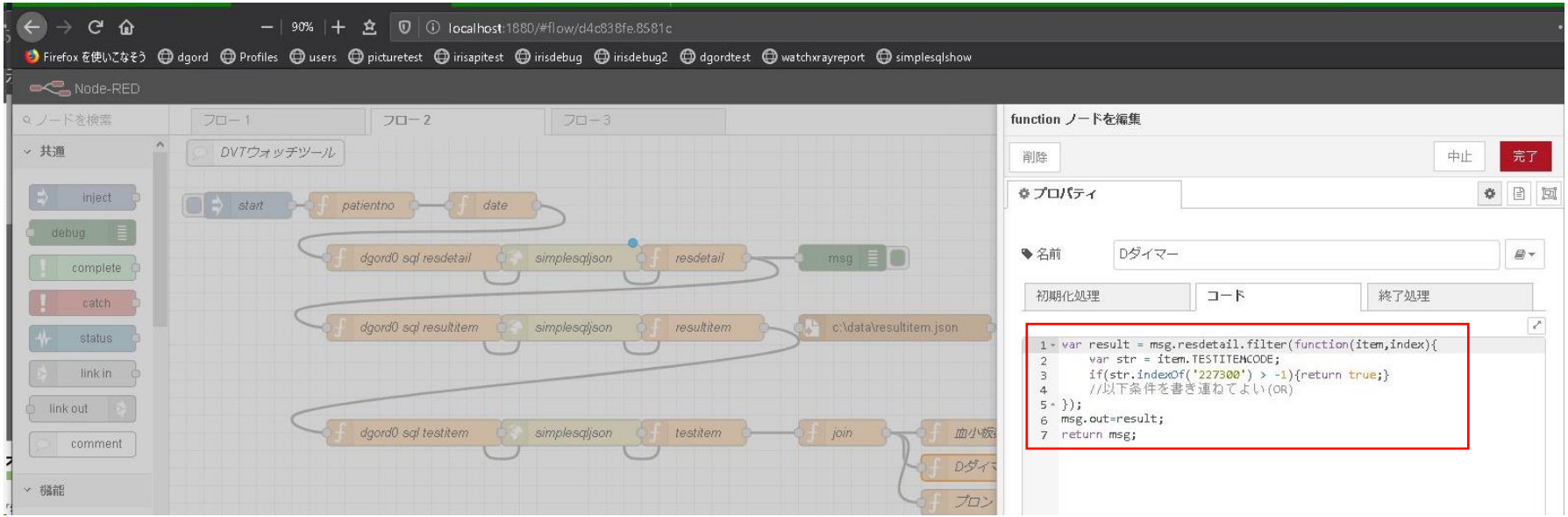
記事中のキーワードを用いたスイッチアクション

患者ID、期間をもとに診察記事を抽出し、含まれるキーワードによってフラグを複数生成するデモ
このようなコード体系では、運用後のロジック追加の際に機能全体の稼働保証を憂慮せずに済むメリットが大きい



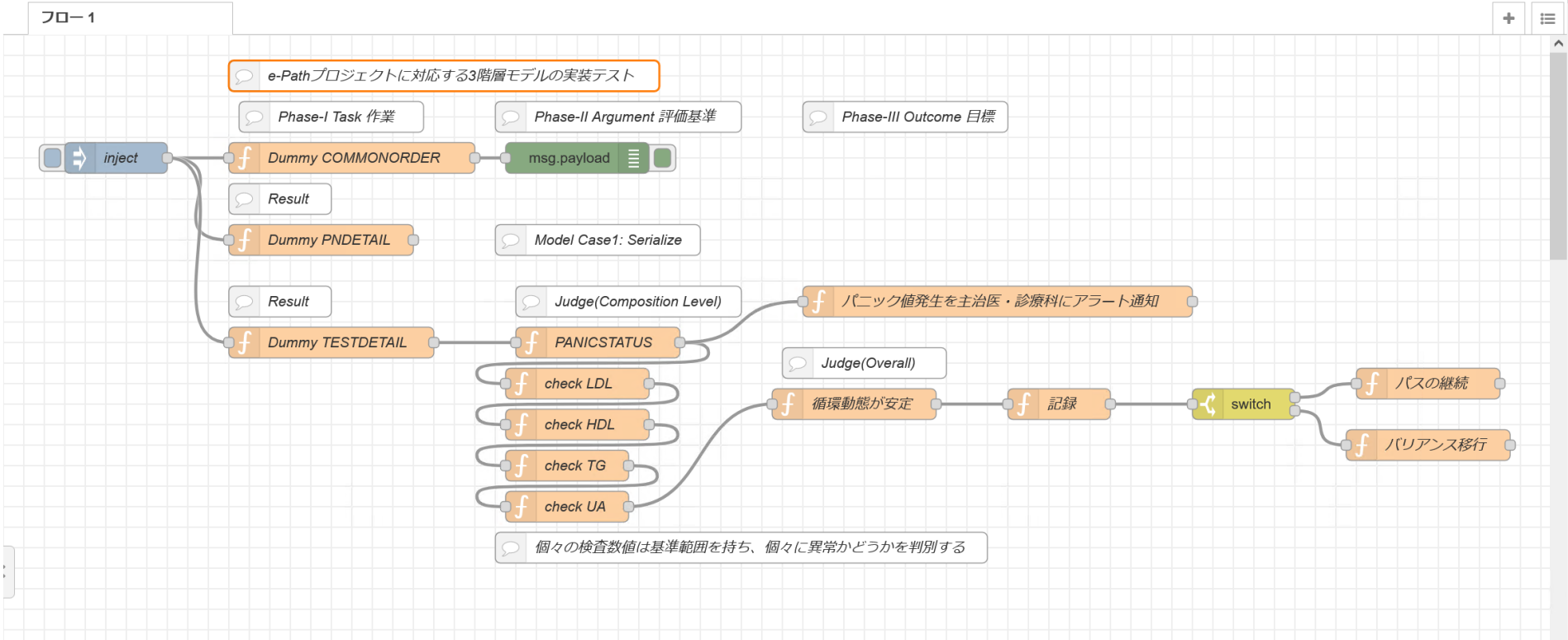
結果一覧から目的の検査項目を抽出する方法

検査結果一覧から特定の項目（ここではDダイマー)をフィルタする書き方



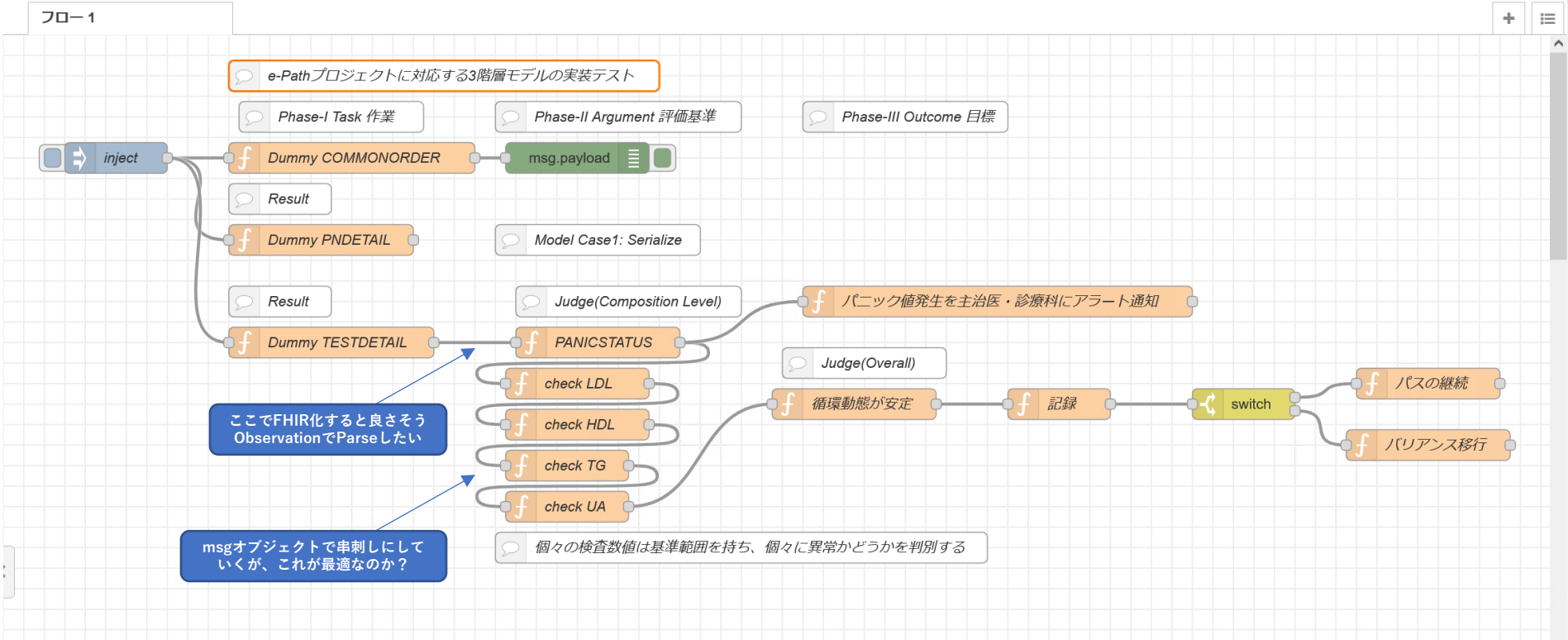
IPCIアーキテクチャによるe-Path3階層の実装テスト

NEC MegaOAKHRとのオーダリングDB接続を実現



IPCIアーキテクチャによるe-Path3階層の実装テスト

NEC MegaOAKHRとのオーダリングDB接続を実現

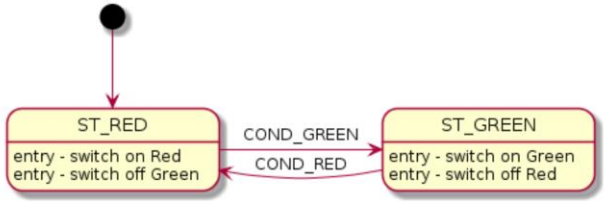


Node-REDとUMLに特徴的なFinite State Model(FSM)の実装性

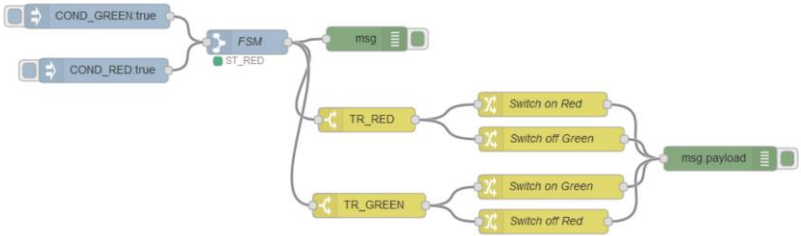
Example

Simple pedestrian light with manual trigger

The following example demonstrates the FSM node for a simple pedestrian traffic light state machine. It has two states ST_RED and ST_GREEN, where the related light gets switched on. The ST_RED state is entered at statup. The transitions between the two states are triggered by COND_GREEN and COND_RED repectively.



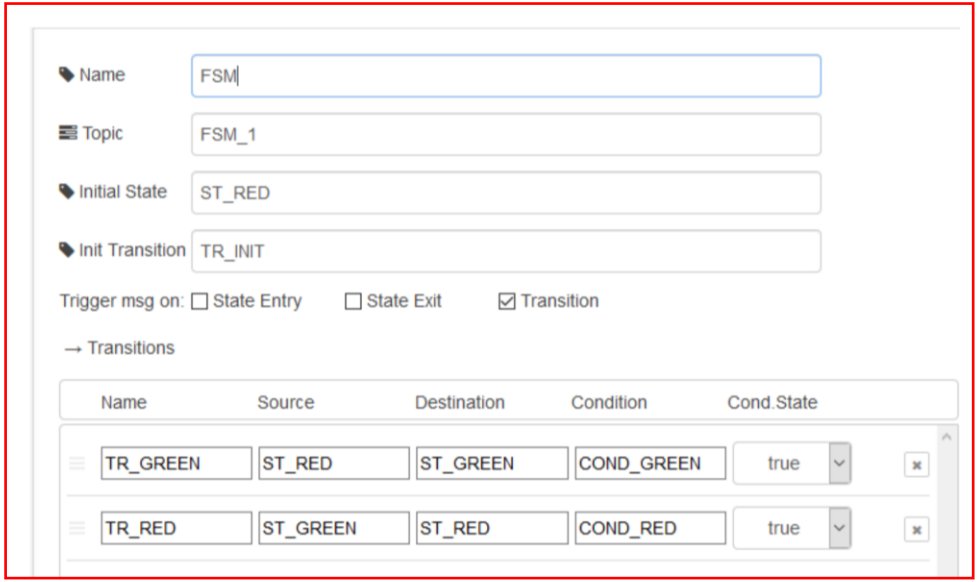
The flow in node-red:



The switch nodes filter the result message of FSM for the entry event of the states and the change node generates the final action of switching the traffic lights.

The exported flow can be downloaded from here: FLOW.

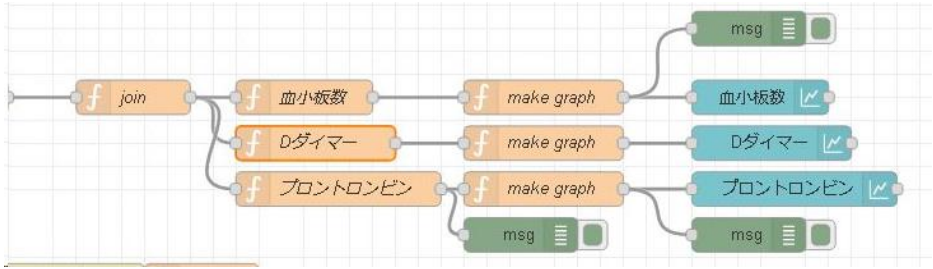
The FSM node is configured as shown in the UML state diagram.



UML-FSMのストレートな記述性を確認できた

Dashboard機能

Node-RED上に実装されている機能



グラフテンプレートとデータ構造の記述の仕方

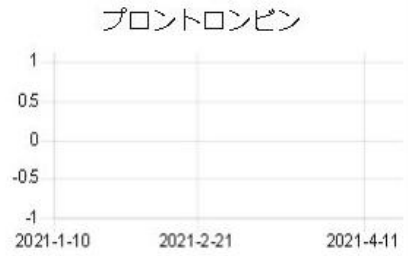
名前 make graph

初期化処理 | コード | 終了処理

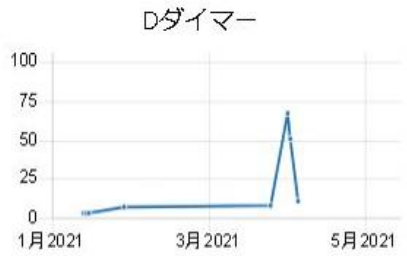
```
var temp = [{
    "series": ["prontronbin"],
    "data":[[]],
    "labels": [""]
}];
for(var i = 0; i < msg.out.length; i++){
    var plotdata = {"x" : msg.out[i].UPDATEDATE, "y" : parseFloat(msg.out[i].TESTRESULT)};
    temp[0].data[0][i] = plotdata;
}

msg.payload = {};
msg.payload = temp;
return msg;
```

DVT検査指標

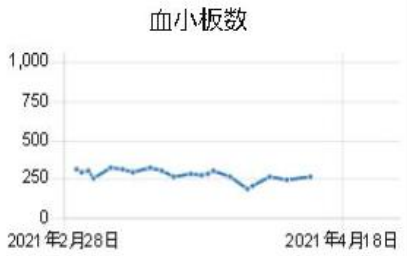
プロントロンビン



Dダイマー

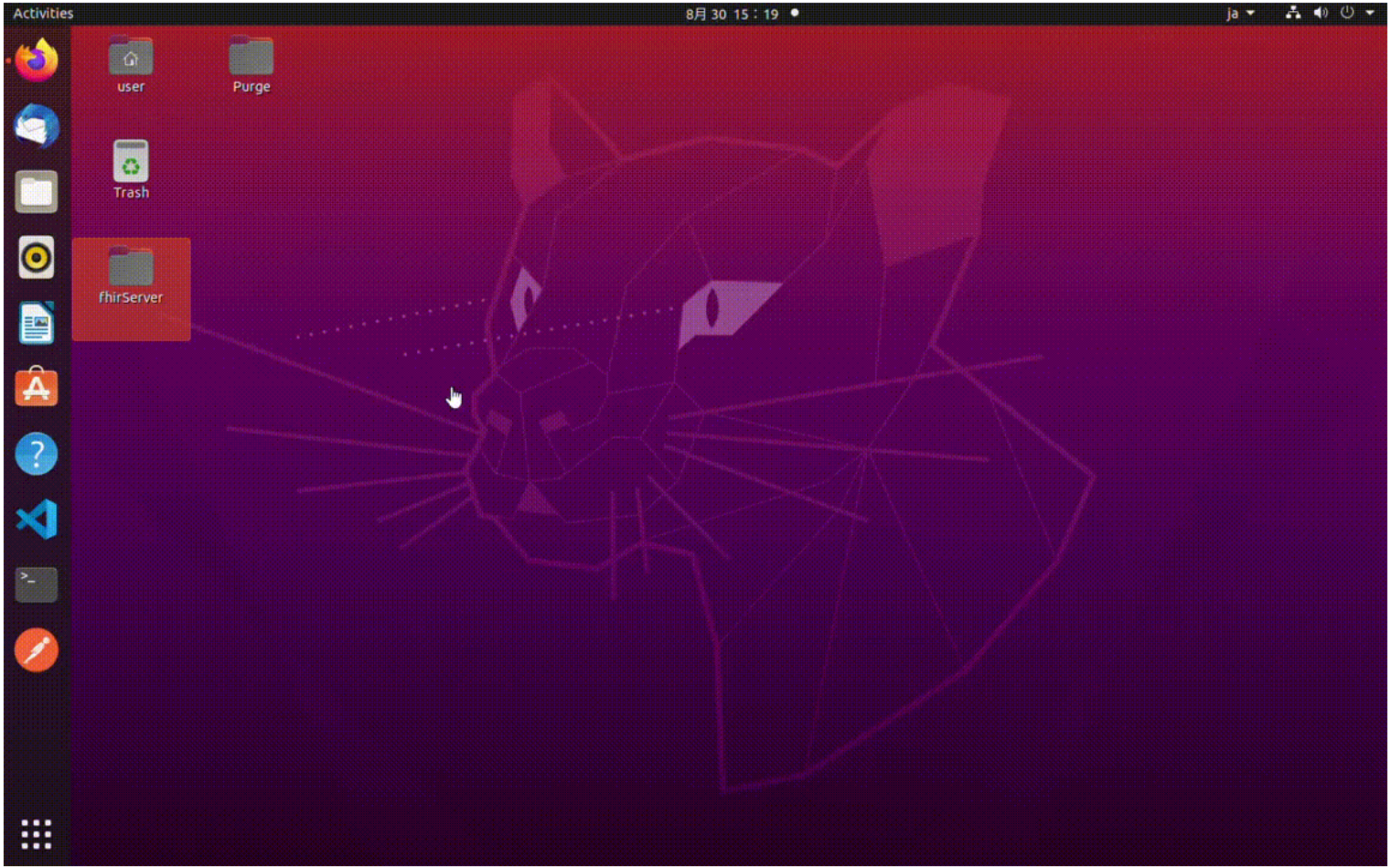


血小板数



IPCI-FHIRサーバの起動と利用

FHIRサーバ機能とPatientリソースの起動



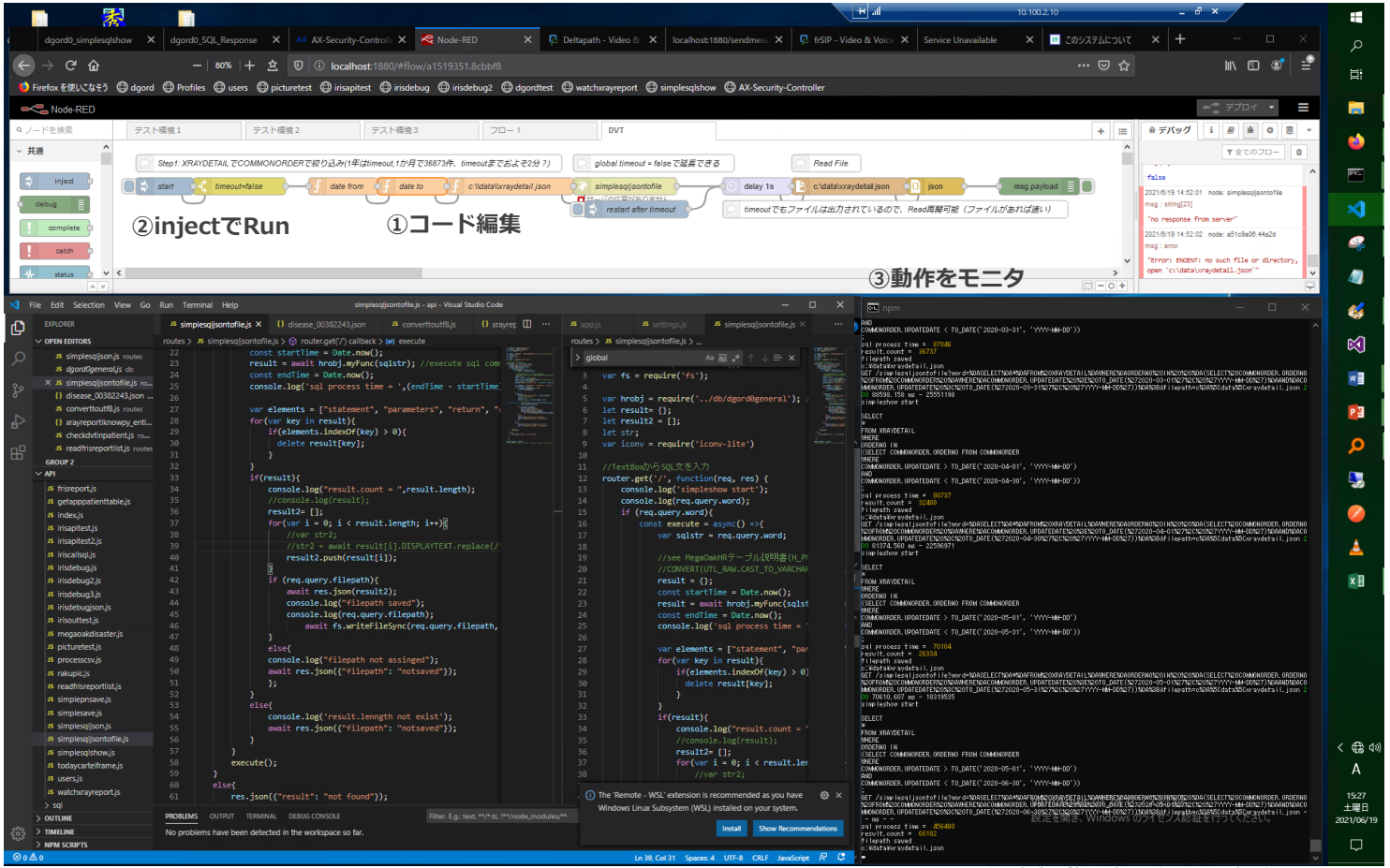
VSCodeを用いたNode.jsコーディング

IPCIサーバ 開発系画面の一例

Node-RED

VisualCode

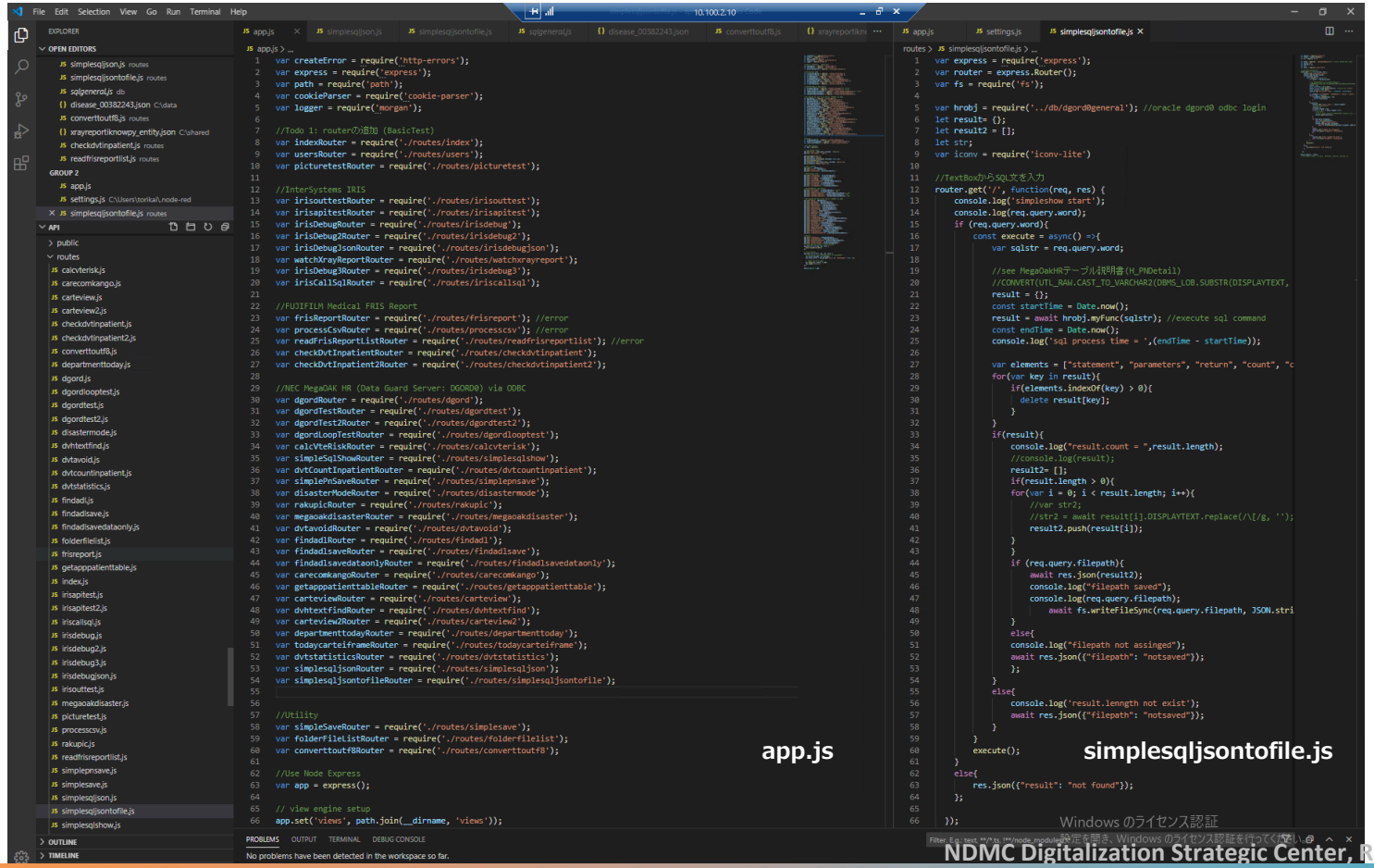
Node.js
npm



VisualStudioCode

var命名規則:route名+Routerで統一

route一覧



操作が理解できた方は
周りの方の操作支援をお願い致します

解説：
病院情報システム内でIPCIを利用する
セットアップ・設定の検討

医療情報学の永続性を妨げる要素

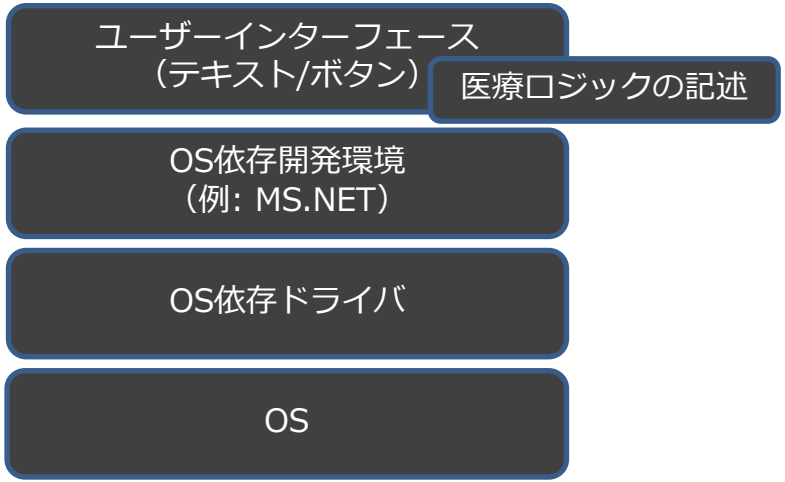
医療情報学はなぜこうも「落ち着かない」感じがするのか？

課題点		対応策
低い耐久性/高い独自性	デバイス / アプリケーション / OS	エッジ側規格化(Linux)
（特に無線）方式の変化	ネットワーク	802.11系
UIのOS依存	アプリケーション	Web UI
下位互換性の喪失	OS	仮想化/コンテナ化
演算能力	CPU/メモリ	メニーコア化/専用回路
スピンドル脆弱性	ストレージ	半導体ストレージ

どの領域にも「耐久性」の主張が困難 - 維持費ショート問題

IPCIコンセプト導入による開発ワークフローの変化

- **既存型**



この状況で、OSのバージョンアップが起きると、UIが追従できなくなると同時に、医療スタッフからヒアリングした（普遍性の高い）医療ロジックが継承できなくなり、喪失リスクが生じる

可能な反論：
「医療ロジックだけをUIと別にコーディングしておけばよい」（Doc-View概念に忠実な実装）

だがUI自体が「医療における安全性ノウハウ」を同時に提供している点において、医療ロジックはUIにも埋め込まれている（ここで医療ロジックとUIの分離あいまい性が生じる）

また現場での「迅速な」実装を求められた際、Doc-Viewはリファクタが困難であった→オブジェクト指向自身の性質に由来

医療機器現場では、さらに「ハードウェアと通信するドライバのOS依存と、記述できるエンジニアの長期雇用の課題」を内包する

IPCIコンセプト導入による開発ワークフローの変化

- **導入後**

Web化は「UIのクロスプラットフォーム」をもたらした
→UI自身がOS非依存になり、耐久性に貢献する

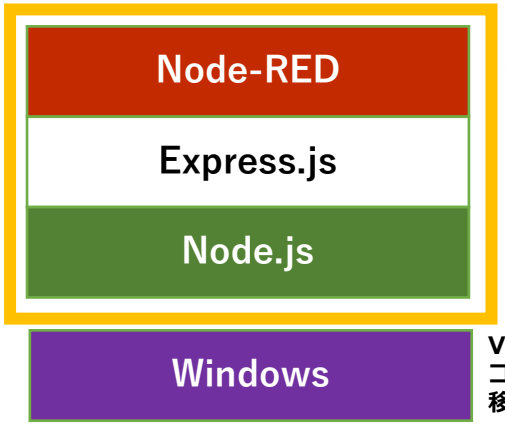
医療ロジック自身を「高速なスクリプト言語」かつ
「リファクタが容易になる言語」で記述する
→Node.js
(サーバサイドJavaScript+プロトタイプベースオブジェクト指向)
を活かすことで、耐久性とリファクタ性に貢献する

機器との通信部分はOS以上をすべてコンテナ化することで、ドライバの稼働をOSを更新せずに担保することで、耐久性に貢献する（ただし適切なセキュリティでIOを包むことが必須）

OSS(特にOpenJS系)の特徴と開発方針

・基本はスクリプト言語である→ソースコードレベルで移植性が高い
・ソフト間の関係性はNode.js+Expressフレームワークに準じている
・個人が開発したソースは検証が多くされているものとそうでないものが混在、見分ける必要あり
・言語仕様が変わらない、また必要機能が増えない限り遡及した保守は不要
・レスポンスとしては低一中速まではRESTでよい、それ以上は単一サーバ内に集約する

・これまで推進してきたNode.js+Expressの医療ワークフローとして、同じJavaScript上で完全に（字義通り）統合される



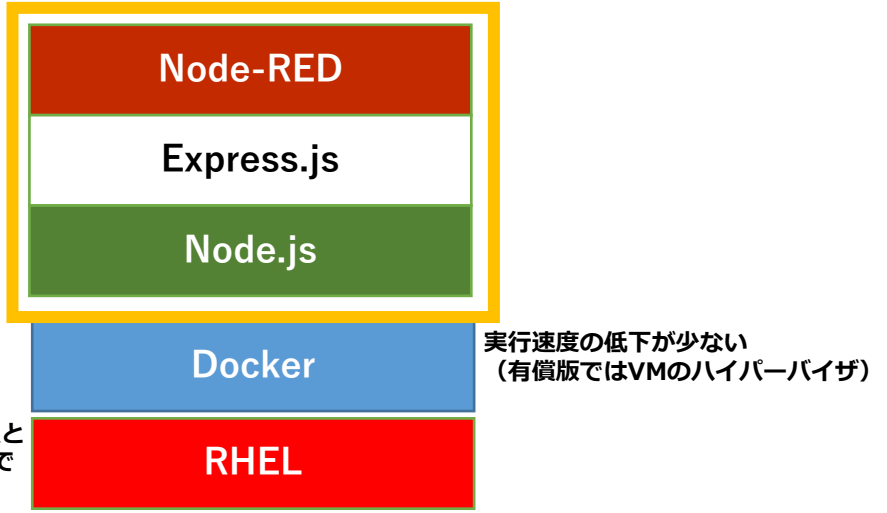
パッケージとして考える

医療ロジックの記述箇所

RESTライブラリを構成

V8エンジンの上で動いているプログラムとコンパイラ互換のコードで構成することで移植性を担保している

RESTサービス全体（サーバ間通信設定）まで保存したい場合に
Docker（必要に応じてK8）を使う→プログラムの永続性

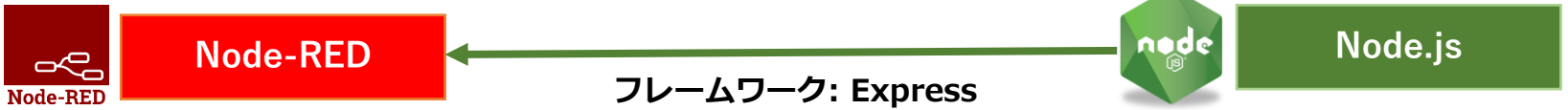


実行速度の低下が少ない
（有償版ではVMのハイパーバイザ）

求める効果：「医療ロジックの永続化」→共通クリニカルパス策定のプラットフォーム、シームレスなIoT/AI連携が見込まれる

Node.jsとNode-REDの開発階層

役割の分担



GUIの要素：Dashboard
GUIの要素：イベントハンドラ
FHIRプロパティでmsgフローを作成したユニバーサルな
医療ロジック・クリニカルパスの蓄積
運用上変更の多い層
→msgオブジェクトでブロック化

スクリプトベースの処理担当

GUIを要しないREST処理
まとまった処理、あまり変更されない層

いずれの処理においても、共通して使用する言語はJavaScript(ECMAScript2015以降,JS)で一貫している
言語選択の重要性：

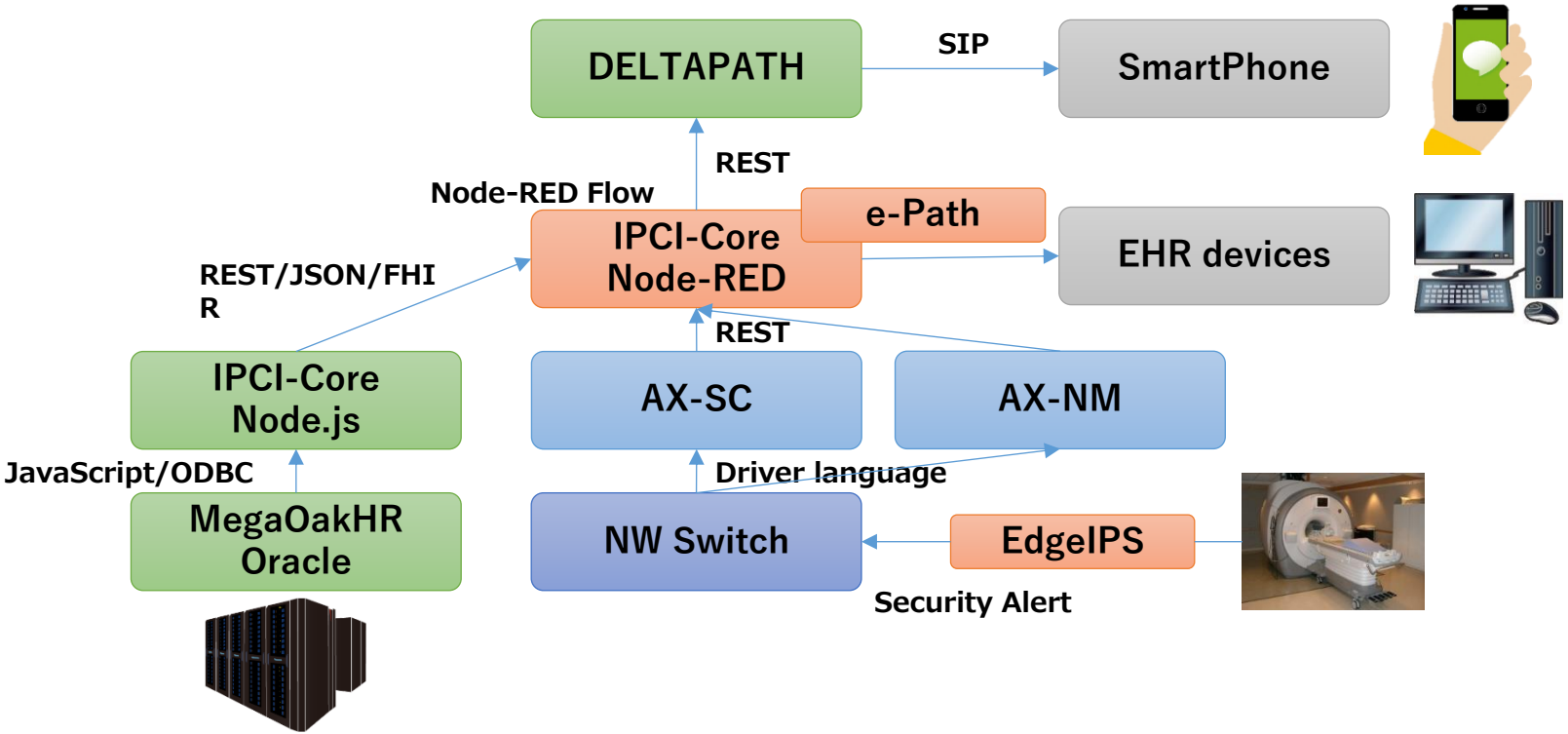
PythonではJSONネイティブな書き方が存在するが、UIの生成については別途HTML/CSS/JavaScriptを組むことになる→処理文法が増える欠点あり

重要な提案は、Node-RED内の医療ロジックを”FHIR準拠形式”で記述していくことで、コードブロック単体での再利用性まで高める

JSON形式の高速データベースの必要性→Mテクノロジー学会主体で開発（Mアーキテクチャ+ObjectScript）、ただしその他のデータベース形式にも広く接続性を担保するような活動にすること

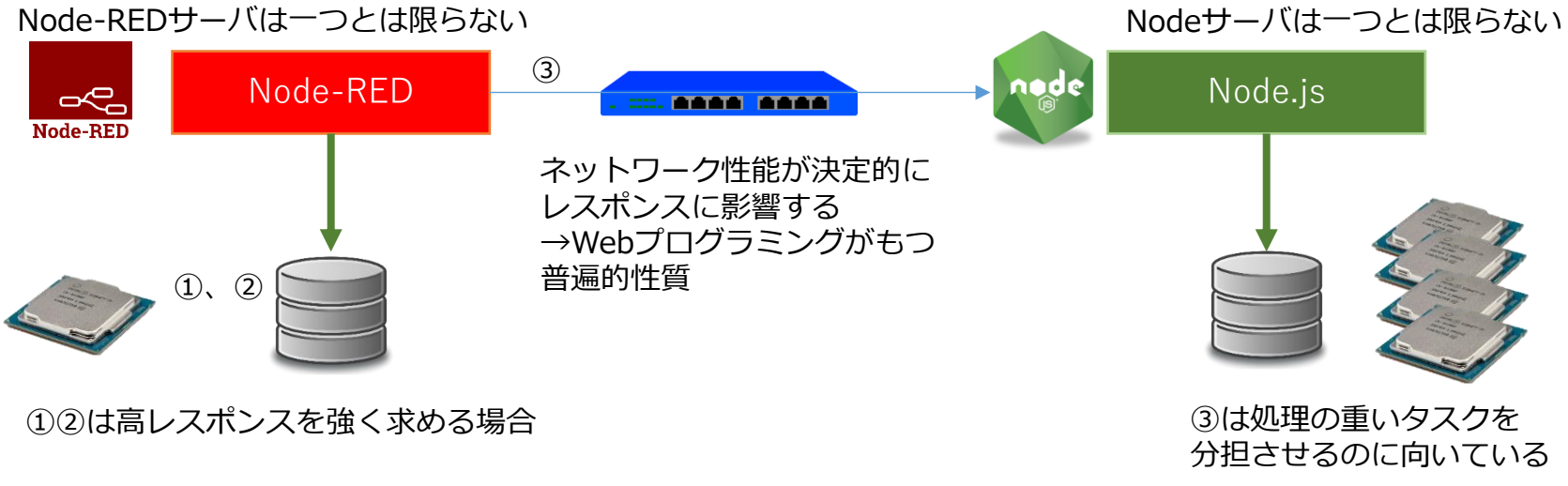
JSはスクリプト式でコンパイラよりは演算処理が遅い→コンパイラプログラムをサーバのどこに位置付けるかを定めるハードウェアを含む演算速度の増強が不可欠

手段：IPCIを中核とするロジックシステム・ロジック接続形態の設計



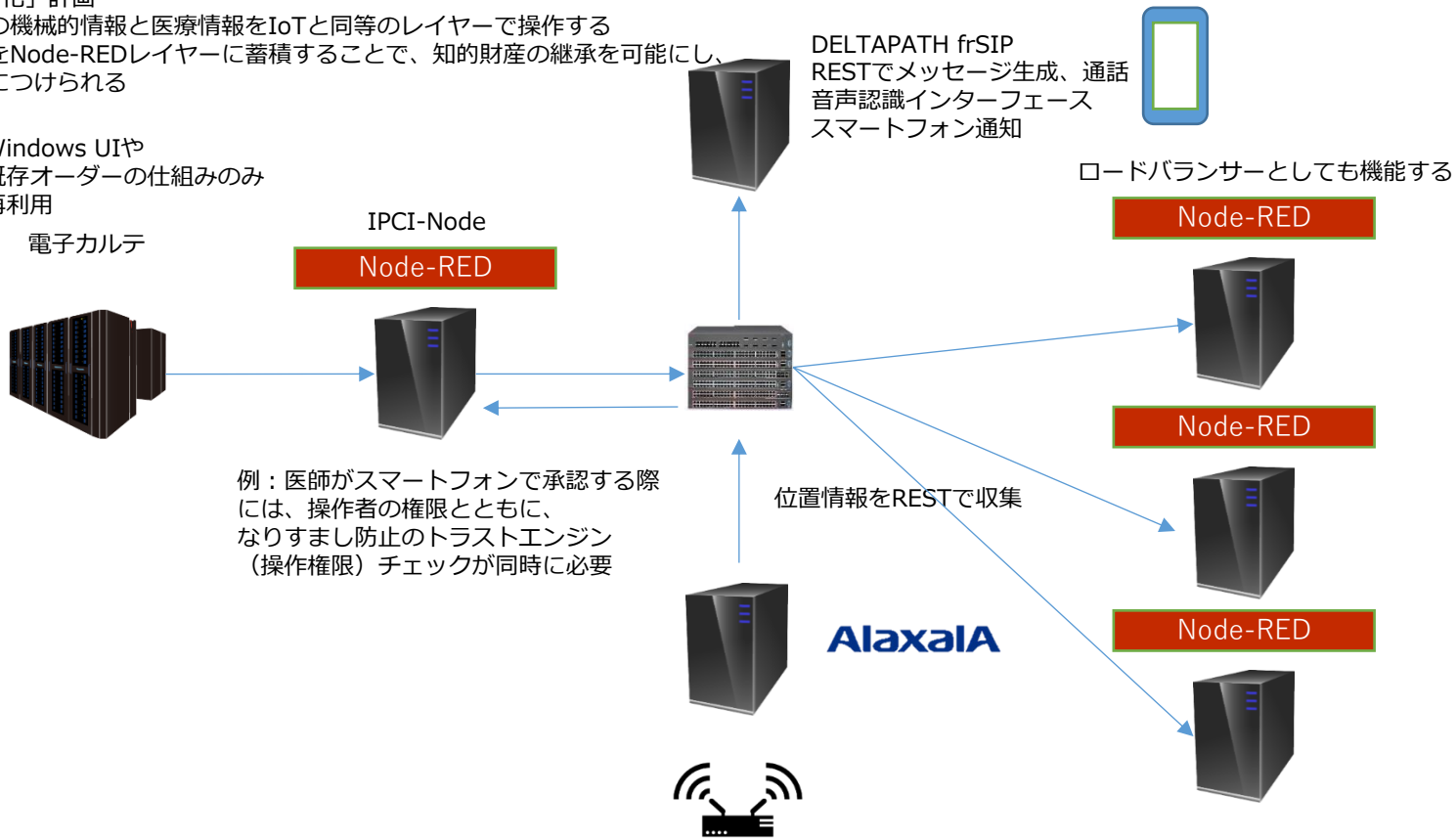
実装上の工夫

Node-REDではrequireでnode_moduleを直接呼び出すコードはfunctionに書けない
対処法
①requireを含むスクリプトをフローオブジェクト化
②requireを含むスクリプトをコマンドフローから呼ぶ
（ただし戻り値処理が少しややこしい）
③RESTでラップしてNode.js-Expressへ投げ、戻り値をJSONオブジェクトとして受ける
（記述はしやすいが、RESTコールはそれほど高速でない）



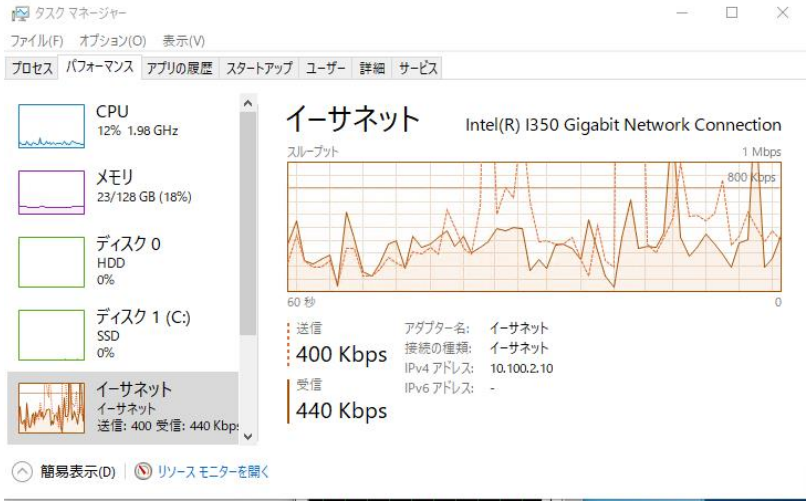
IPCI-Nodeオーケストレーション

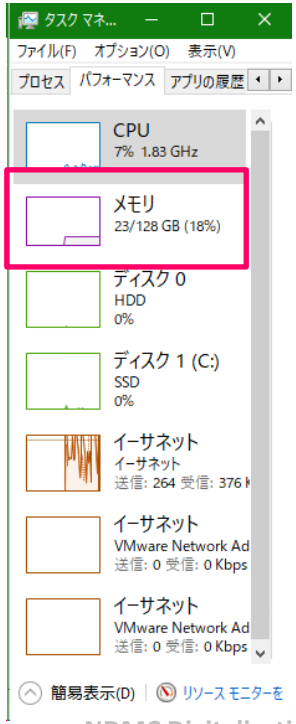
「病院まるごとIoT化」計画
個々のサーバからの機械的情報と医療情報をIoTと同等のレイヤーで操作する
医療ロジック地震をNode-REDレイヤーに蓄積することで、知的財産の継承を可能にし、
時間の経過を味方につけられる

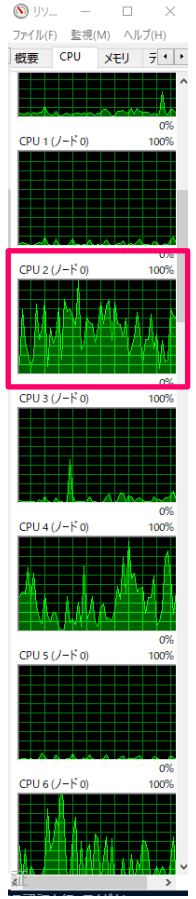


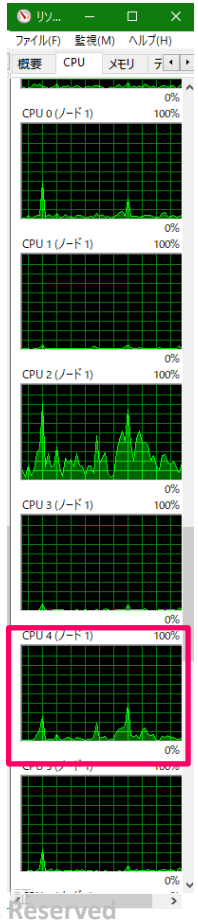
IPCIに適したCPU/メモリ/ストレージ/ネットワークの考え方

- SUPERMICROでIPCI構成をテストした際の検討(物理16コア)
- 非同期処理を行うために、複数コアは必要である
- しかし実例では4コア程度に負荷が集中している
- →動作周波数が高く（できれば常時5GHz）、かつ4－６コア程度のマシンが適切と考える
- メモリは23GB程度であり、最小構成でも32GBあればOK
- ディスク容量は数TBで、増設を可能とする構成が望ましい
- 安価に抑えるなら10Gbps-NASなど
- Thunderboltは高速だが技術的に枯れていない？
- テキストはファイルサイズが小さい→速度より遅延のほうが大きく影響







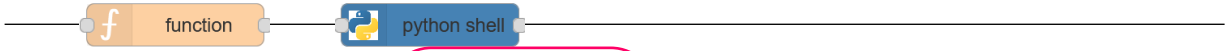


検討：CUI/GUI/Flowプログラミング、それぞれの利点

アプローチ	CUI	GUI	Flow
言語例	C++ bash	Windows Form マウス操作/RPA	LabView Node-RED
命令単位	テキストファイル	ユーザー操作によるロガー	ノード
コード独立性	完全に独立	汎用化には手入れが必要	フレームワーク上での独立性
シーケンス記述	OK	RPA自身の制御は平易とは言えない	OK
パラレル記述	コード記述にスキルが必要	マルチタスク性という意味ではよい	OK
オブジェクトマニピュレーション	キーボード操作と画面切り替えが煩雑	視覚的で効率よい	ノード単位で可能
データフロー	同一コードに記述すると分かりにくい	ユーザー自身で都度判断	分岐が容易
利用例	数学アルゴリズム的 規則単位の多用	繰り返しが少ない処理（ファイル整理）	処理が分岐する場合 将来変更が多い場合

検討：Node-REDとPythonモジュールの分担について

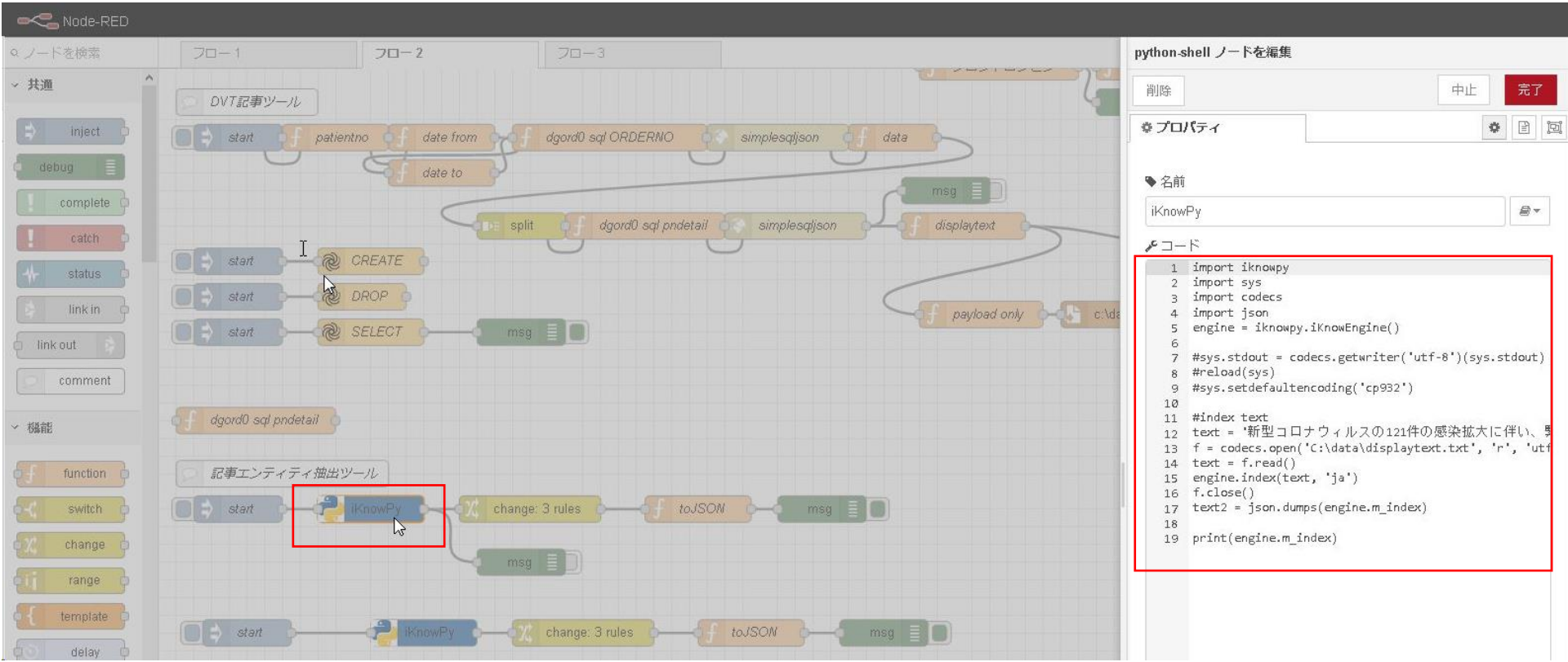
各モジュールに分担する機能と理由の検討



	JavaScript	Python	exec	VBA	VBScript	各社独自スクリプト
タイプ	スクリプト	スクリプト	スクリプト	スクリプト	スクリプト	スクリプト
動作ベース	Node.js	Python	MS-DOS	Excel	MS-DOS	各ソフト上
ハードウェアアクセス	可	可	可	可	可	可
ML	△	○	×	×	×	（さまざま）
JSON	○	○	×？	△	？	（さまざま）
ロジックGUI	Node-RED	×	×	デバッグ容易	×	独自アプリ上
マルチOS	○	○	×(bashは近い)	×	×	（さまざま）
中核サーバ	Express	http.server Flask	×	×	×	（さまざま）
単体アプリケーション	Electron Angular	Kivy Tkinter PyQt wxPython PySimpleGUI →PyInstaller	.bat MS-DOS窓で可能	VBへの移植以外不可 （画面表示をさせずに実行することは可能）	.vbs	（あまり聞いたことがない）
単体App作成効率	△（Node.jsスクリプトレベルならPythonと効率はそれなり）	○	△	○	△	×（独自言語文法の習得が必要）
WebApp作成効率	○ （但し単体Appより低い）	△ (HTMLを書くことになる)	×	△-× (HTMLを書くことになる)	×	（さまざま）

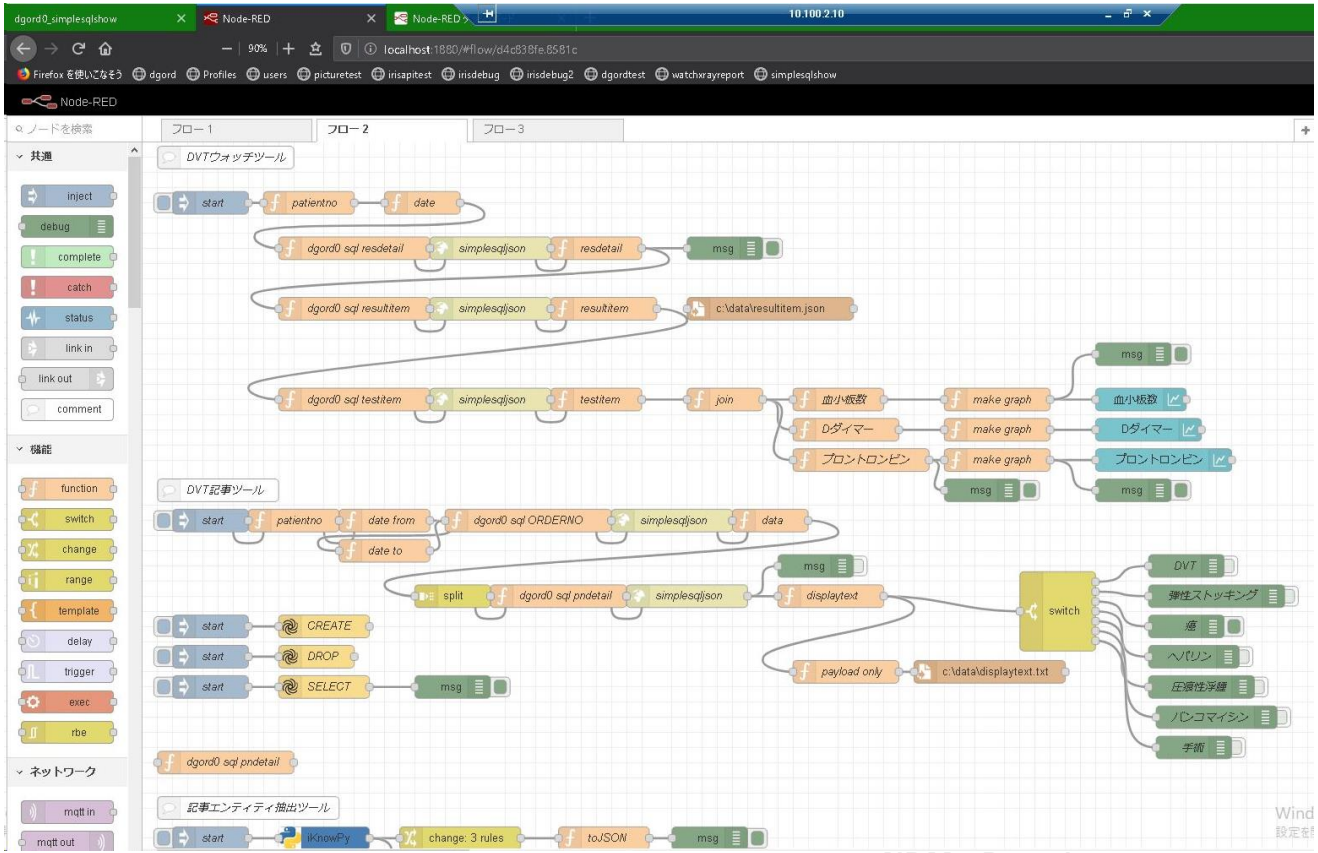
プロトタイプベースオブジェクト指向の特長

InterSystemsがGitで公開しているiKnowPyを呼び出し（ファイルで渡し、結果はmsg.payloadで戻せる）
将来的な機械学習機能の実装にも対応できる



フロープログラミングの特徴

医療ロジックや複数の情報要素を束ねる際に有効な手法
コーディングでは表現しづらい並列処理や、変換ロジックだけのモジュール化などが容易

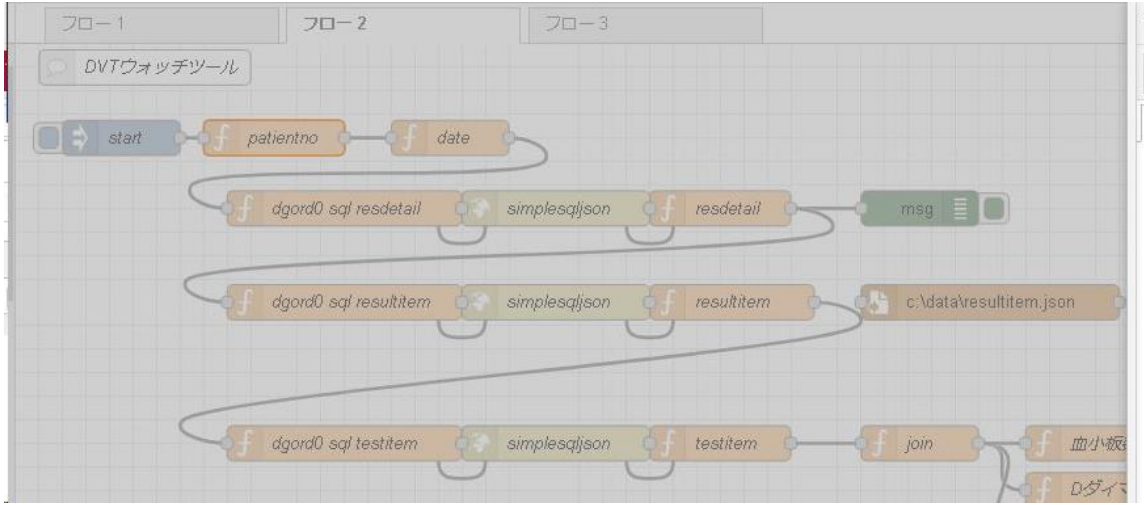


プロトタイプオブジェクト指向の特長

動的に要素を定義、追加、編集、削除できる

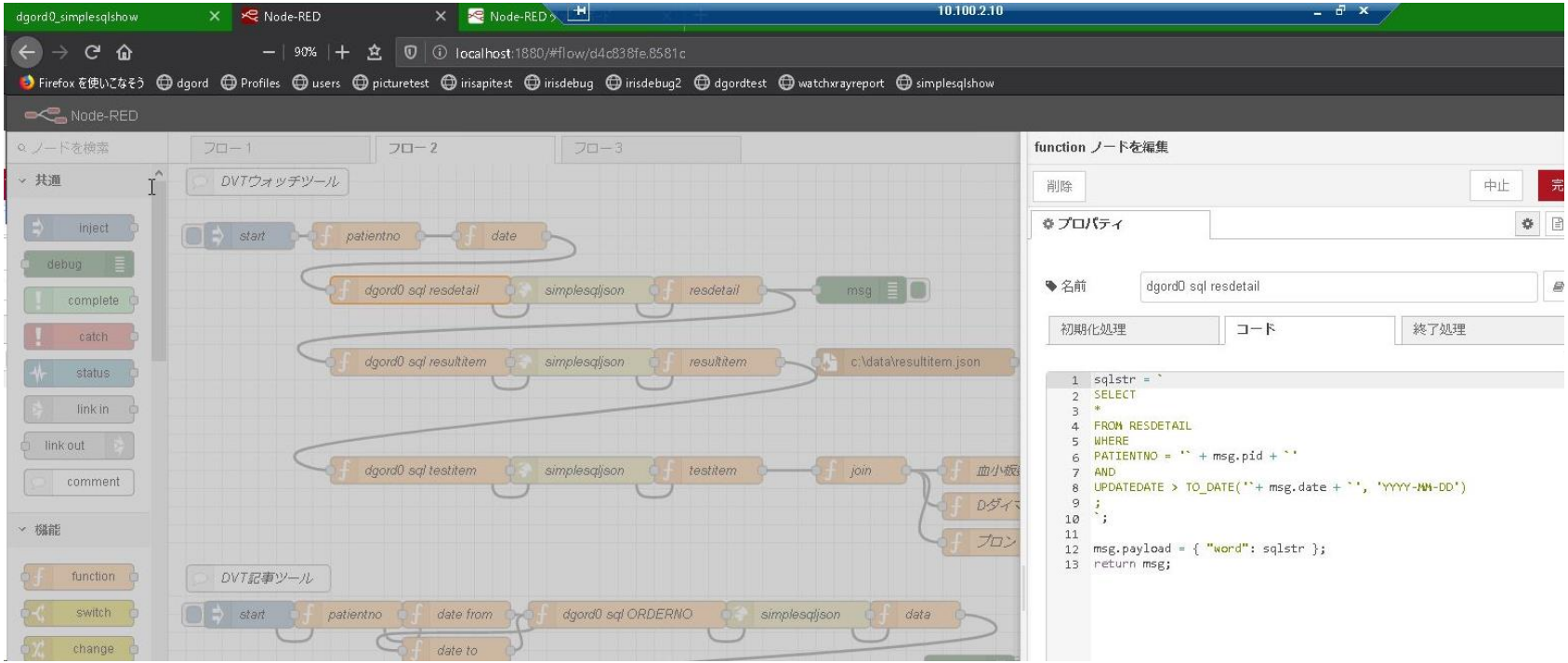
Node-REDの中ではmsgオブジェクトを（多くはmsg.payload）使ってメッセージ処理を行う

例：患者ID、期間を指定して、検査結果をMegaOAK HRのDBから抽出し、Dダイマー等のフィルタで分類し、これらをグラフに表示するデモ



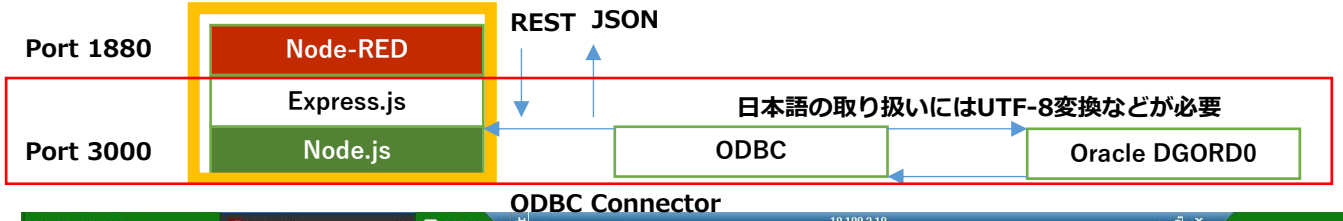
SQLのみをロジック化する手法

msg.payload中にSQL構文を記述

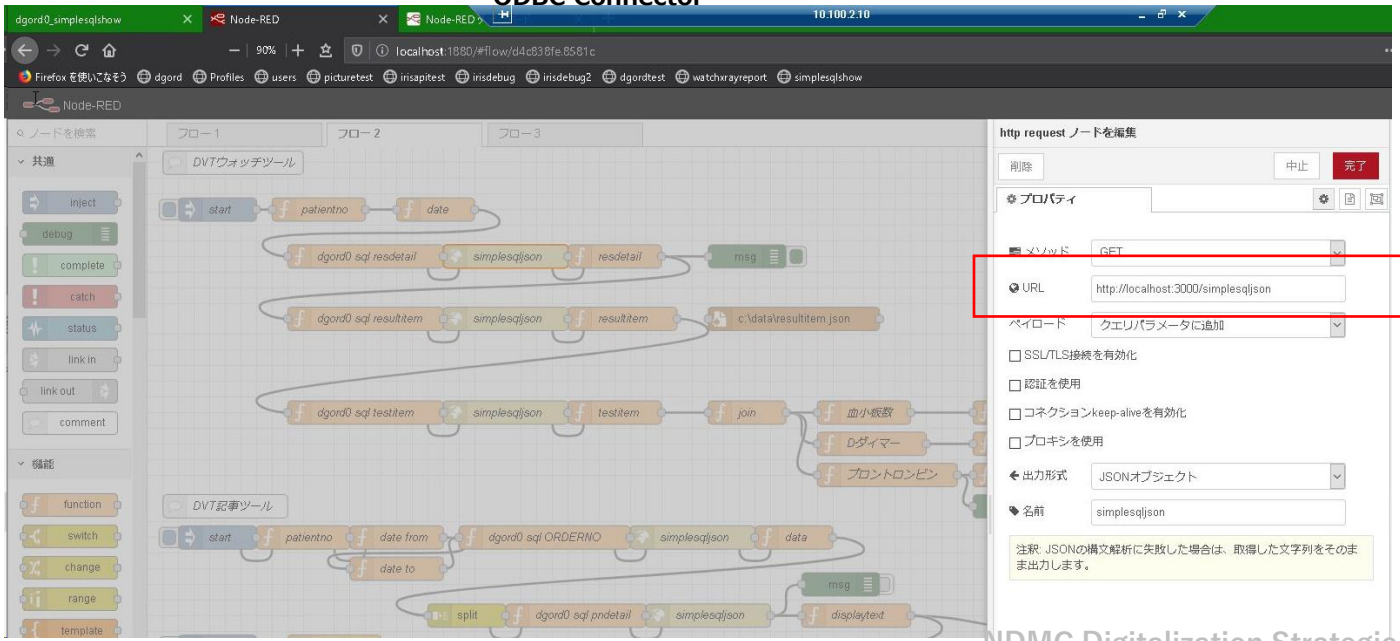


ODBCラッパーとDB接続

Node.js + Expressで作成したREST-ODBCラッパーにSQL構文を渡し、REST/JSONで戻す
同一筐体内にポートの異なる2つのサーバがある状態

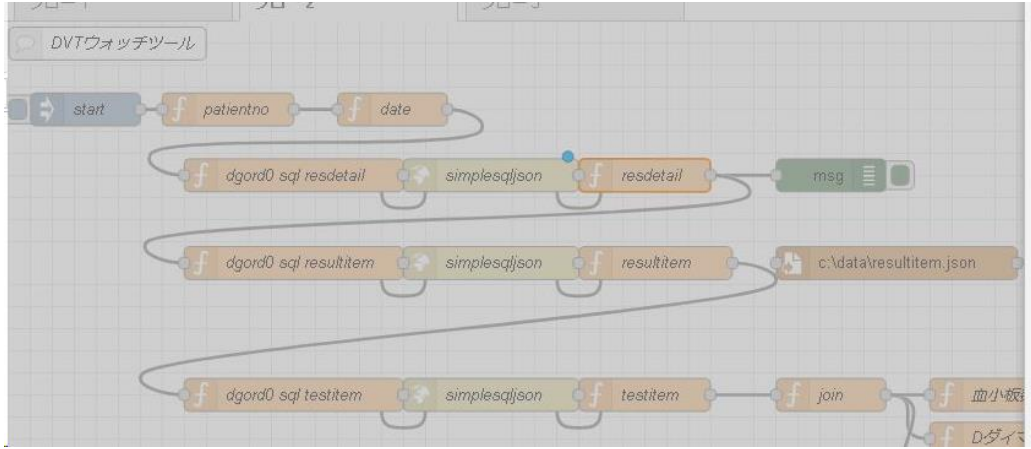


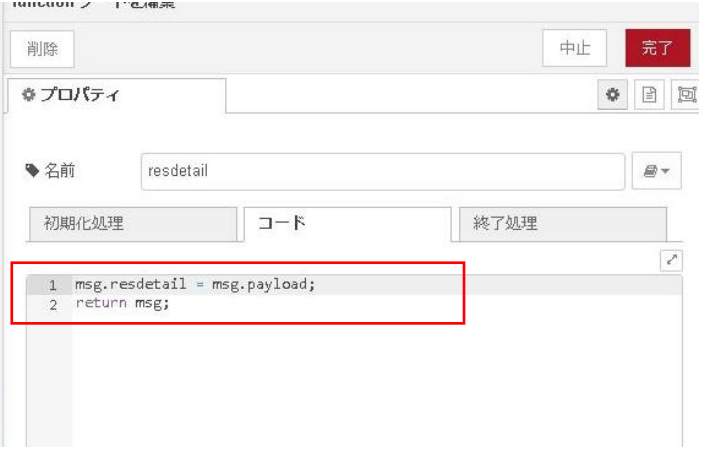
これまで開発してきた
要素技術部分



msg.resdetailの生成

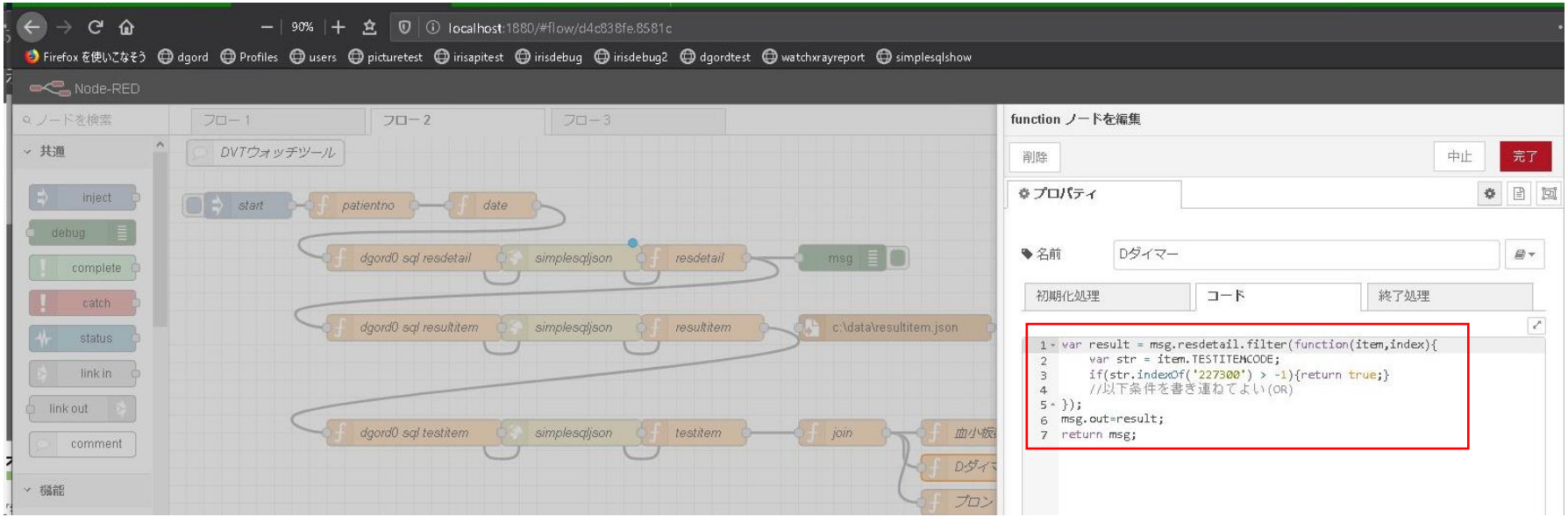
msg.resdetailにオブジェクトを載せ替え





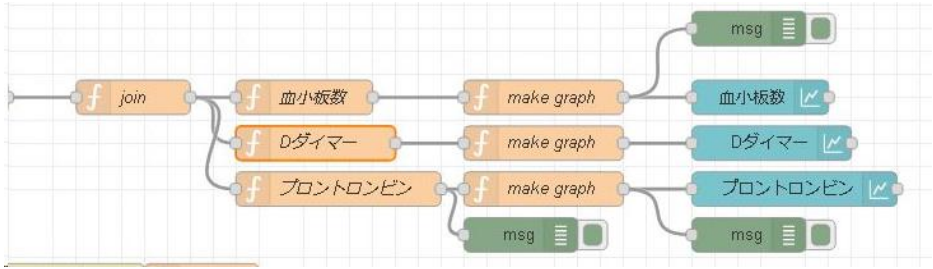
結果一覧から目的の検査項目を抽出する方法

検査結果一覧から特定の項目（ここではDダイマー)をフィルタする書き方



Dashboard機能

Node-RED上に実装されている機能



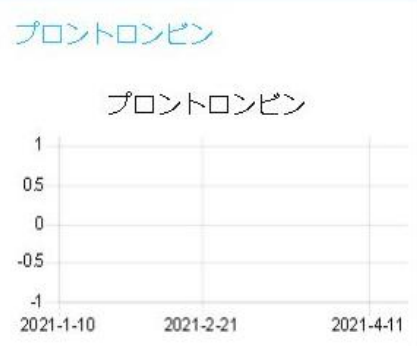
グラフテンプレートとデータ構造の記述の仕方

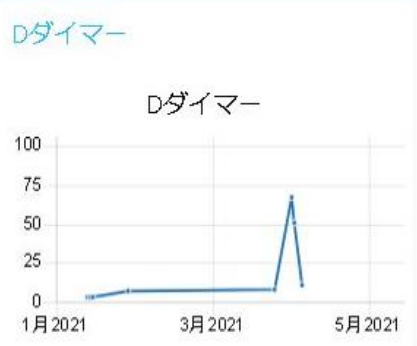
名前 make graph

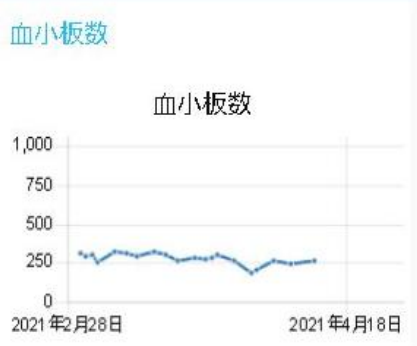
初期化処理 | コード | 終了処理

```
var temp = [{
    "series": ["prontronbin"],
    "data":[[]],
    "labels": [""]
}];
for(var i = 0; i < msg.out.length; i++){
    var plotdata = {"x" : msg.out[i].UPDATEDATE, "y" : parseFloat(msg.out[i].TESTRESULT)};
    temp[0].data[0][i] = plotdata;
}

msg.payload = {};
msg.payload = temp;
return msg;
```

DVT検査指標

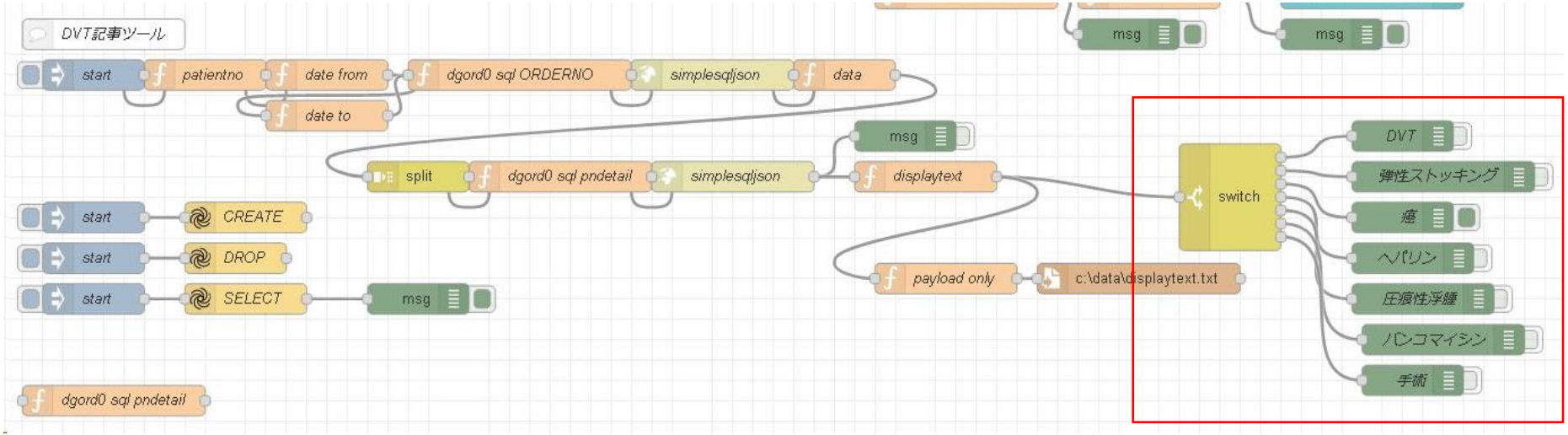






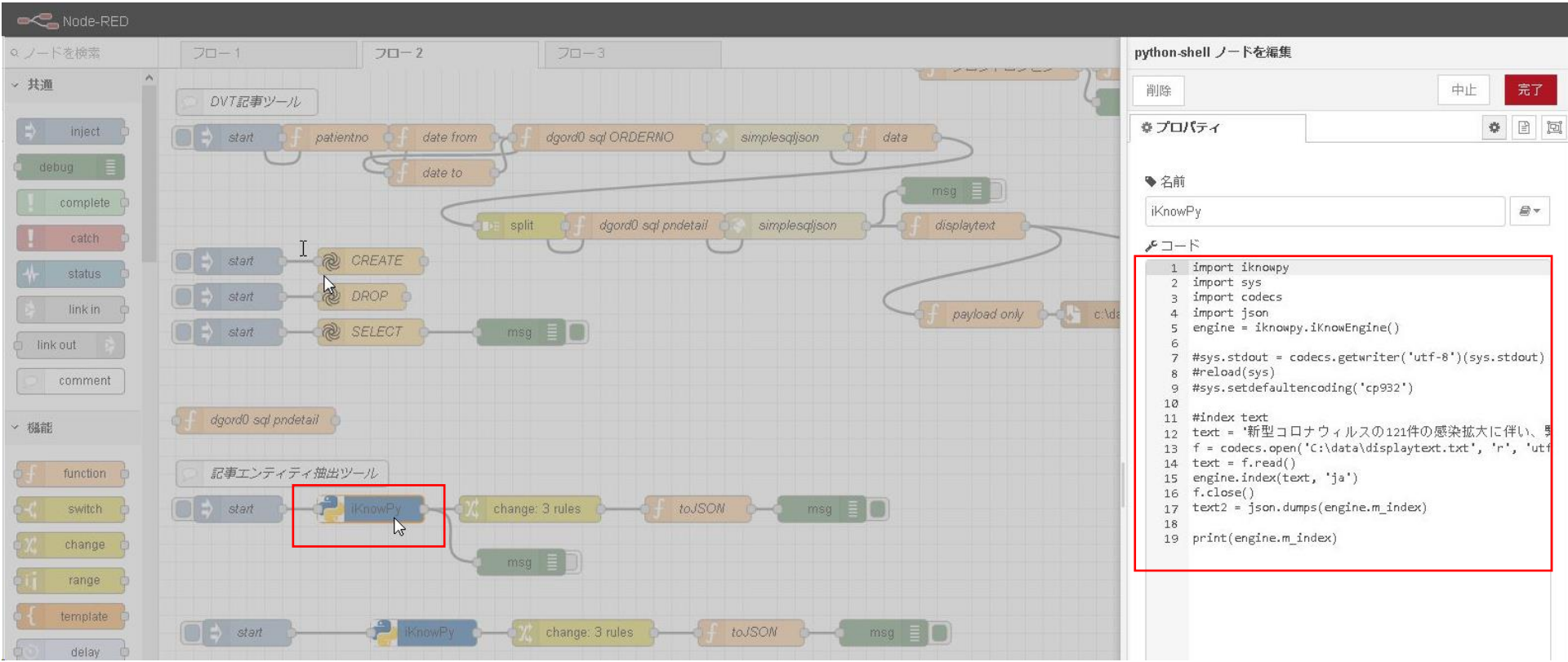
記事中のキーワードを用いたスイッチアクション

患者ID、期間をもとに診察記事を抽出し、含まれるキーワードによってフラグを複数生成するデモ
このようなコード体系では、運用後のロジック追加の際に機能全体の稼働保証を憂慮せずに済むメリットが大きい



プロトタイプオブジェクト指向の特長

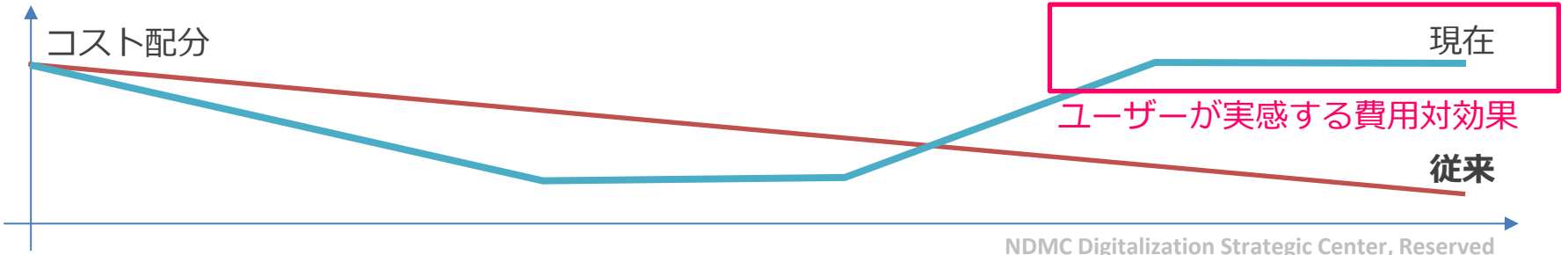
InterSystemsがGitで公開しているiKnowPyを呼び出し（ファイルで渡し、結果はmsg.payloadで戻せる）
将来的な機械学習機能の実装にも対応できる



IPCIコンセプトによる医療情報ビジネスへのインパクト

“Right Business” = “人間が時間・技能をかけて労働する内容に合わせた費用拠出をする”方針の推進

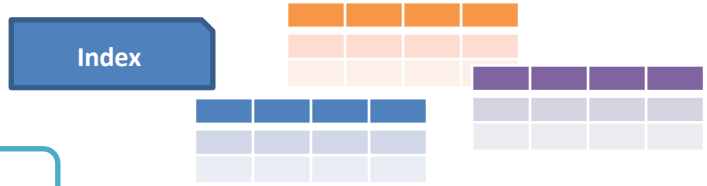
①	②	③	④	⑤
ハードウェア購入費	ソフトウェア購入費	保守費	実装・接続・設定費	改造費
囲い込み	不労利幅	脆弱設備	下請け化	内部工数
従来 独自規格 個別性	ライセンス負担 バージョン変化	多大な 人力負担	リスク計算が不十分で 高額化する一般管理費	継承問題 個別性
現在 共通化 仮想化	信頼性の高い オープンソース	冗長化 可視化	リファクタコスト低減 アーキテクチャ	スクリプト化 標準化

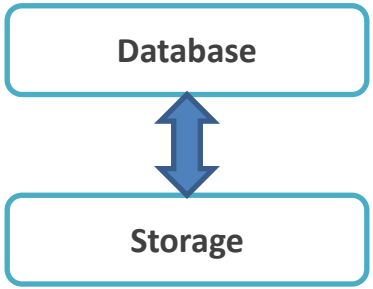


DBのindex構築の属人性、医療データベースが抱える問題、根本改善につい

医療に限らないデータベース設計の価値
・高レスポンス→DWH問題への解
・SQL構文のシンプルさ→属人性からの離脱
・保守・バージョンアップ省力化

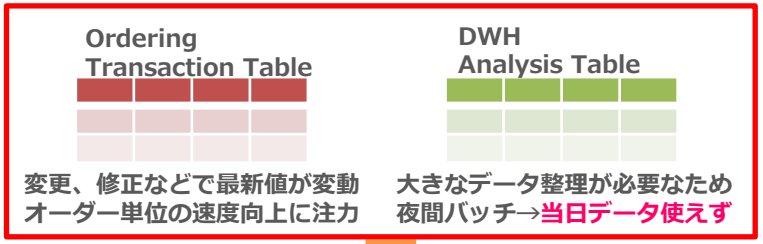
問題点
Indexの最適化問題
技術者の減少
DB数が多大
Oracle独自API
改造工数の肥大化



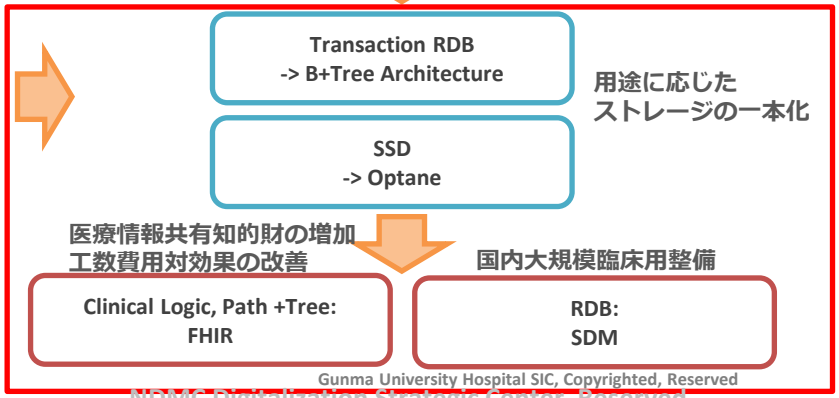


問題点
HDD RAID6の遅さ：RAC不安定性の原因
SSDでも処理速度が不十分
ストライピングでの書き込み速度低下

かつての取り組み：
別途DWHを持つ方法：DB容量の2重投資、SQLテーブルの複雑さが障壁
→集計したいデータ項目が分かりにくく、分かったとしても実用的な時間で抽出できない



IPCIコンセプト提案



大幅な改造の停止
改造工数抑制

群大病院としての
中期的データ保持形態案

GUI/Transaction
Vender-Made DB

The Tutorial is Completed
お疲れ様でした