



# 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



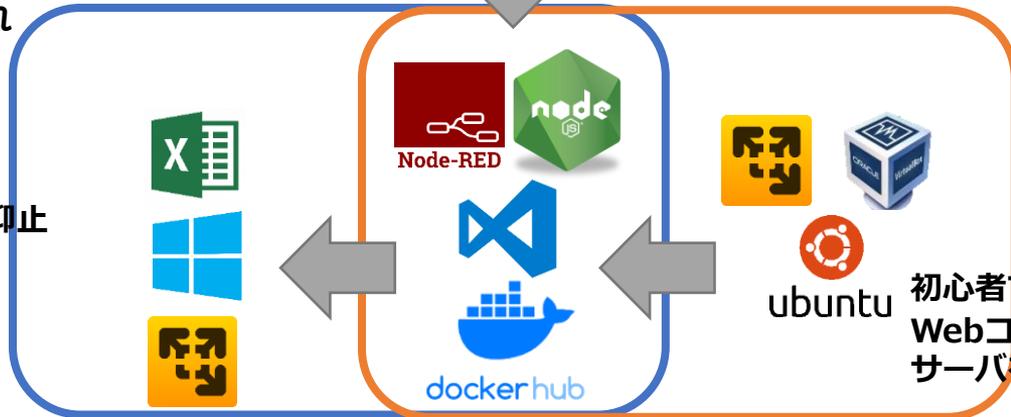
Androidは現時点では必ずしも成功した  
開発プラットフォームとは言えない



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



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

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

pip/npmなどのコマンドインストーラが  
再び浸透してきた



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



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」



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

dockerhub

として、

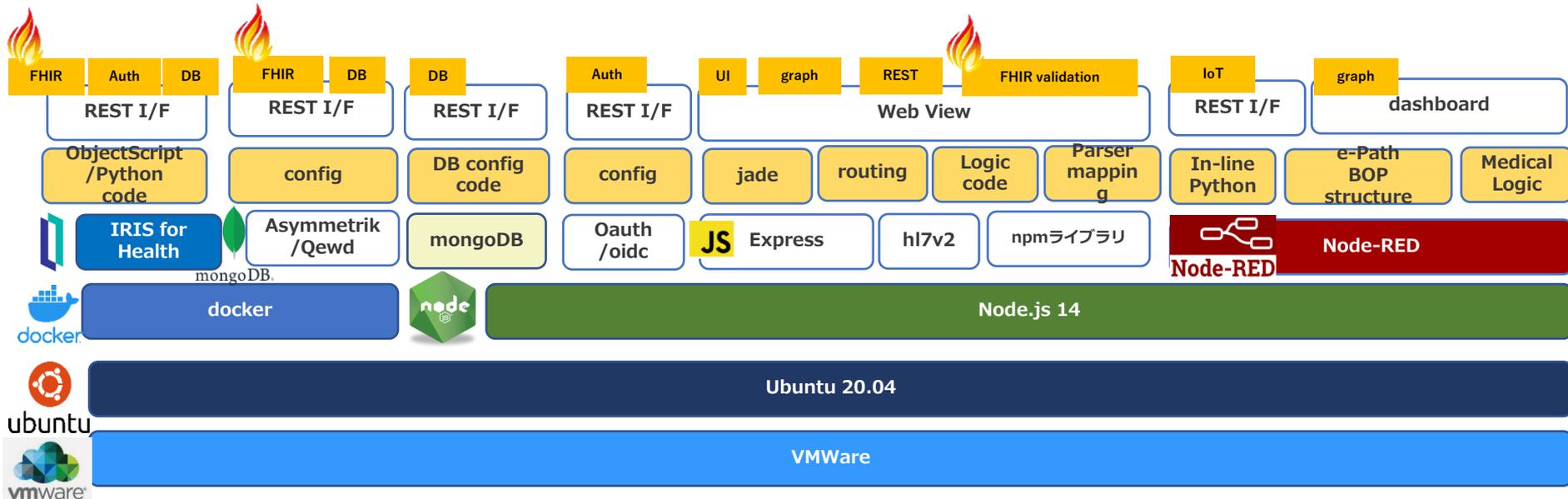


Node-RED : 医療ロジック・Webインターフェース  
上に医療アルゴリズムを永続的に蓄積・利用するサーバ

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

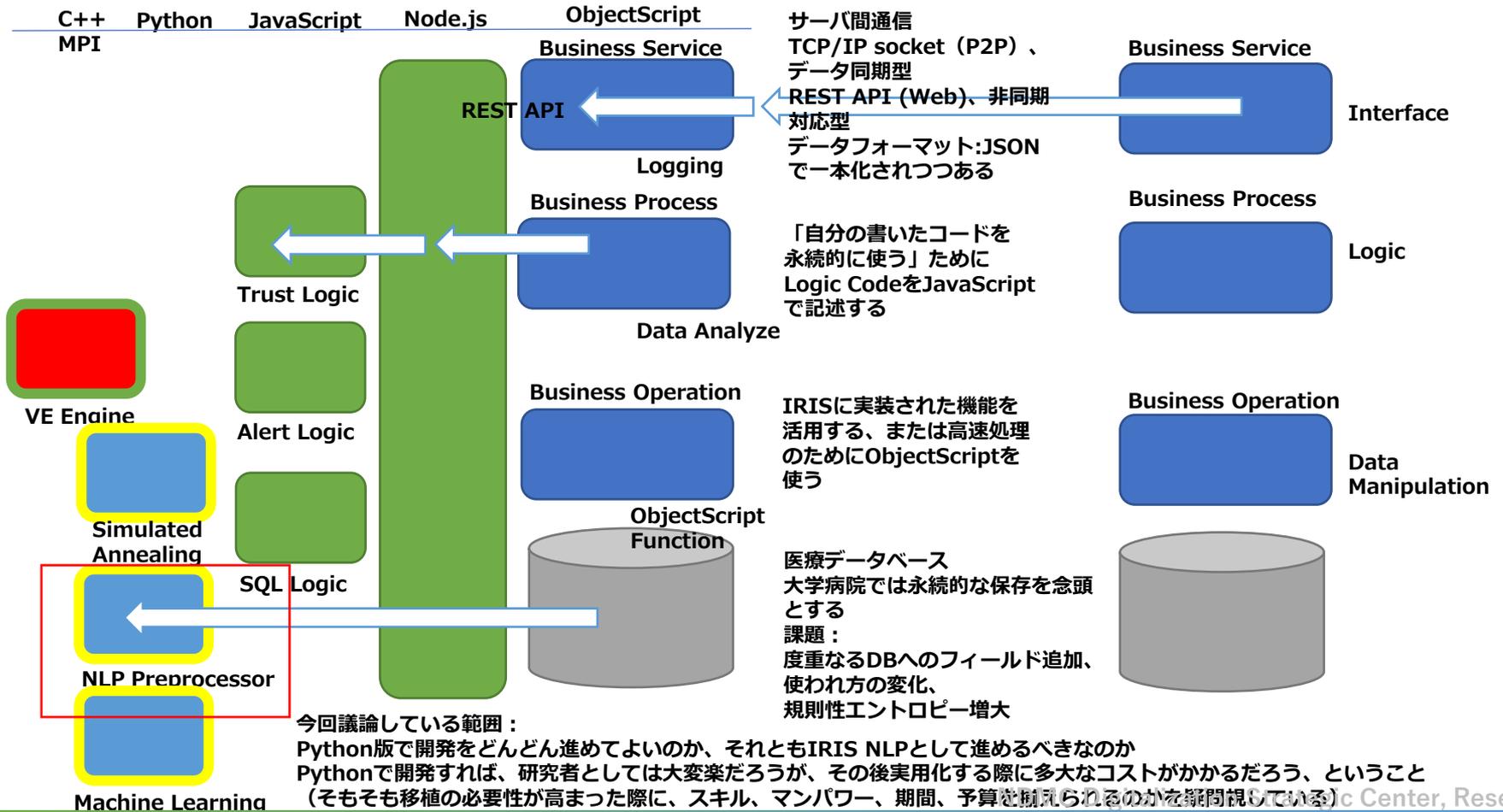


# 「継続的に開発できる」病院情報システムとは何か



使用言語

C++ Python JavaScript Node.js ObjectScript  
MPI



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、その後実用化の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、メンバー、期間、予算が制られるのかを疑問視している)

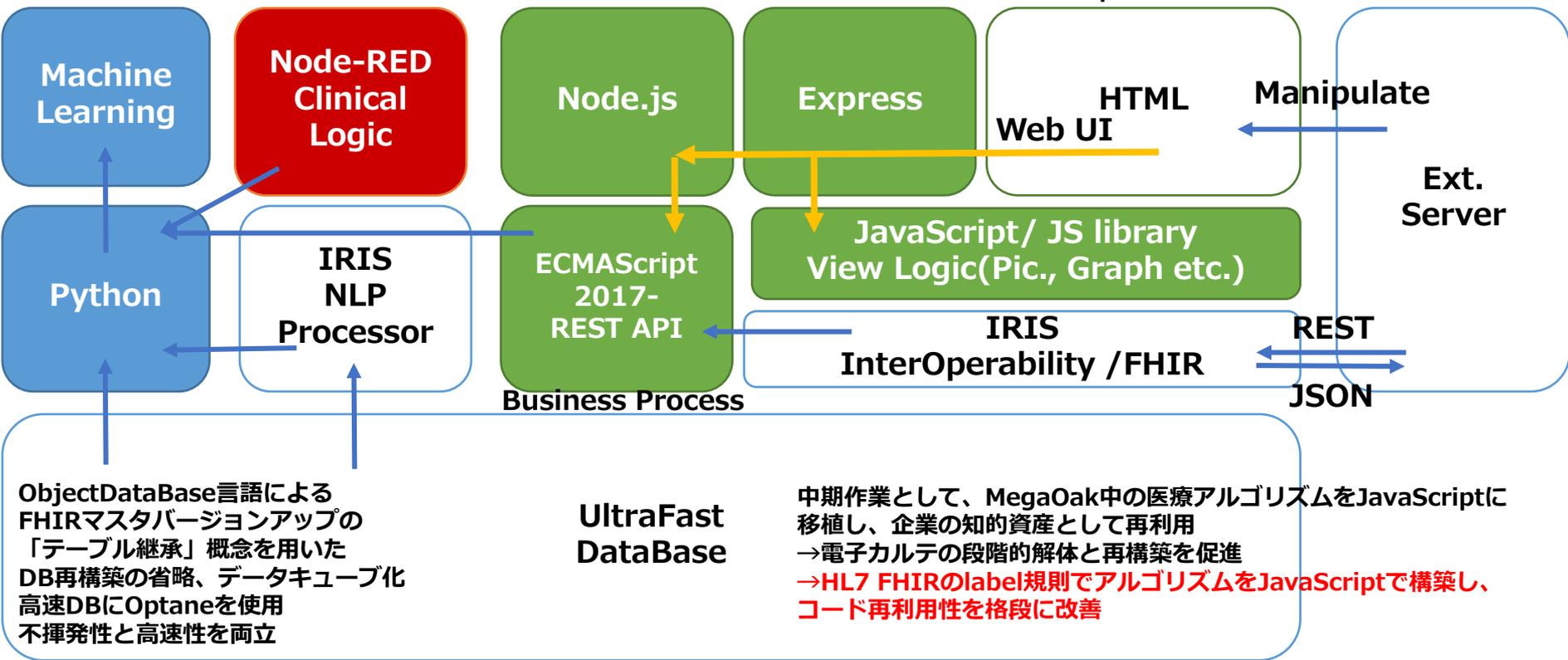
# Medical Informatics/ DX System Framework

## 病院システムにおける In-Process Clinical Intelligence (IPCI)



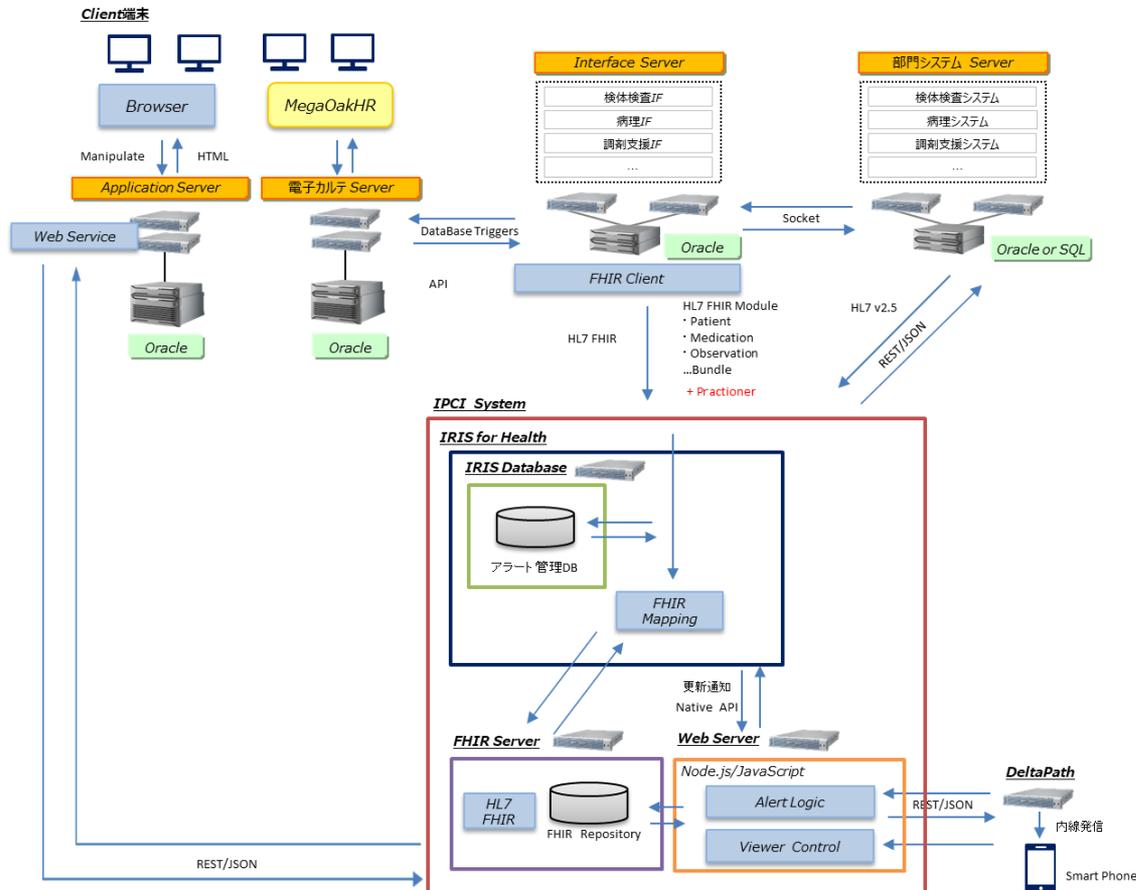
ベクトルプロセッサ等接続 **クリニカルパスの記述箇所**

開発全体のグルー言語をJavaScriptでエコシステム構築



# 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



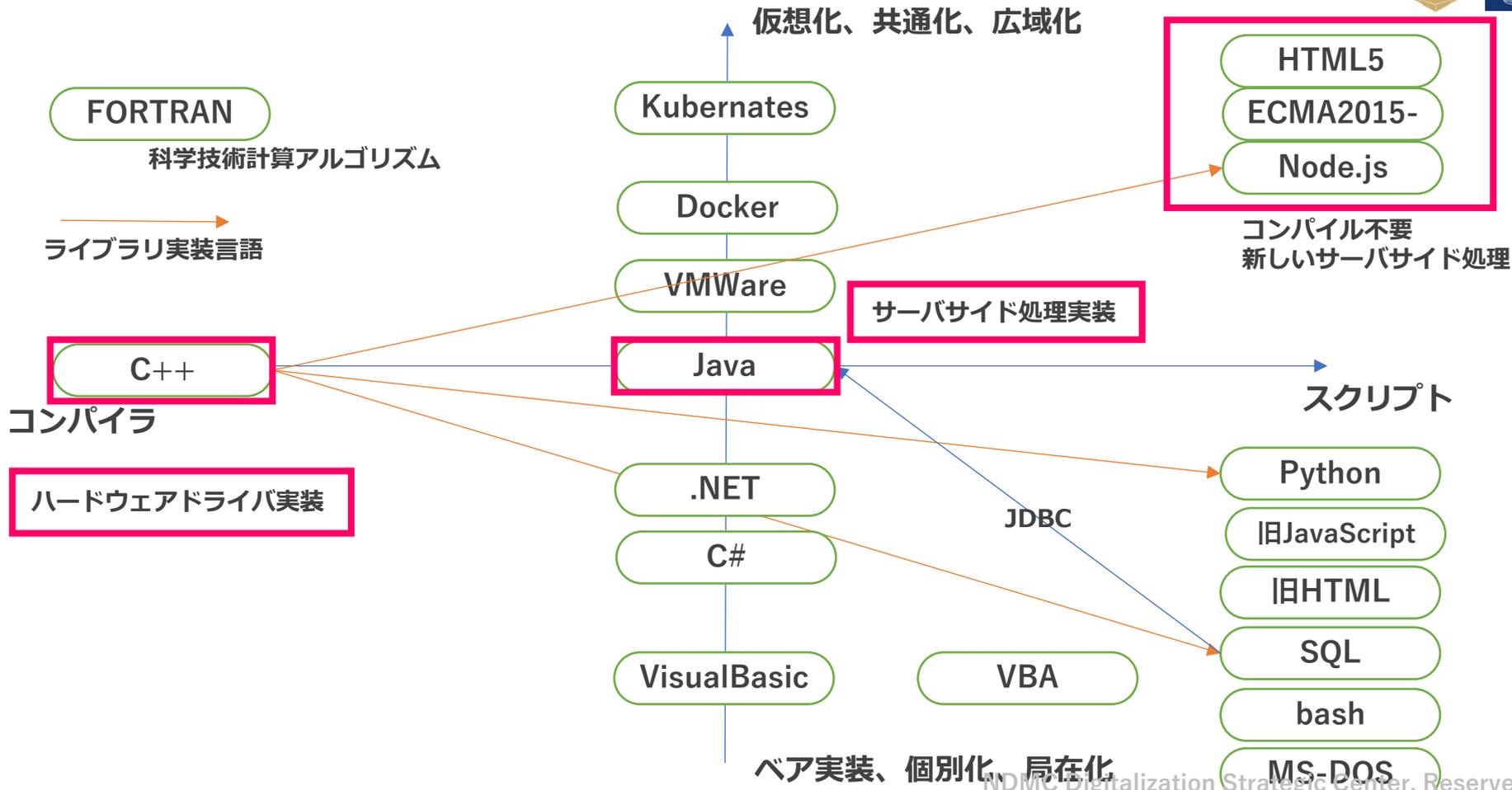
- 「環境構築」自体をオートメーションする→スキルの高いエンジニアの作業結果を複製して恩恵を受ける
- (Linuxのルート環境レベルで独立した)構造として独立することで、機能実装する際の相互関連性を減らす
- OS依存のプログラム(=マシン語に近いコンパイラを使う)を使わないようにすること
- マイクロコード思想：単純なプログラムやサービスは、再利用性が高くできる可能性がある
- 「コード作成者自身でないスタッフが、コードを容易に理解でき、迅速に活用できること」
- 文法ができるだけ簡素であること
- ライブラリが使いやすく、利用すると便利になる→エコシステムが成立している



express



# 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



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



Node-RED Con 2021

HOME ABOUT —SPEAKERS SESSIONS GALLERY SUPPORT ORGANIZER CONTACT US

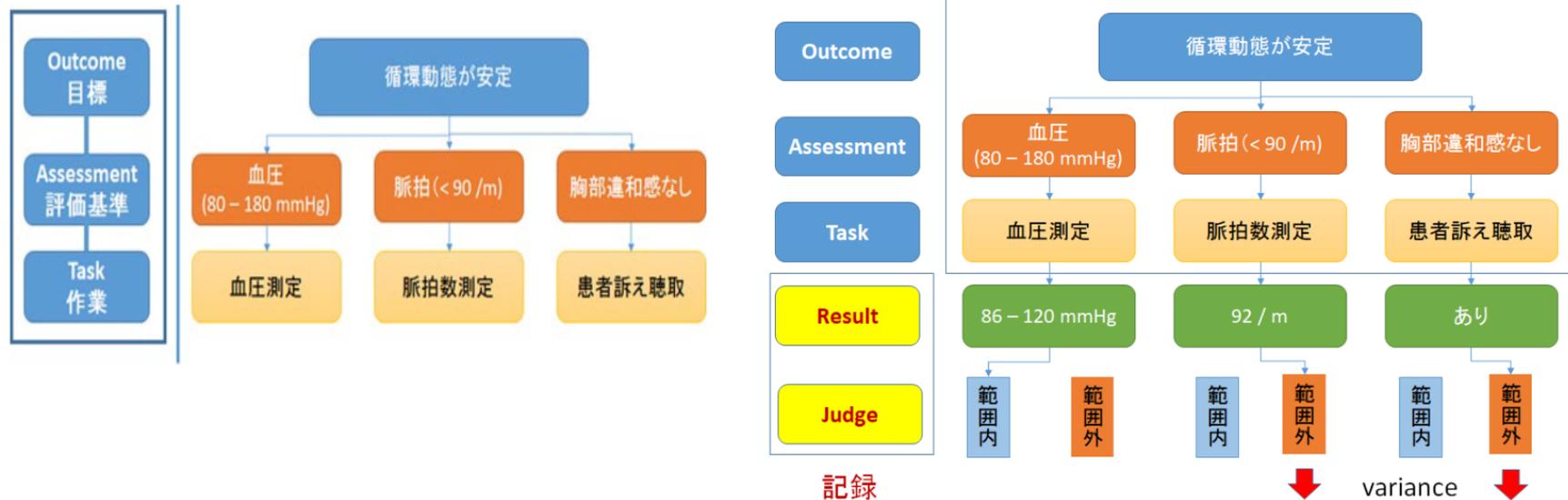
|  |   |   |  |
|--|---|---|--|
|  <p>山崎 亘<br/>株式会社ウフル</p>  |  <p>TODD SHARP<br/>Oracle</p>                |  <p>伊藤 大輔<br/>株式会社 日立製作所</p>                                 |  <p>MARC POUS<br/>balena.io</p> |
|  <p>青木 隆雄<br/>株式会社ウフル</p> |  <p>高野 幸太<br/>群馬大学医学部附属病院シ<br/>ステム統合センター</p> |  <p>JONATHAN "PEL" DE<br/>HALLEUX<br/>Microsoft Research</p> |  <p>小路 慎浩<br/>株式会社ウフル</p>       |

# 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

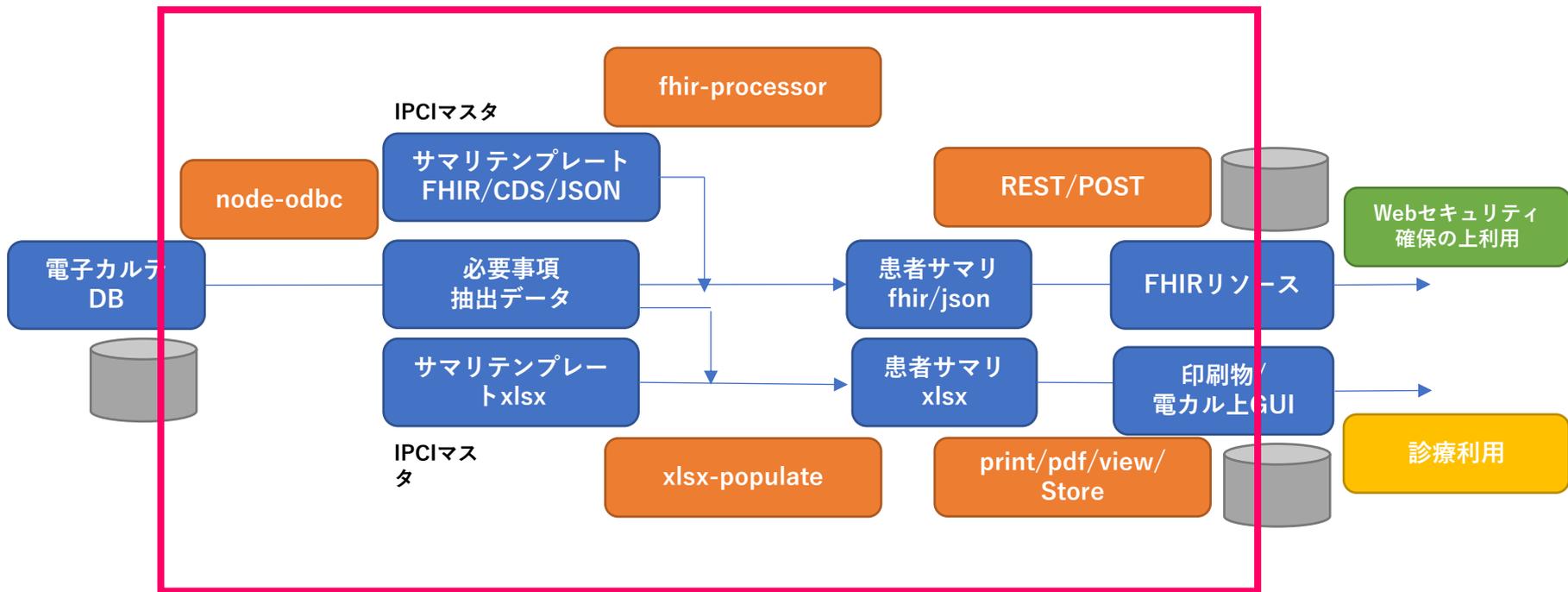


[https://e-path.jp/img/prjt/outcum\\_unit.pdf](https://e-path.jp/img/prjt/outcum_unit.pdf)

# 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

msgオブジェクトの発生



function1

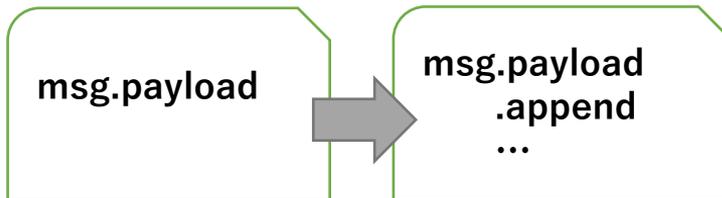
各ノード自身で独立して動かせる  
切り離れたまま置いていてもエラーにならない  
内部的に固有のIDを持つ  
ノード自身は内部的にNode.jsのJSONデータとなっている (非常に重要)  
→テキストデータによってノードの移植が可能

function1

JSの機能でプロパティを動的に付与

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

msgオブジェクト全体



function2

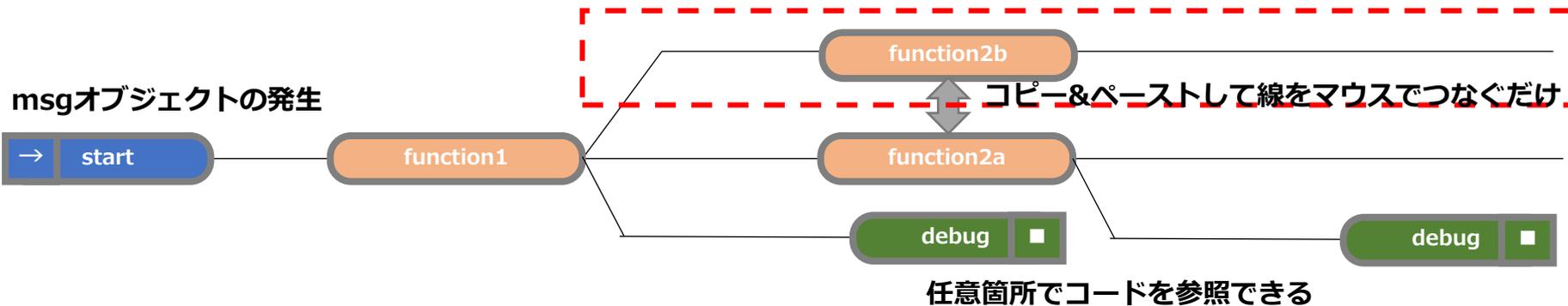
チームコーディング上の課題：  
functionの記述によってはmsgオブジェクトの内容をクリアできるので、msgのオブジェクト名と保持/削除ルールをわかりやすくする必要あり

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い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



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)



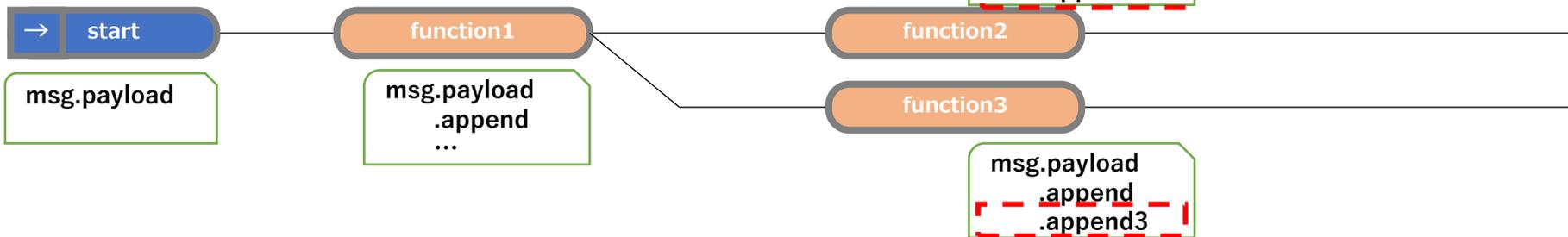
# msgオブジェクトを通じたモジュール化が容易



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くと、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、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



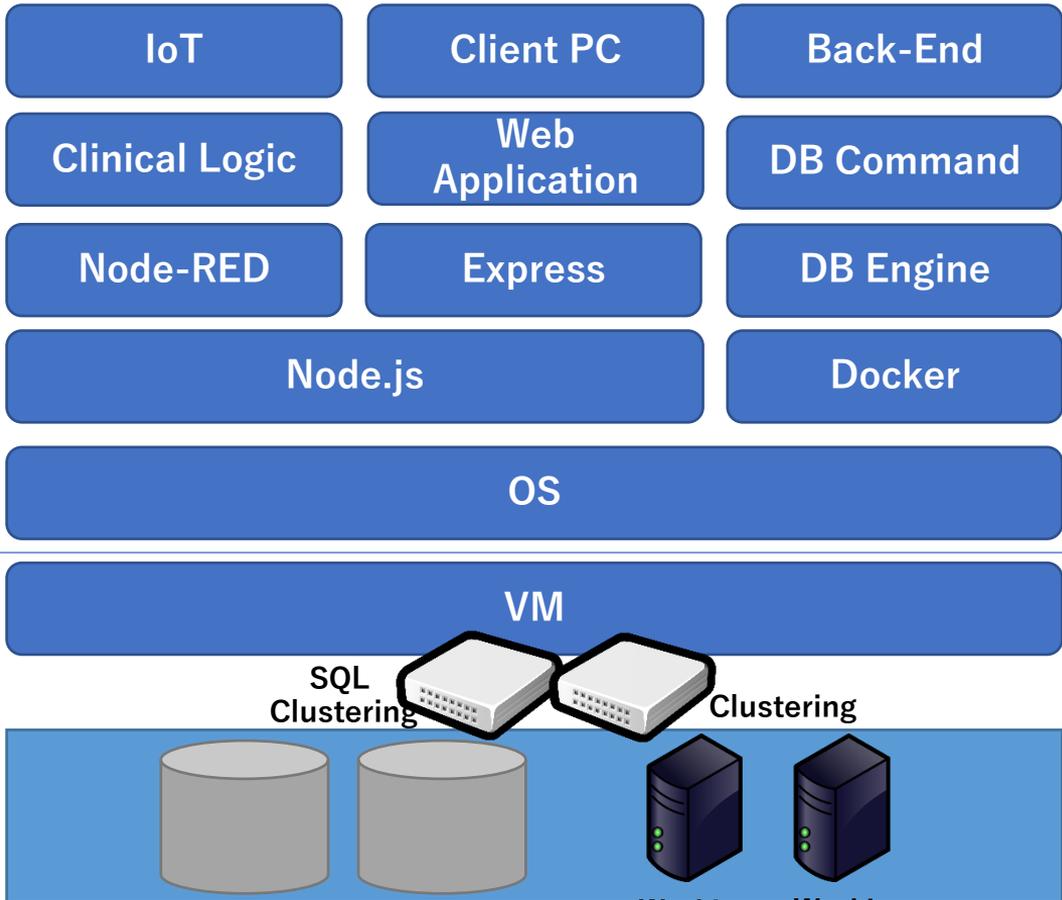
# IPCIをコアとしたスケーリング構成（大規模サーバへの応用）



In-Process Clinical Intelligence (IPCI)

ECMAScript

Virtual Fault Tolerance (VFT)



DB Macro  
Python

Linux bash

VMScript



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

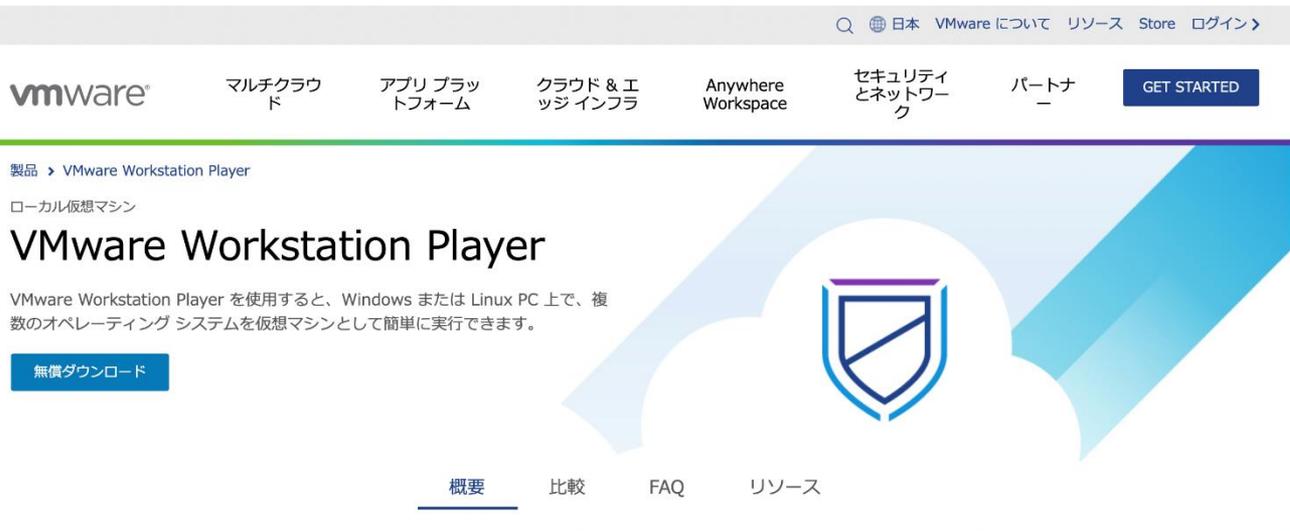
3 VM起動

4 Node-RED起動

5 Node-REDフロー作成

6 FHIRサーバ利用

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



The screenshot shows the VMware Workstation Player product page. At the top, there is a navigation bar with the VMware logo and several menu items: マルチクラウド (Multi-Cloud), アプリプラットフォーム (Application Platform), クラウド & エッジインフラ (Cloud & Edge Infrastructure), Anywhere Workspace, セキュリティとネットワーク (Security and Network), and パートナー (Partners). A 'GET STARTED' button is also present. Below the navigation bar, the page title is 'VMware Workstation Player' and it is categorized as 'ローカル仮想マシン' (Local Virtual Machine). The main content area features a large blue shield icon with a diagonal line. Below the icon, there are tabs for '概要' (Overview), '比較' (Comparison), 'FAQ', and 'リソース' (Resources). The '概要' tab is selected. The text below the tabs reads: 'VMware Workstation Player を使用すると、Windows または Linux PC 上で、複数のオペレーティング システムを仮想マシンとして簡単に実行できます。' (Using VMware Workstation Player, you can easily run multiple operating systems as virtual machines on Windows or Linux PC.) A '無償ダウンロード' (Free Download) button is located on the left side of the main content area.

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

## 1 台の PC 上で複数の仮想オペレーティング システムを実行

授業で効率的な仮想インターフェイスを使用したい場合でも、BYO デバイスで会社のデスクトップを保護する方法が必要な場合でも、Workstation Player は、VMware vSphere Hypervisor テクノロジーを利用したシンプルでセキュアなローカル仮想化ソリューションを提供します。



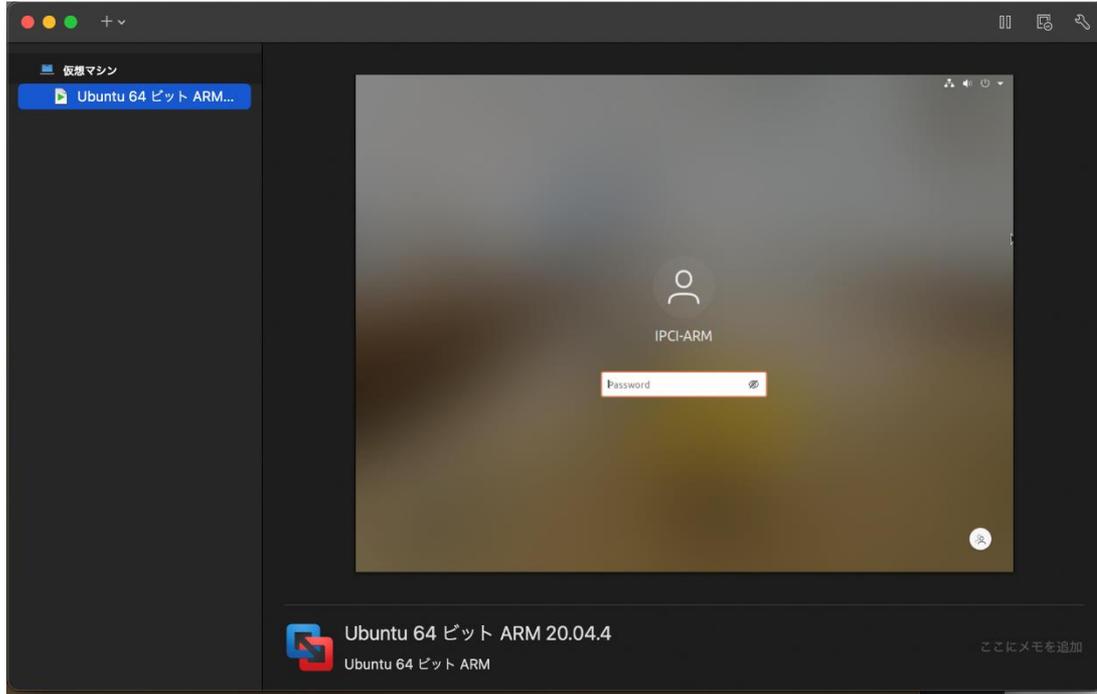
シンプルで強力なローカル環境の仮想化

20 年以上にわたって開発され、vSphere と同じハイパーバイザー



「場所を選ばない働き方」の実現

ほぼすべての Windows PC または Linux PC 上でセキュアな仮想コ



- **VMWare WorkstationPlayerで起動**

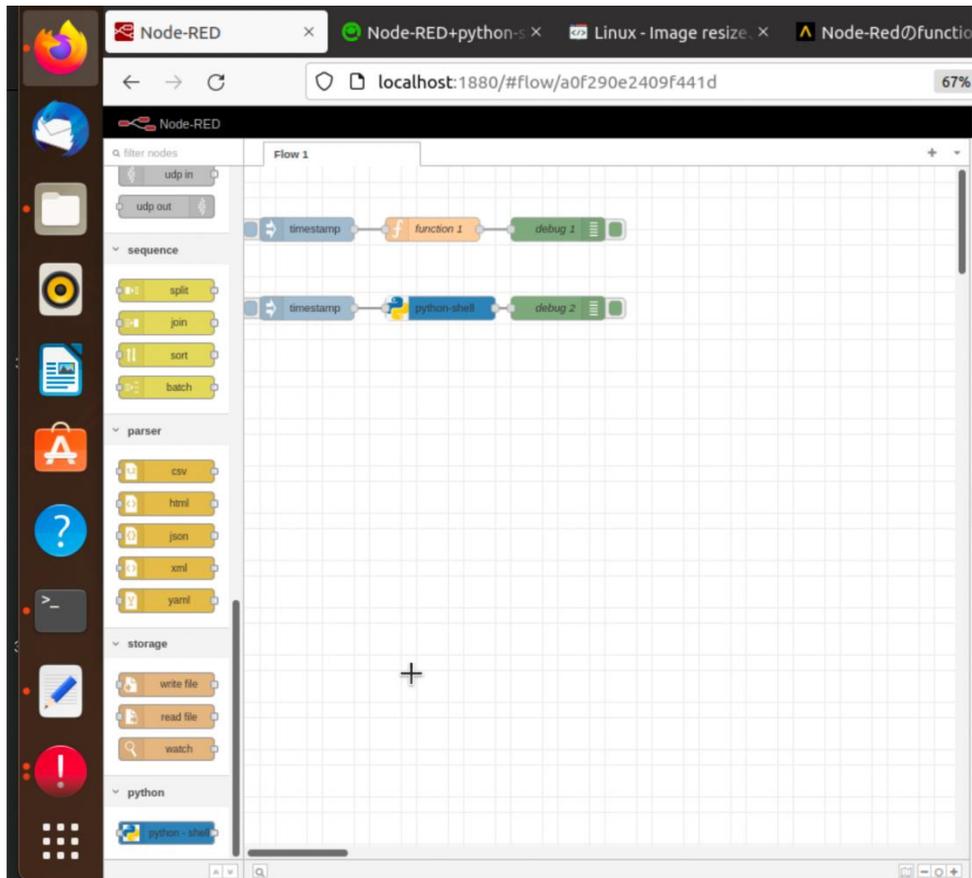
# ターミナル（コマンド入力窓） 立ち上げ



```
ipci@ipci-arm-virtual-machine: ~  
ipci@ipci-arm-virtual-machine: ~$ node-red  
Welcome to Node-RED  
=====  
18 Aug 14:53:15 - [info] Node-RED version: v3.0.2  
18 Aug 14:53:15 - [info] Node.js version: v10.19.0  
18 Aug 14:53:15 - [info] Linux 5.13.0-30-generic arm64 LE  
18 Aug 14:53:15 - [info] Loading palette nodes  
18 Aug 14:53:16 - [info] Settings file : /home/ipci/.node-red/settings.js  
18 Aug 14:53:16 - [info] Context store : 'default' [module=memory]  
18 Aug 14:53:16 - [info] User directory : /home/ipci/.node-red  
18 Aug 14:53:16 - [warn] Projects disabled : editorTheme.projects.enabled=false  
18 Aug 14:53:16 - [info] Flows file : /home/ipci/.node-red/flows.json  
18 Aug 14:53:16 - [info] Creating new flow file  
18 Aug 14:53:16 - [warn]  
  
-----  
Your flow credentials file is encrypted using a system-generated key.  
  
If the system-generated key is lost for any reason, your credentials  
file will not be recoverable, you will have to delete it and re-enter  
your credentials.
```

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

The screenshot displays the Node-RED web interface in a browser. The browser address bar shows the URL `localhost:1880/#flow/a0f290e2409f441d`. The interface includes a left sidebar with a node palette, a central workspace with a flow diagram, and a right sidebar with a debug console.

The flow diagram shows two parallel paths:

- Path 1: `timestamp` node → `function 1` node → `debug 1` node.
- Path 2: `timestamp` node → `python-shell` node → `debug 2` node.

The debug console on the right displays three error messages:

```
8/18/2022, 3:53:59 PM node: 7b45054cd4814e09
msg: string[0]
"Cannot find module '/home
/ipci/.node-red/node_modules
/nodered-contrib-mypython/python-
shell/'"

8/18/2022, 3:54:32 PM node: 7b45054cd4814e09
msg: string[0]
"Cannot find module '/home
/ipci/.node-red/node_modules
/nodered-contrib-mypython/'"

8/18/2022, 3:57:08 PM node: 7b45054cd4814e09
msg: string[45]
"Cannot read property 'runString' of
undefined"

8/18/2022, 3:57:40 PM node: 7b45054cd4814e09
msg: string[45]
"Cannot read property 'runString' of
undefined"
```

# 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

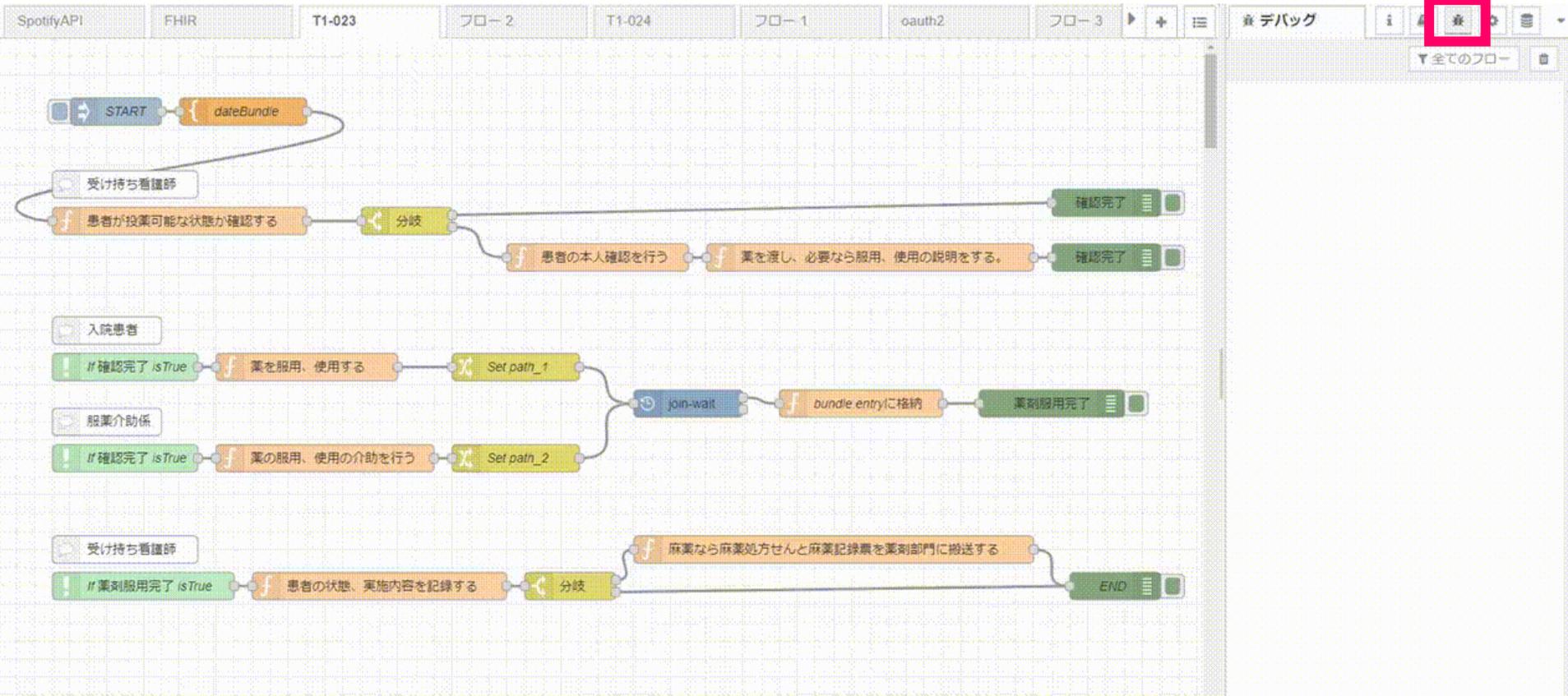
The screenshot shows the Node-RED web interface in a browser window. The address bar shows `localhost:1880/#flow/a0f290e2409f441d`. The main workspace contains a flow named "Flow 1" with two parallel paths. The top path consists of a `timestamp` node, a `function 1` node, and a `debug 1` node. The bottom path consists of a `timestamp` node, a `python-shell` node, and a `debug 2` node. A red box highlights the right-side connector of the `timestamp` node in the bottom path. The debug console on the right shows the following output:

```
8/18/2022, 3:53:59 PM node: 7b40054c4884e93  
msg : string[]  
"Cannot find module '/home  
/ipcl/.node-red/node_modules  
/nodered-contrib-mypython/python-  
shell/'"
```

# 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



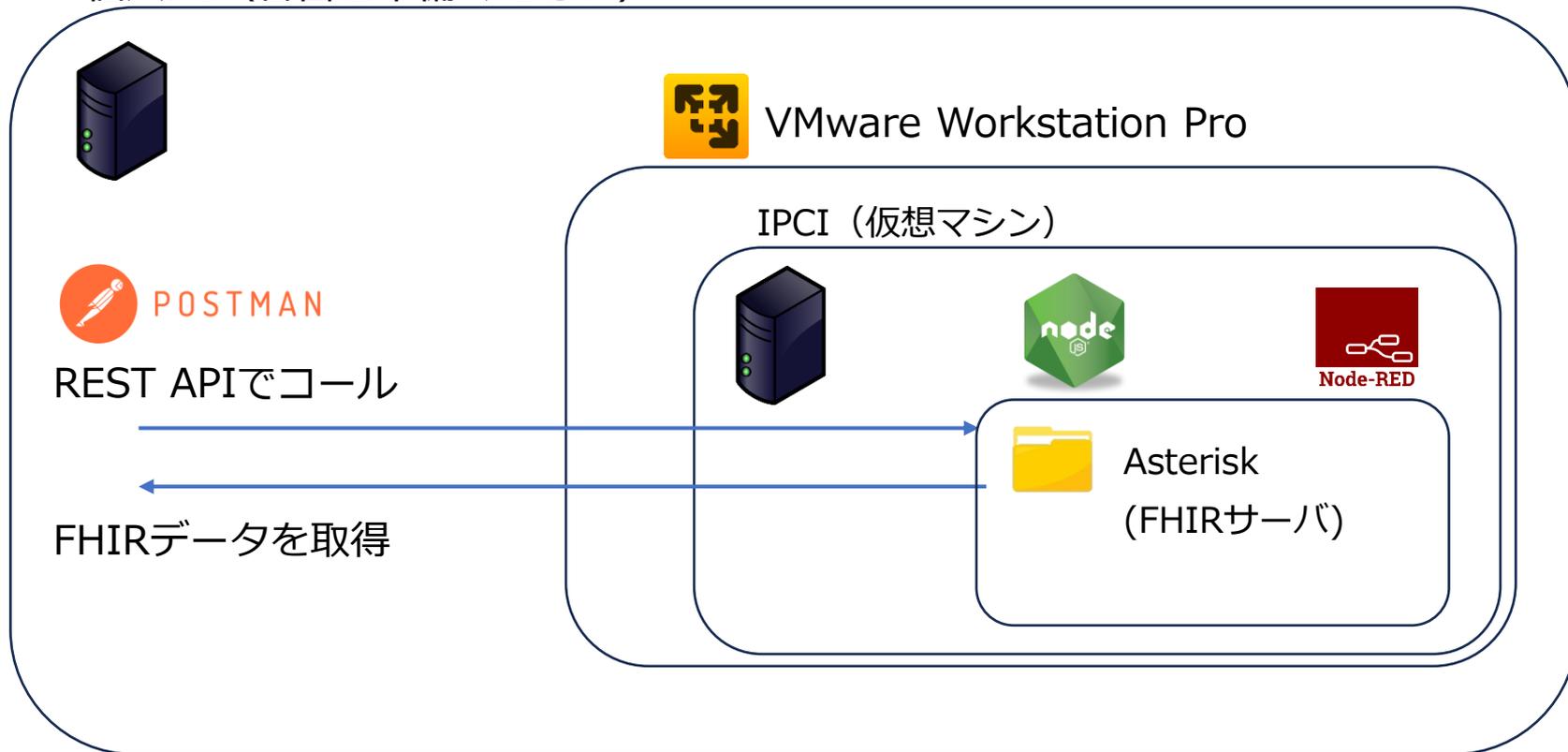


# FHIRサーバとして使用する

---

# 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

**BROADCOM** Products Solutions Support and Services Company How To Buy

## VMware Workstation Pro for Personal Use (For Windows) 17.6

Primary Downloads Open Source

Search 17.6 522390 English

I agree to [Terms and Conditions](#)  Expand All

| File Name  | Last Updated         | SHA2   | MD5                              |
|--|----------------------|--|----------------------------------|
| <b>VMware Workstation Pro for Personal Use (For Windows)</b>                     |                      |  |                                  |
| VMware-workstation-full-17.6.0-24238078.exe(447.97 MB)<br>Build Number: 24238078 | Aug 28, 2024 03.44AM | e34461ffbc38ca7baa7928f7f37575ef31129961099ea96b43a64b06462778 | ff4007cc5af53d83790fe87ed2d850f5 |

1 to 1 of 1 records

Products Solutions Support and Services Company How To Buy

Copyright © 2005-2024 Broadcom. All Rights Reserved. The term "Broadcom" refers to Broadcom Inc. and/or its subsidiaries.  
Accessibility Privacy Supplier Responsibility Terms of Use Site Map

# Postmanのダウンロード



## Postmanのwebページで各OSに合わせたソフトウェアをダウンロード

The image shows a comparison between the Postman desktop application and the web version. On the left, the desktop app page is shown with a green box highlighting the download buttons for Windows, Mac, and Linux. On the right, the web version page is shown with a screenshot of the Postman interface.

**Build APIs together**  
Over 30 million developers use Postman to build APIs. You can get started by signing up or downloading the desktop app.

Download the desktop app for

- Windows
- Mac
- Linux

**Download Postman**  
Download the app to get started using the Postman API Platform today. Or, if you prefer a browser experience, you can try the web version of Postman.

**The Postman app**  
Download the app to get started with the Postman API Platform.

[Windows 64-bit](#)

By downloading and using Postman, I agree to the [Privacy Policy](#) and [Terms](#).

[Release Notes](#) →

Not your OS? Download for Mac ([Intel Chip](#), [Apple Chip](#)) or Linux ([x64](#), [arm64](#))

**Postman on the web**

**Postman Interface Screenshot:**  
The screenshot shows the Postman web interface with a collection named 'Notion's Public Workspace'. The selected environment is 'Notion API / Databases / Retrieve a database'. The endpoint is 'GET https://api.notion.com/v1/databases/id'. The response is a table with columns 'KEY', 'VALUE', and 'DESCRIPTION'. The 'id' field has the value '{{DATABASE\_ID}}' and is marked as 'Required'.

| KEY | VALUE           | DESCRIPTION |
|-----|-----------------|-------------|
| id  | {{DATABASE_ID}} | Required.   |

## 環境準備のためのダウンロード



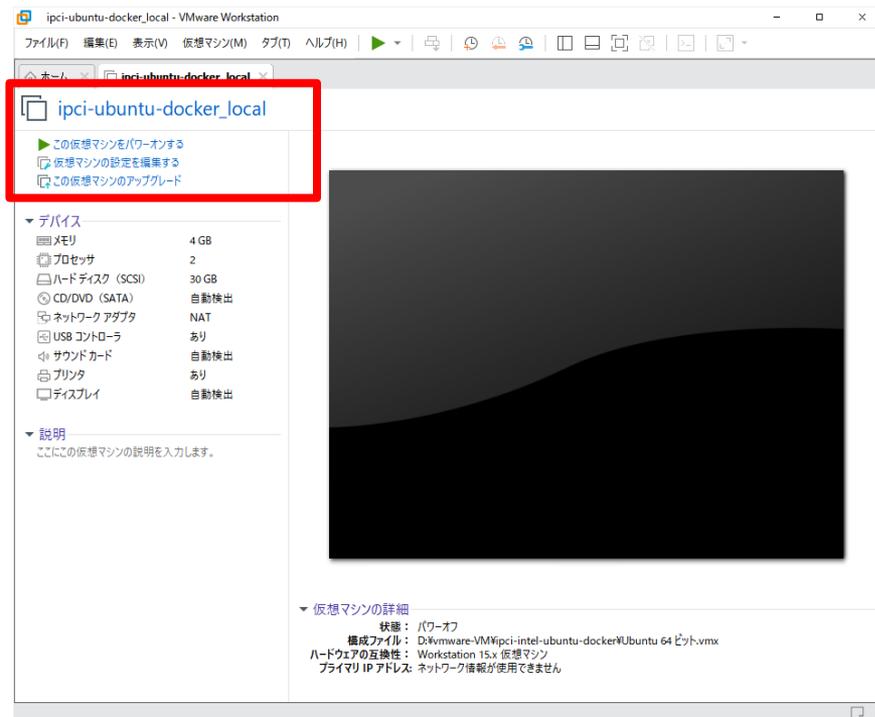
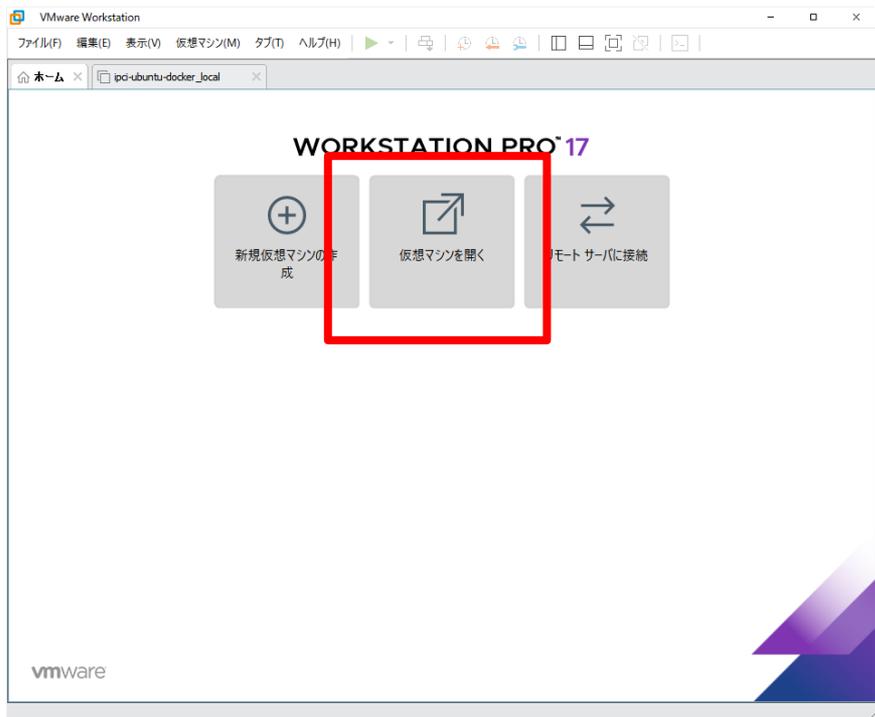
- M学会のチュートリアル専用ページからIPCIの仮想マシンイメージとハンズオンファイルをダウンロードします。

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

# IPCIの仮想マシンの起動



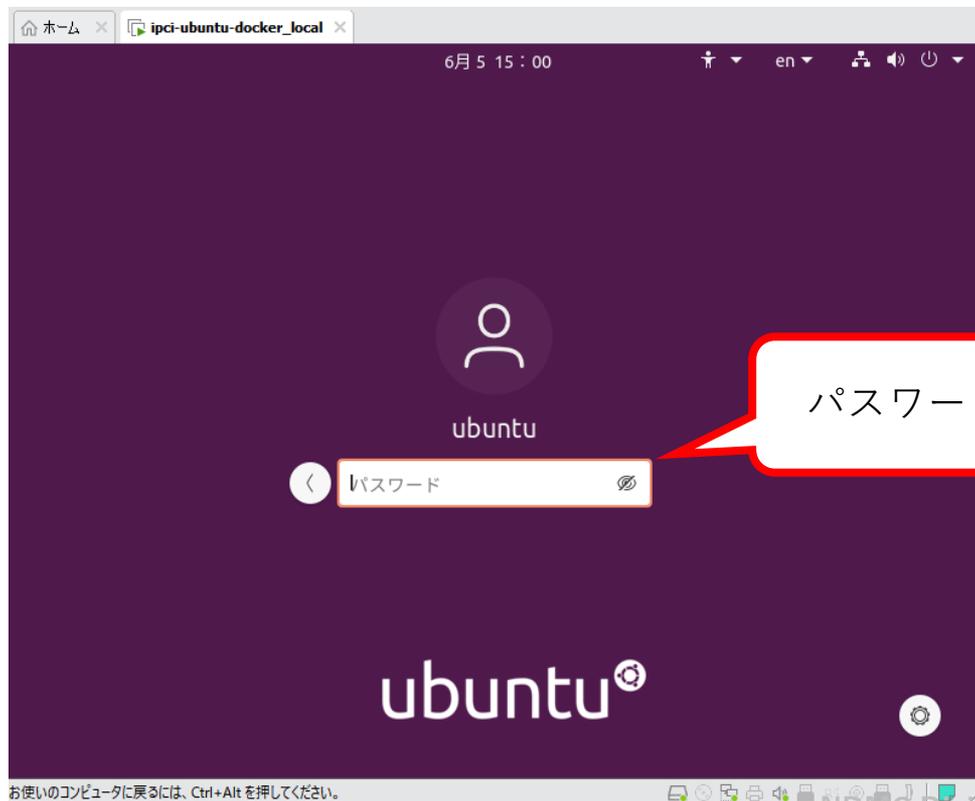
- IPCIの仮想マシンイメージをVMware Workstation Playerから開いて起動します。



# IPCIの仮想マシンのログイン



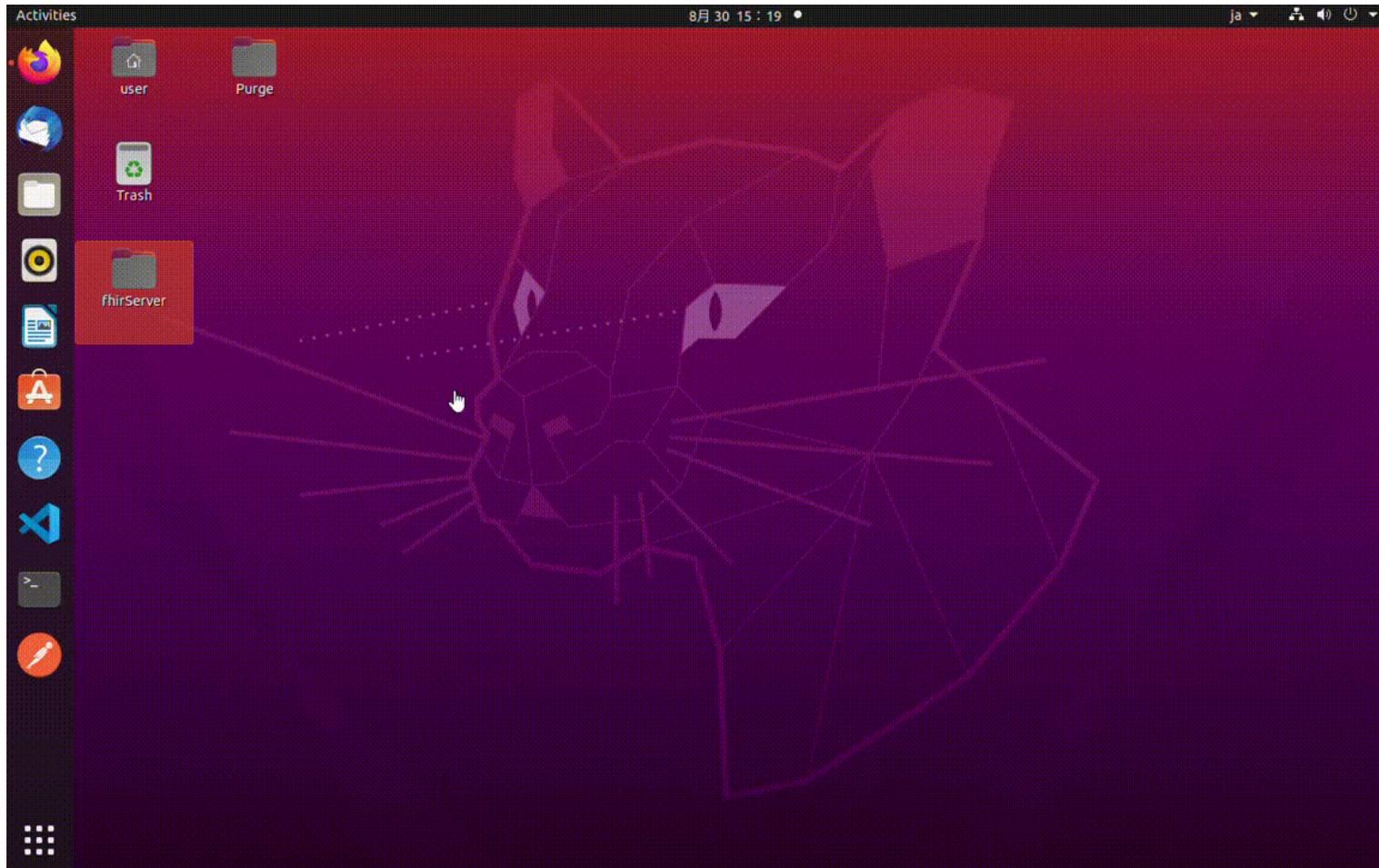
- IPCIの仮想マシンへログインします。



パスワードはadmin

お使いのコンピュータに戻るには、Ctrl+Altを押してください。

# IPCI-FHIRサーバ機能とPatientリソースの起動



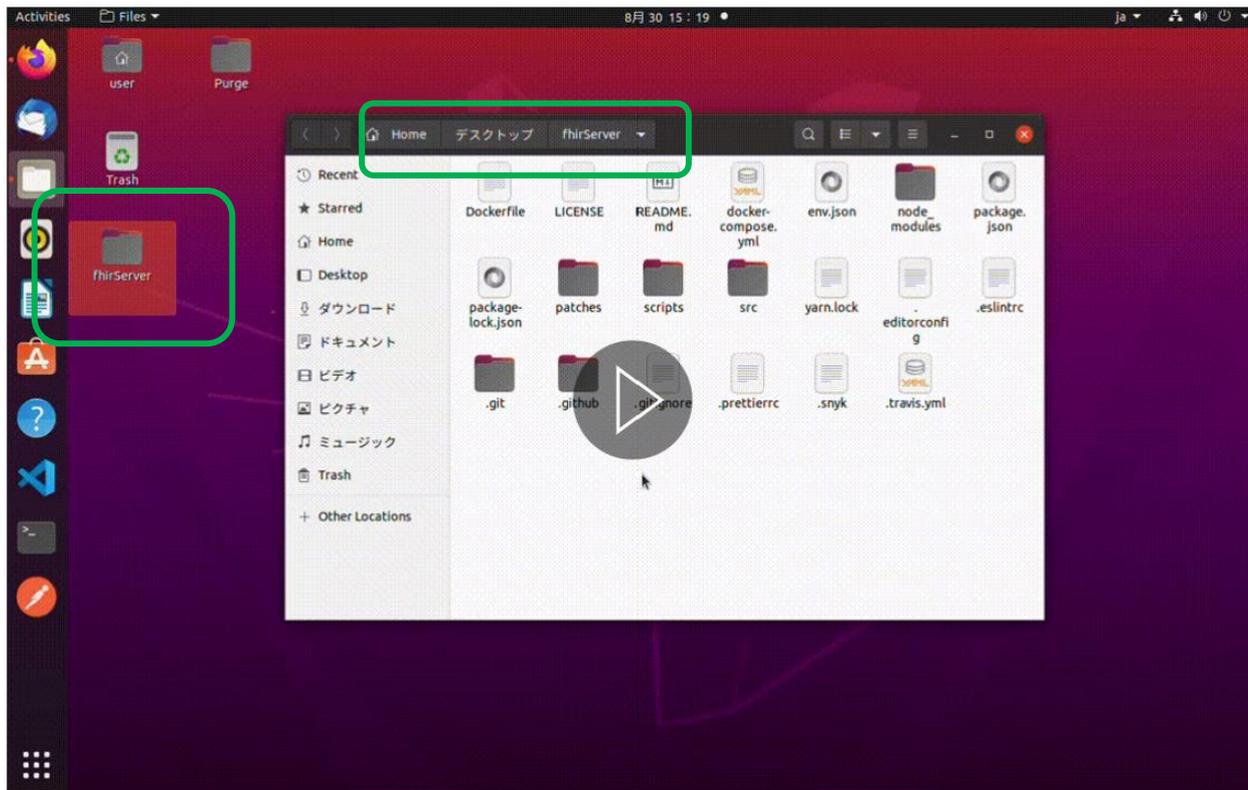
# FHIRサーバ起動(1)



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

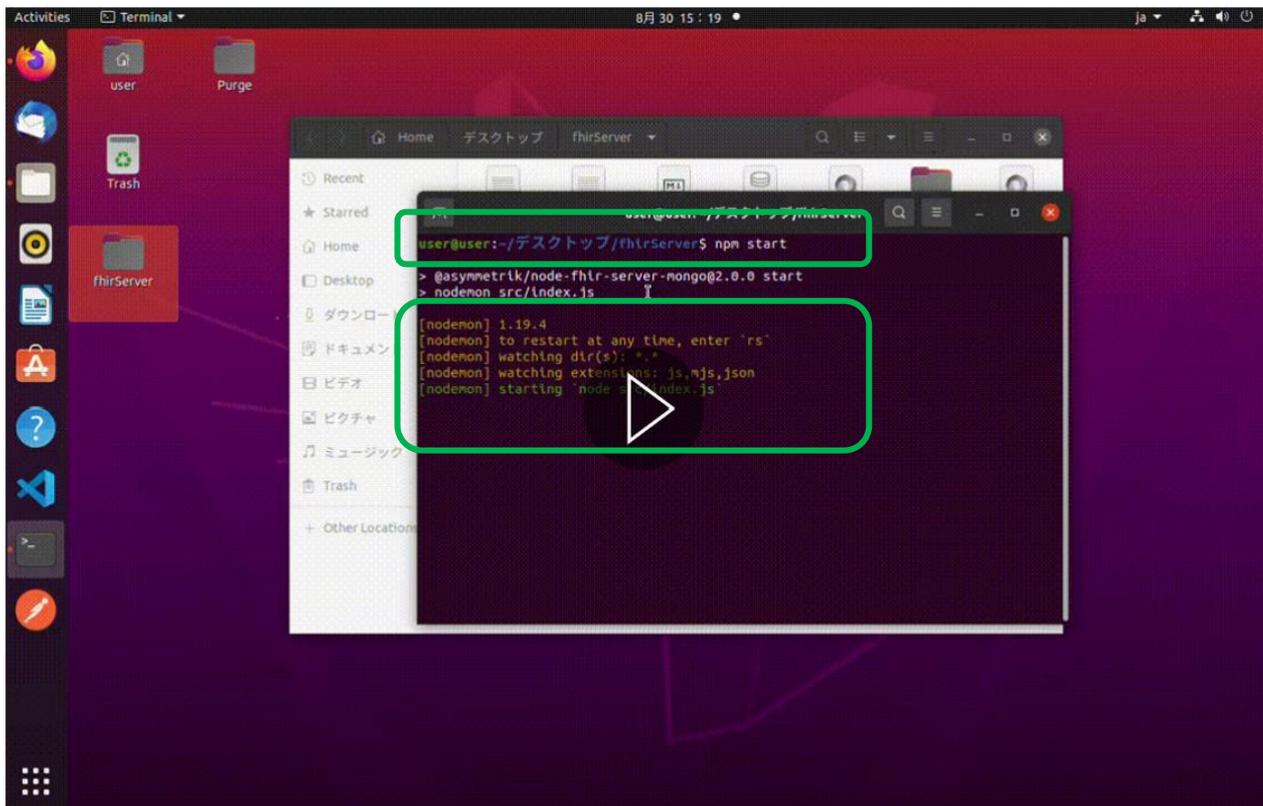


# FHIRサーバ起動(2)



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

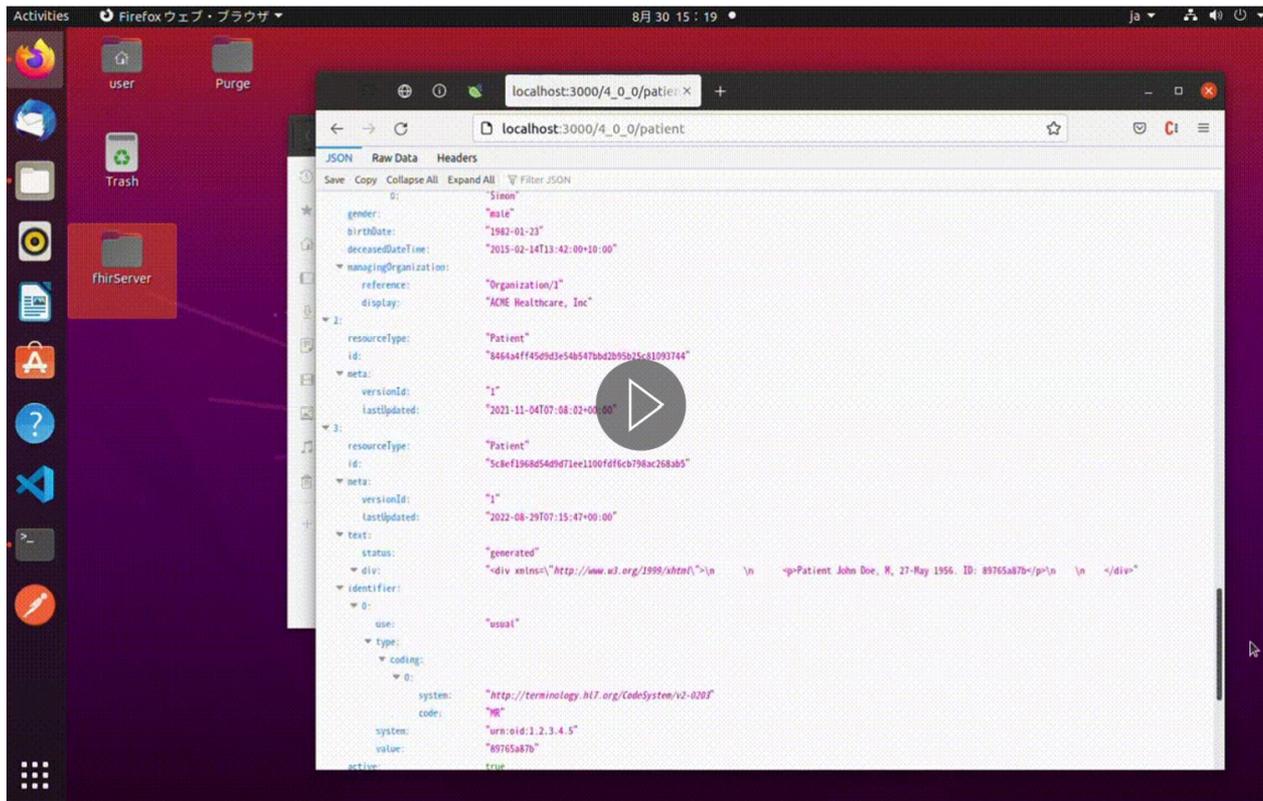
# FHIRサーバの動作確認



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



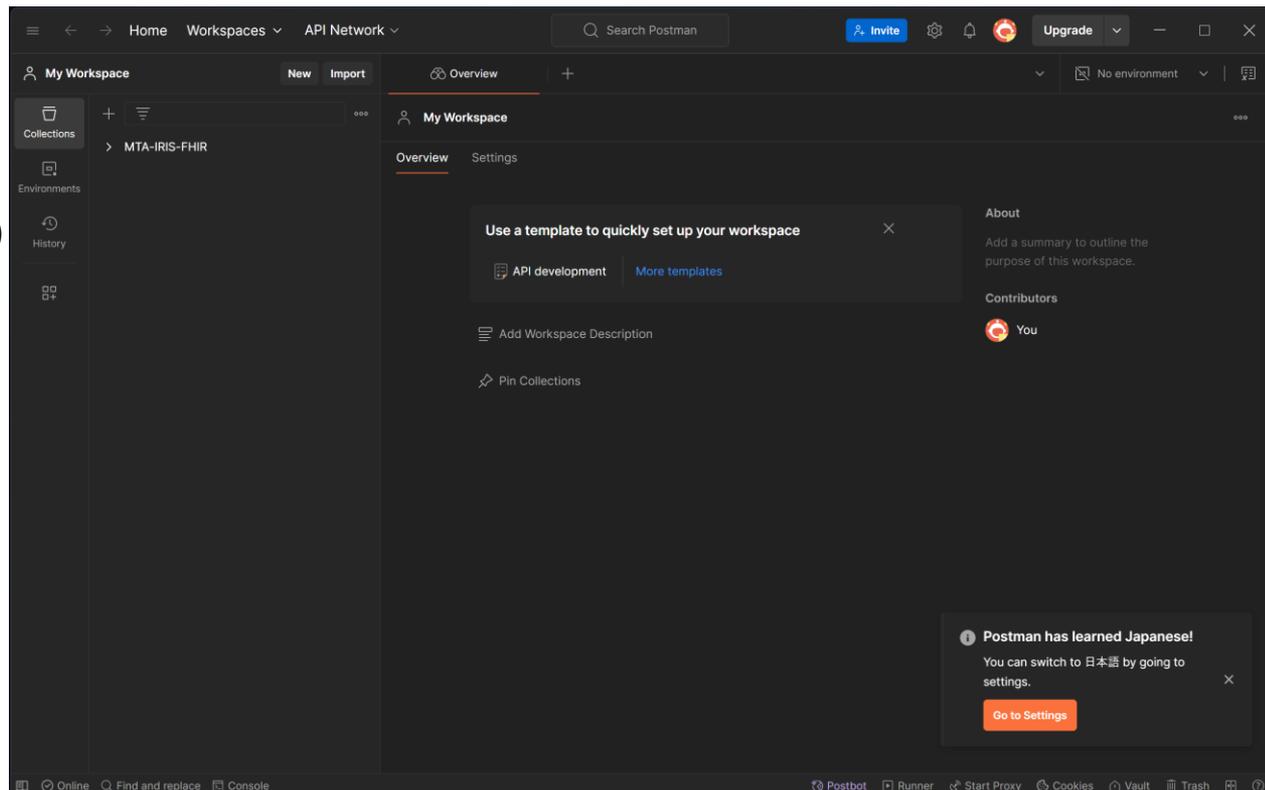
Postmanを起動

GETタブを起動



Asterisk

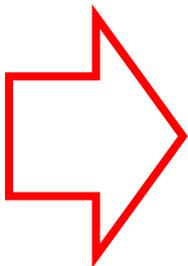
(FHIRサーバ)



# Node-Redの起動



- ターミナルを開いてNode-Redを起動します。



```
user@user: ~  
user@user:~$ node-red  
5 Jun 15:04:11 - [info] Welcome to Node-RED  
=====
```

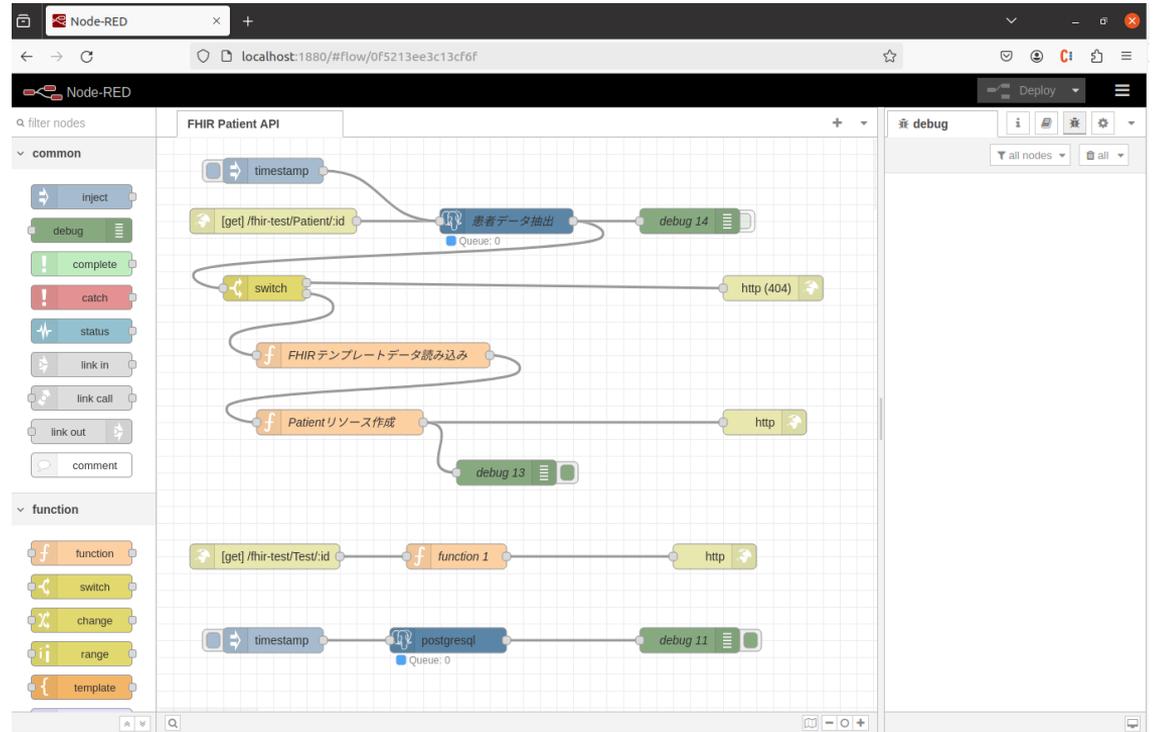
```
5 Jun 15:04:11 - [info] Node-RED バージョン: v3.0.2  
5 Jun 15:04:11 - [info] Node.js バージョン: v16.17.0  
5 Jun 15:04:11 - [info] Linux 5.15.0-107-generic x64 LE  
5 Jun 15:04:12 - [info] パレットノードのロード  
5 Jun 15:04:13 - [info] 設定ファイル: /home/user/.node-red/settings.js  
5 Jun 15:04:13 - [info] コンテキストストア : 'default' [module=memory]  
5 Jun 15:04:13 - [info] ユーザディレクトリ : /home/user/.node-red  
5 Jun 15:04:13 - [warn] プロジェクトは無効化されています : editorTheme.projects.  
enabled=false  
5 Jun 15:04:13 - [info] フローファイル : /home/user/.node-red/flows.json  
5 Jun 15:04:13 - [info] サーバは http://127.0.0.1:1880/ で実行中です  
5 Jun 15:04:13 - [warn]
```

# Node-Redの表示



- Webブラウザ開いてNode-RedのURLを開きます。

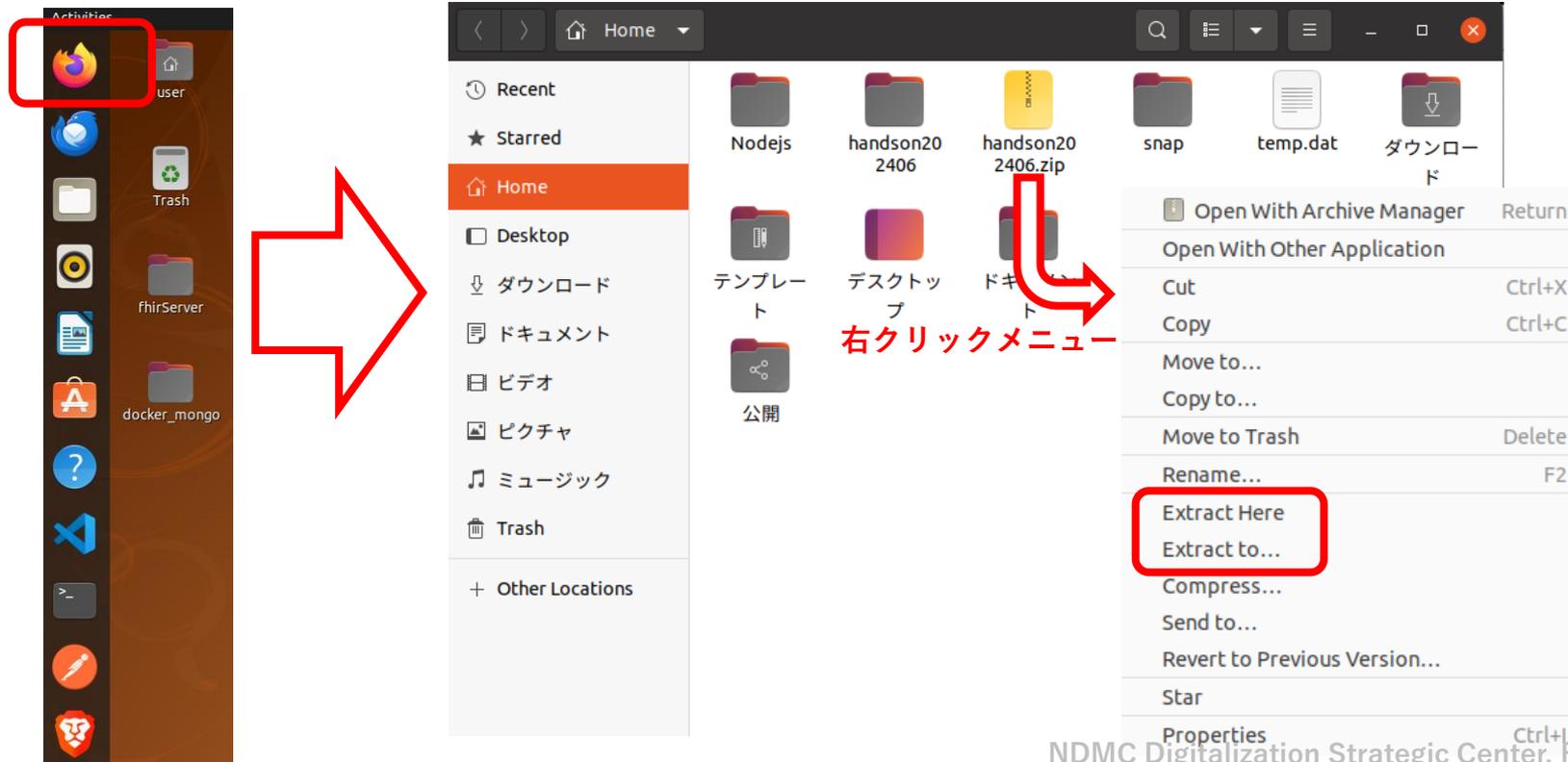
<http://localhost:1880/>



# 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
```

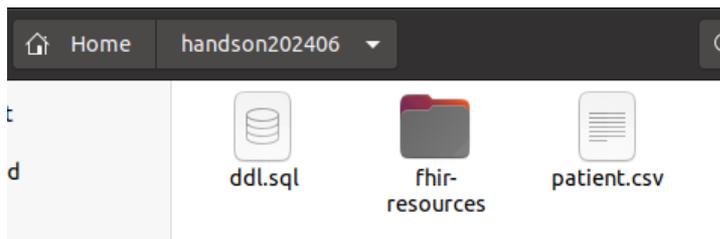
```
sudo -u postgres createuser -d -P "user"
```

```
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
```

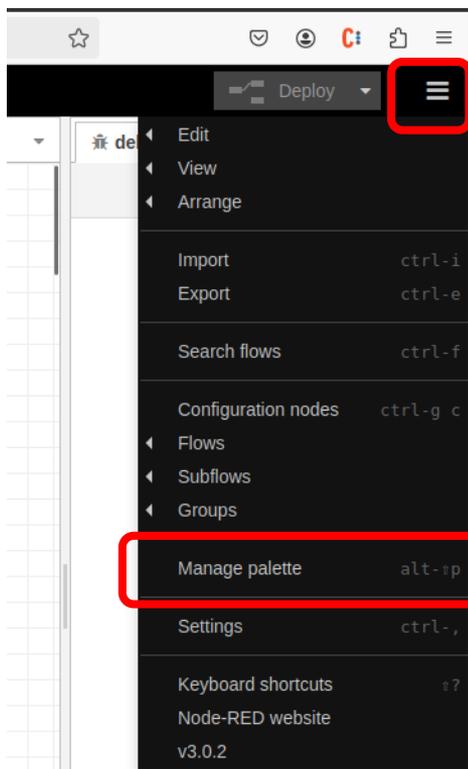
```
psql -d emrdb
```

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

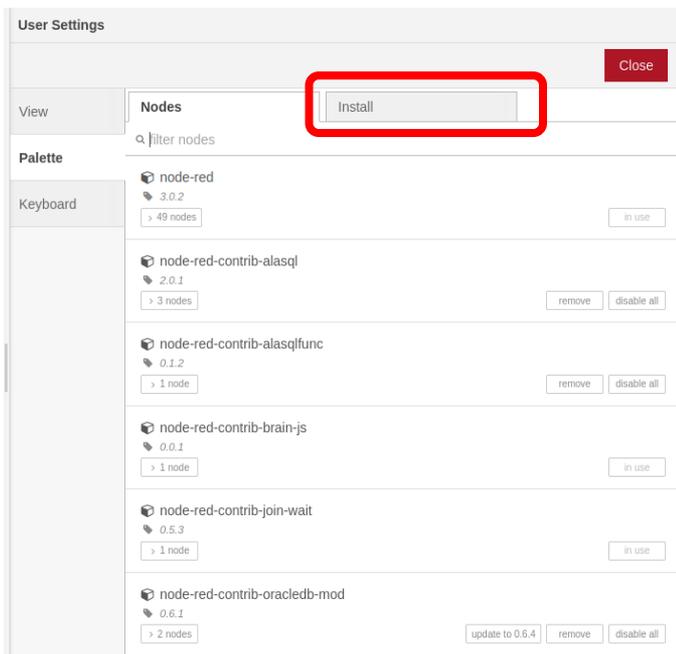


# Node-RedのDB接続プラグインインストール

- Node-Redの管理画面にあるManage Palletを開きます。

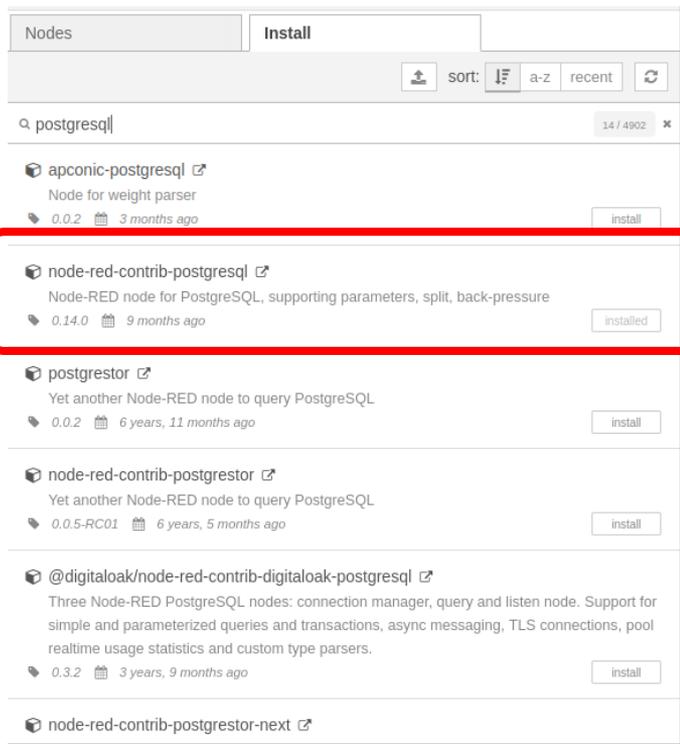


Node-Redの画面の左上にある  
三本線メニューを開く。



## 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



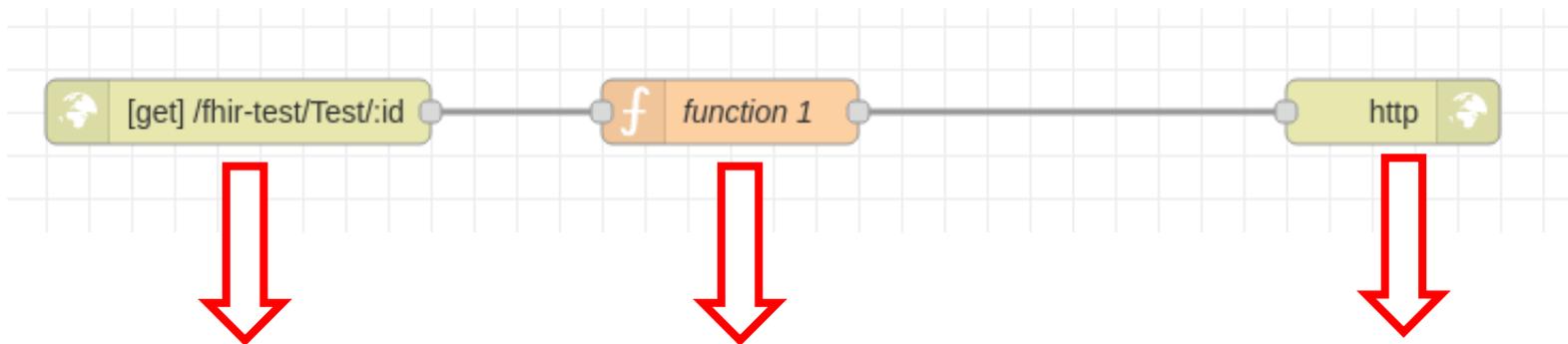
- Node-Redの画面から新規フローを作成します。



# 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。



Edit http in node

Delete Cancel Done

Properties

Method GET

URL /fhir-test/Test/:id

Name Name

Delete Cancel Done

Properties

Name function 1

Setup On Start On Message On Stop

```
1 var res=new Object();
2
3 res.PatientId = msg.req.params.id;
4 res.PatientName = "テスト 患者" + msg.req.params.id;
5
6 msg.payload=res;
7 return msg;
```

Edit http response node

Delete Cancel Done

Properties

Name Name

Status code msg.statusCode

Headers

## 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。

The screenshot shows a function editor window with the following elements:

- Buttons: Delete, Cancel, Done
- Properties section: Name: function 1
- Event tabs: Setup, On Start, **On Message**, On Stop
- Code editor content:

```
1 var res=new Object();
2
3 res.PatientId = msg.req.params.id;
4 res.PatientName = "テスト 患者" + msg.req.params.id;
5
6 msg.payload=res;
7 return msg;
```

## 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



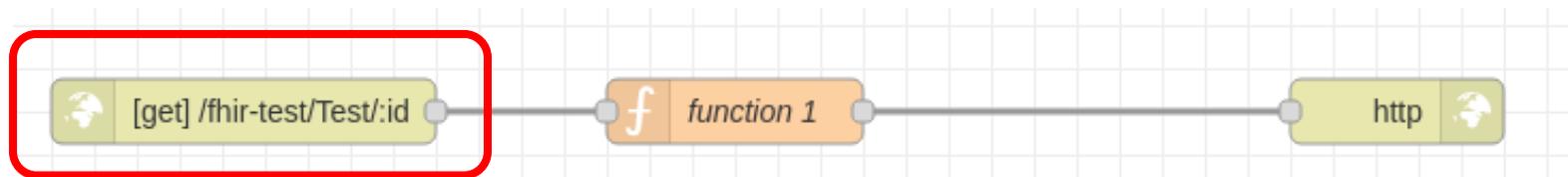
- postmanを開いて以下のURLをGETで実行するとfunction1で実装したレスポンスが返ってくることを確認します。

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

The screenshot displays the Postman application interface. The top bar shows the request method as GET and the URL as `http://localhost:1880/fhir-test/Test/1234`. The 'Send' button is highlighted with a red box. Below the URL bar, the 'Query Params' section is visible, showing a table with columns for KEY, VALUE, and DESCRIPTION. The 'Body' section is also visible, showing the response in JSON format:

```
1 {
2   "PatientId": "1234",
3   "PatientName": "テスト 患者1234"
4 }
```

# フローの解説 1



Edit http in node

Delete Cancel Done

⚙ Properties

☰ Method GET ▾

🌐 URL /fhir-test/Test/:id

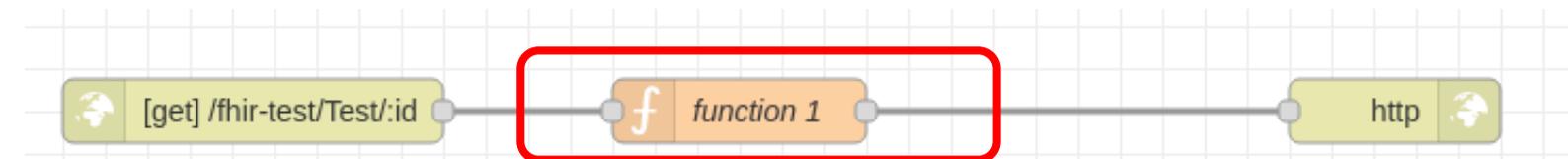
📄 Name Name

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



Properties panel for the 'function 1' node:

Name: function 1

Setup: On Start | **On Message** | On Stop

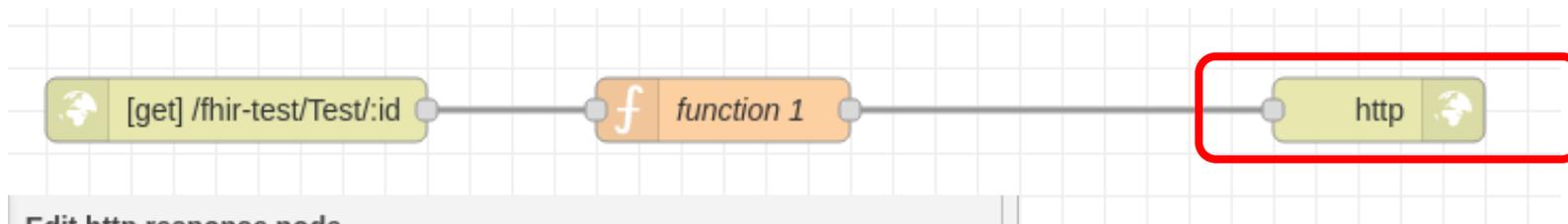
```
1 var res=new Object();
2
3 res.PatientId = msg.req.params.id;
4 res.PatientName = "テスト 患者" + msg.req.params.id;
5
6 msg.payload=res;
7 return msg;
```

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



### Edit http response node

Delete Cancel Done

**Properties**

Name: Name

Status code: msg.statusCode

Headers

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



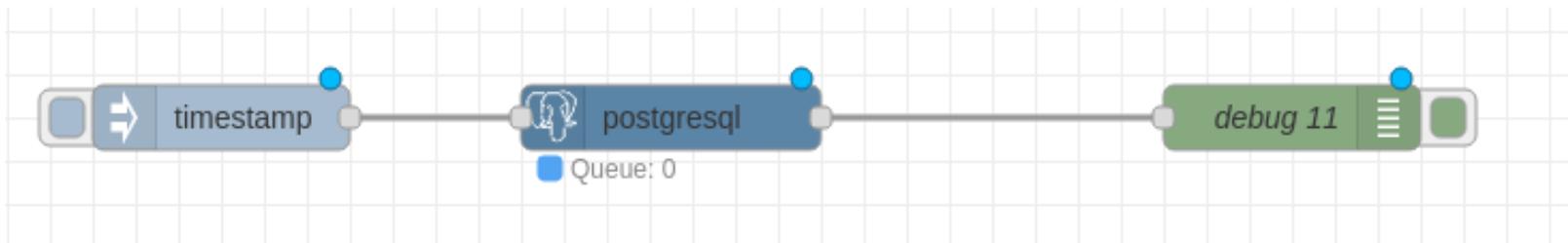
- Node-Redの画面から新規フローを作成します。



## 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。

Edit postgresql node

Delete Cancel Done

**Properties**

Name

Server  

Split results in multiple messages

Number of rows per message

Query

```
1 SELECT * FROM PATIENT
2 WHERE PATIENT_ID='00000011';
```



Edit postgresql node > Edit postgresSQLConfig node

Delete Cancel Update

**Properties**

Name

**Connection**

Host

Port

Database

SSL

**Security**

User

Password

# SQLの記載



- postgresql Nodeの設定で実行するSQLを設定します。

Edit postgresql node

Delete Cancel Done

Properties

Name Name

Server emrdb

Split results in multiple messages

Number of rows per message 1

Query

```
1 SELECT * FROM PATIENT
2 WHERE PATIENT_ID='00000011';
```

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。

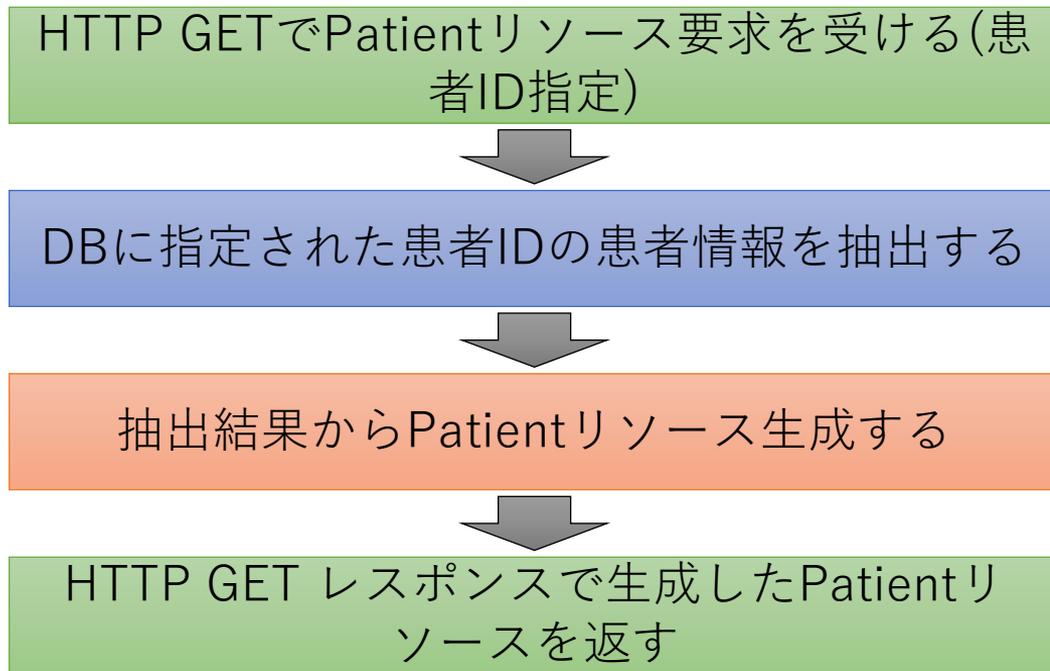
デバッグウィンドウはここをクリックすると表示します。

```
6/5/2024, 4:48:02 PM node: debug 11
msg.payload: array[1]
▼ array[1]
▼ 0: object
  patient_id: "00000011"
  last_name: "テスト"
  first_name: "患者1"
  middle_name: ""
  last_kana_name: "テスト"
  first_kana_name: "かご +1"
  middle_kana_name: ""
  gender: "F"
  birth_date:
    "2000-09-30T15:00:00.000Z"
  zip_code: "860-8556"
  address: "熊本県熊本市本荘1丁目1-1"
  tel_no: "373-5527"
  cellular_phone_no: ""
  e_mail_address: ""
```

## 処理の流れの確認



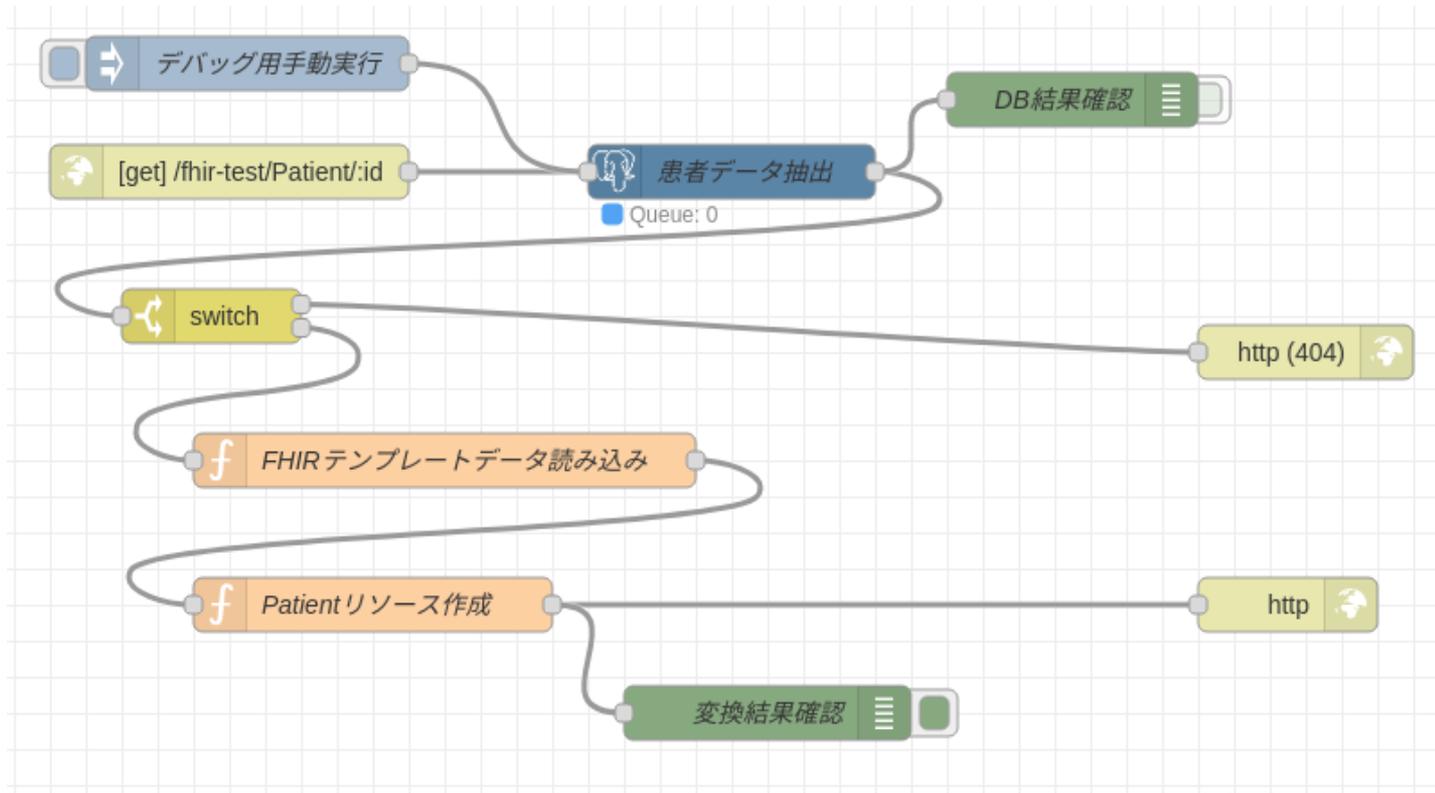
- 基本的にはハンズオン1とハンズオン2の組み合わせとなりますがこれにFHIRリソースへの変換が入ります。大まかな処理の流れは次のようになります。





## コンポーネントの配置

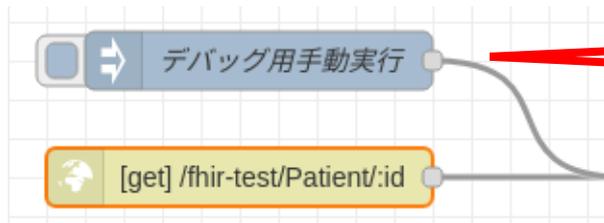
- 処理概要に沿って各種コンポーネントを配置します。



# 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。

**Edit http in node**

Delete Cancel Done

**Properties**

Method GET

URL /fhir-test/Patient/:id

Name Name

**Edit inject node**

Delete Cancel Done

**Properties**

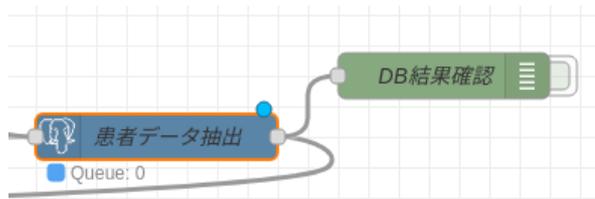
Name デバッグ用手動実行

msg.req.params.id = 00000011

## 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。



### Edit postgresql node

Delete Cancel Done

#### Properties

Name: 患者データ抽出

Server: emrddb

Split results in multiple messages

Number of rows per message: 1

#### Query

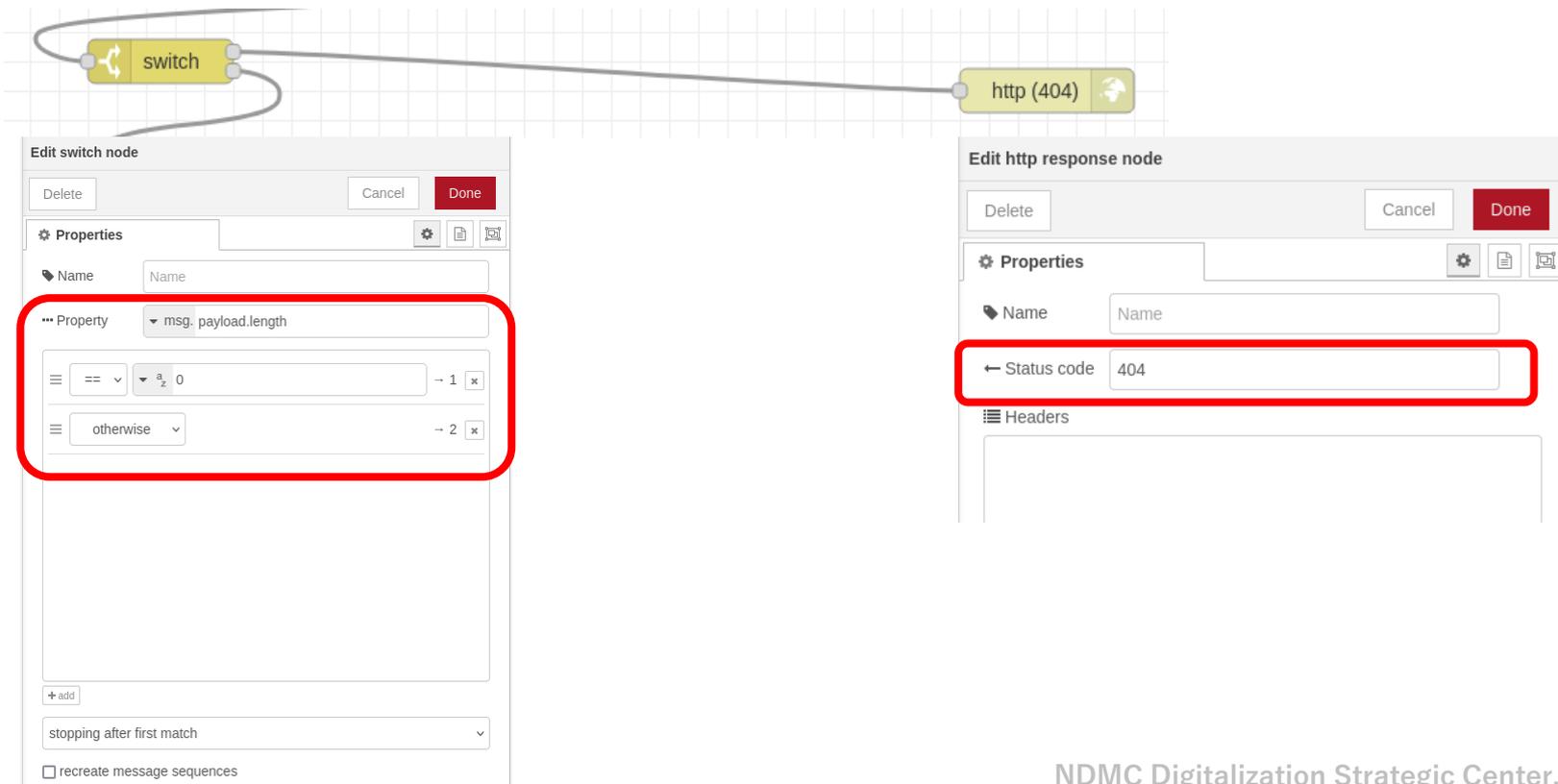
```
1 SELECT * FROM PATIENT
2 WHERE PATIENT_ID = '{{ msg.req.params.id }}'
3 ;
```

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。



```

Edit function node
Delete Cancel Done
Properties
Name FHIRテンプレートデータ読み込み
Setup On Start On Message On Stop
1 var basepath = "/home/user/hands-on202406/fhir-resources/"
2
3 flow.set("JsonStr_Patient", fs.readFileSync(basepath + "Pati
4 flow.set("JsonStr_Patient-KanjiName", fs.readFileSync(basepa
5 flow.set("JsonStr_Patient-KanaName", fs.readFileSync(basepat
6 flow.set("JsonStr_Patient-Address", fs.readFileSync(basepath)
7 flow.set("
8
9 return msg;

```

モジュールのインポートは Setupタブで定義します。

Setup On Start On Message On Stop

Outputs 1

Modules

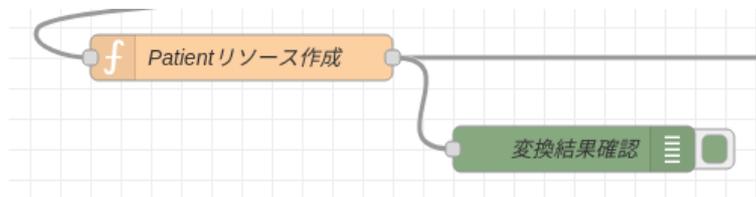
| Module name | Import as |
|-------------|-----------|
| fs          | fs        |

「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。



```
Edit function node
Delete Cancel Done
Properties
Name Patientリソース作成
Setup On Start On Message On Stop
1 var jsonstr = flow.get("JsonStr_Patient");
2 var pat = JSON.parse(jsonstr);
3
4 var row=msg.payload[0];
5
6 pat.id=row.patient_id;
7 pat.identifier[0].value = row.patient_id;
8
9 jsonstr = flow.get("JsonStr_Patient-KanjiName");
10 var name_kj = JSON.parse(jsonstr);
11 name_kj.text = row.last_name + " " + row.first_name;
12 name_kj.family = row.last_name;
13 name_kj.given.push(row.first_name);
14 pat.name.push(name_kj);
15
16 jsonstr = flow.get("JsonStr_Patient-KanaName");
17 var name_kn = JSON.parse(jsonstr);
18 name_kn.text = row.last_kana_name + " " + row.first_kana_na
19 name_kn.family = row.last_kana_name;
20 name_kn.given.push(row.first_kana_name);
21 pat.name.push(name_kn);
22
23
24 if(row.address != ""){
```

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

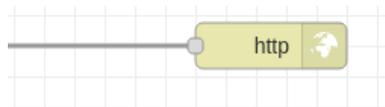


メッセージオブジェクトのpayloadにPatientをセット

# 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。



Edit http response node

Delete Cancel Done

⚙ Properties

👤 Name

← Status code

☰ Headers

+ add

The messages sent to this node **must** originate from an *http input* node

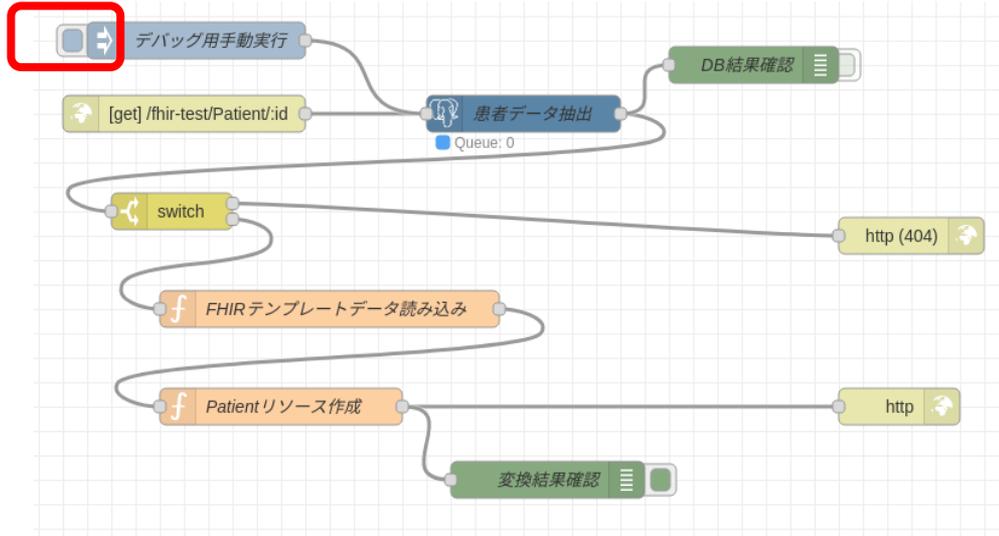
Enabled

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。



debug

all nodes all

6/6/2024, 3:36:52 PM node: DB結果確認  
msg.payload : array[1]  
▶ [ object ]

6/6/2024, 3:36:52 PM node: aa65f2f1be3aa8b6  
msg : string[8]  
"応答がありません"

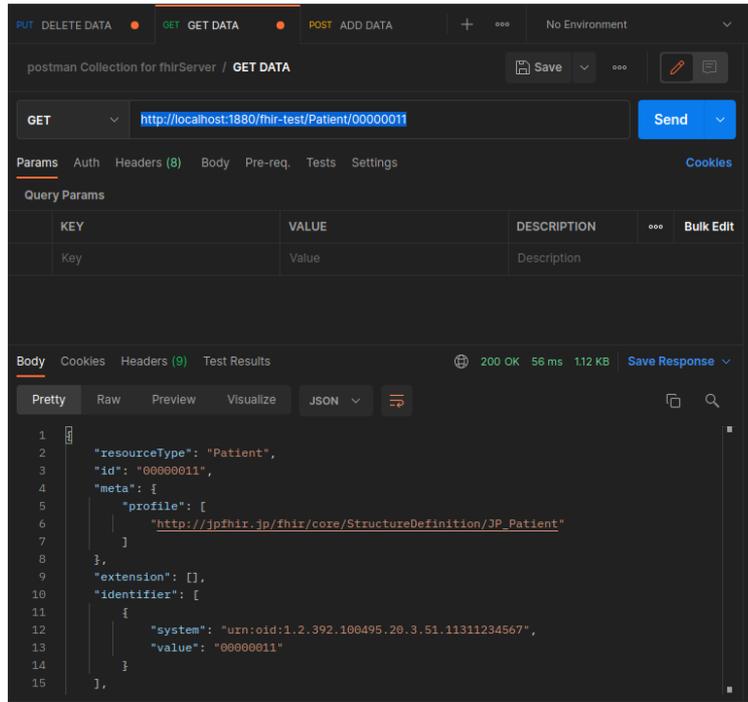
6/6/2024, 3:36:52 PM node: 変換結果確認  
msg.payload : Object

```
▼ object
  resourceType: "Patient"
  id: "00000011"
  ▶ meta: object
    extension: array[0]
  ▶ identifier: array[1]
  ▶ name: array[2]
  ▶ telecom: array[1]
  gender: "female"
  birthDate: "2000-10-01"
  ▶ address: array[1]
```

# 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

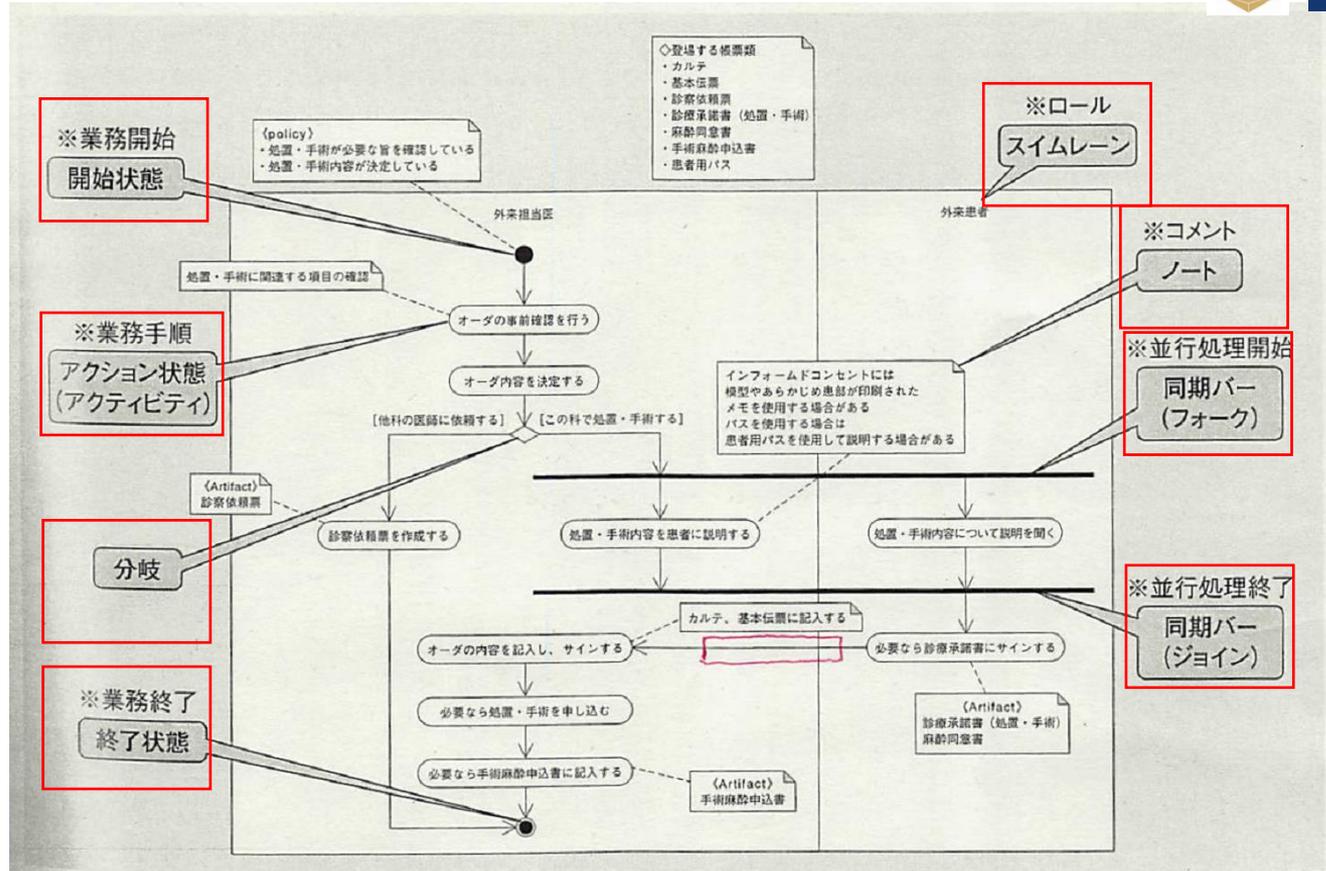


図8-1 アクティビティ図の構成要素

# 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。

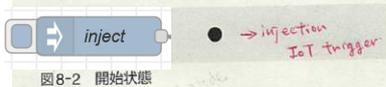


図8-2 開始状態

### (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。

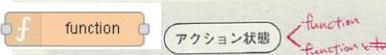


図8-4 アクション状態

### (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。

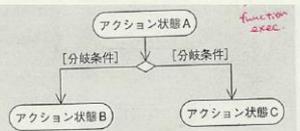


図8-6 分岐

### (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。

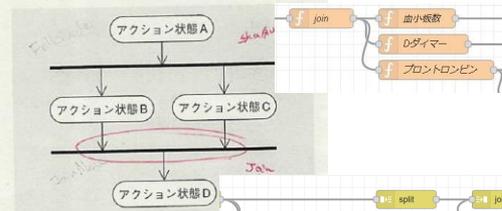
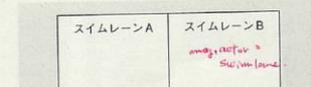


図8-7 同期バー

### (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。

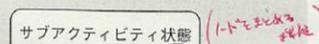


図8-9 サブアクティビティ状態

functionの集約可能

### (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、被線のアシッドでそれが受け渡される方向を記述する。

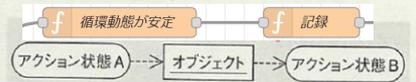


図8-10 オブジェクトフロー状態

### (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。



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。特定の状況を感じ取る範囲を、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。

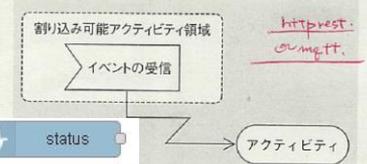


図8-12 割り込み可能アクティビティ領域

### (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。

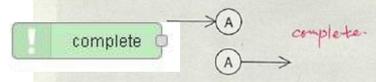


図8-13 コネクタ

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>

GitHub Gist Search... All gists Back to GitHub Sign in Sign up

Instantly share code, notes, and snippets.

ryumtym / node-red.md Secret 0 Stars 0 Forks

Code Revisions 9 Embed <script src="https://... Download ZIP

node-red.md Raw

### 電子カルテと業務革新—医療情報システム構築における業務フローモデルの活用

TI-023: 投薬実施(内服・外用)プロセスの大まかな流れ

1. 受け持ち看護師のプロセスが開始
2. 受け持ち看護師のプロセス完了後、入院患者と服薬介助係のプロセスが同時に開始する
3. 患者と服薬介助係のプロセス終了後、看護師が実施入力を行い、TI-023フロー完了。

The flowchart illustrates the medication administration process (TI-023) across three swimlanes: 受け持ち看護師 (Nurse), 入院患者 (Patient), and 服薬介助係 (Medication Assistant). The process starts with the nurse checking the patient's condition for medication administration. A decision diamond labeled [患者管理業務] (Patient Management) leads to a step [看護管理業務] (Nursing Management) where the nurse checks the patient's name and order details. This step involves the patient. Following this, the nurse provides instructions and obtains consent. The medication assistant then administers the medication, also involving the patient. The process concludes with the nurse's final input.

※内服薬、外用薬を対象とする



# 要素の解説

---



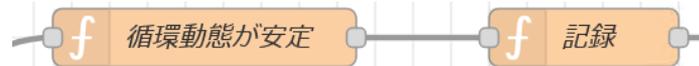
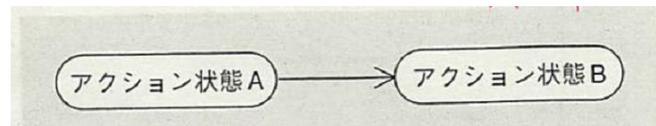
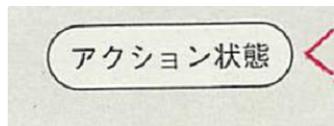
## (1)開始状態

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。

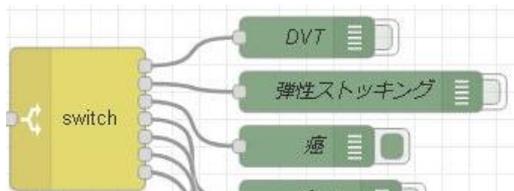
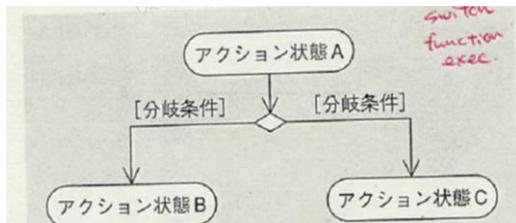


## (3) アクション状態(アクティビティ)

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。

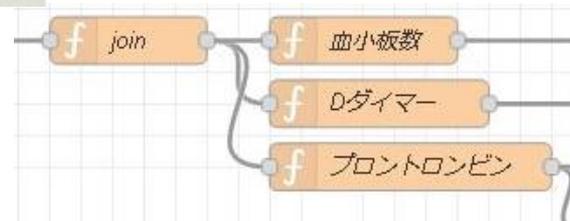
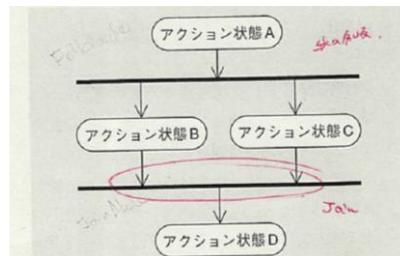
## (4) 遷移

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。



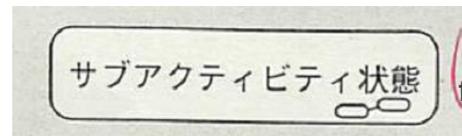
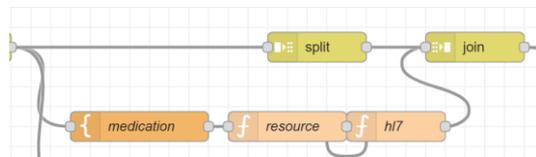
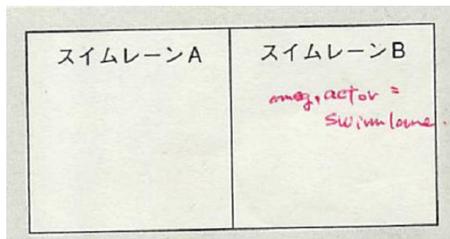
## (5)分岐

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棒で記述する。



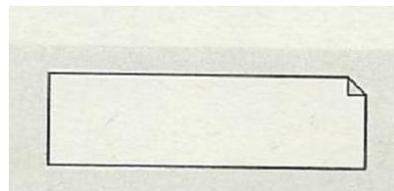
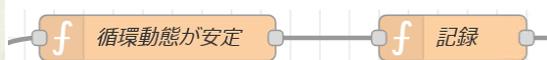
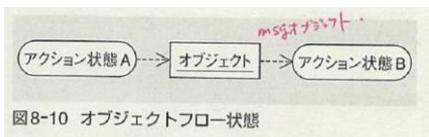
functionの集約可能

## (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。



## (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。

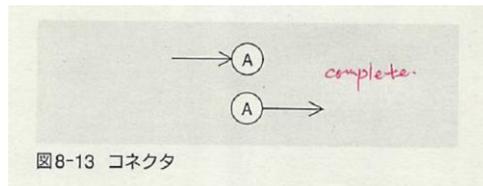
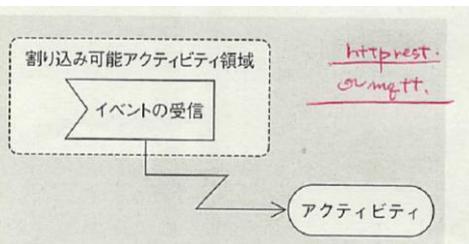


図8-13 コネクタ



## (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。

## (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

---



- 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言語風と言えば「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

---



- ターミナルよりnode-redとタイプ



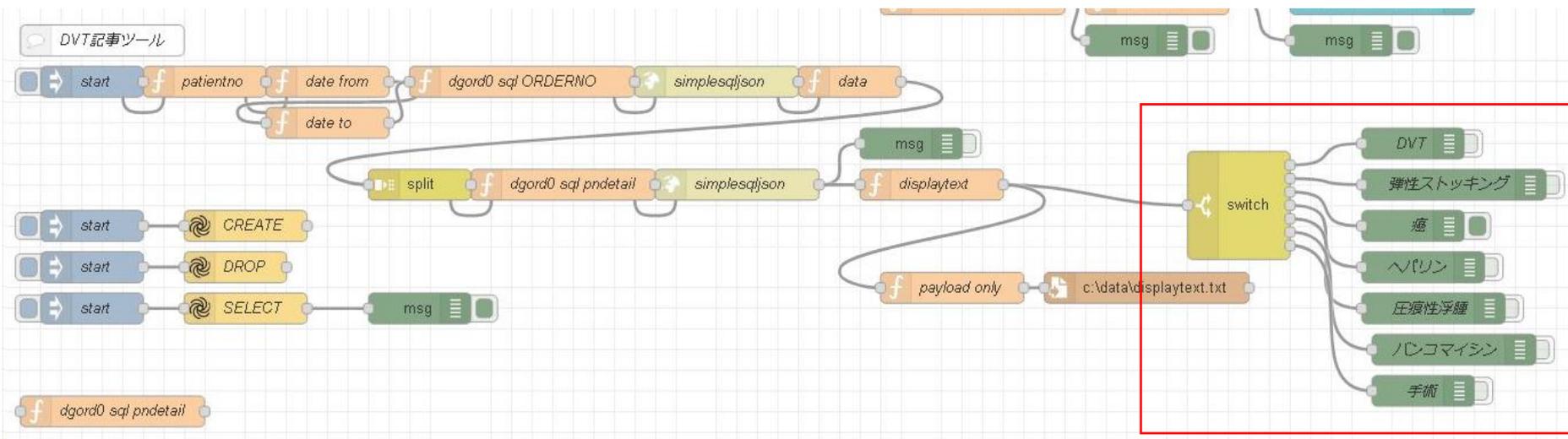
# Node-REDフロー組み立て例

---

# 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

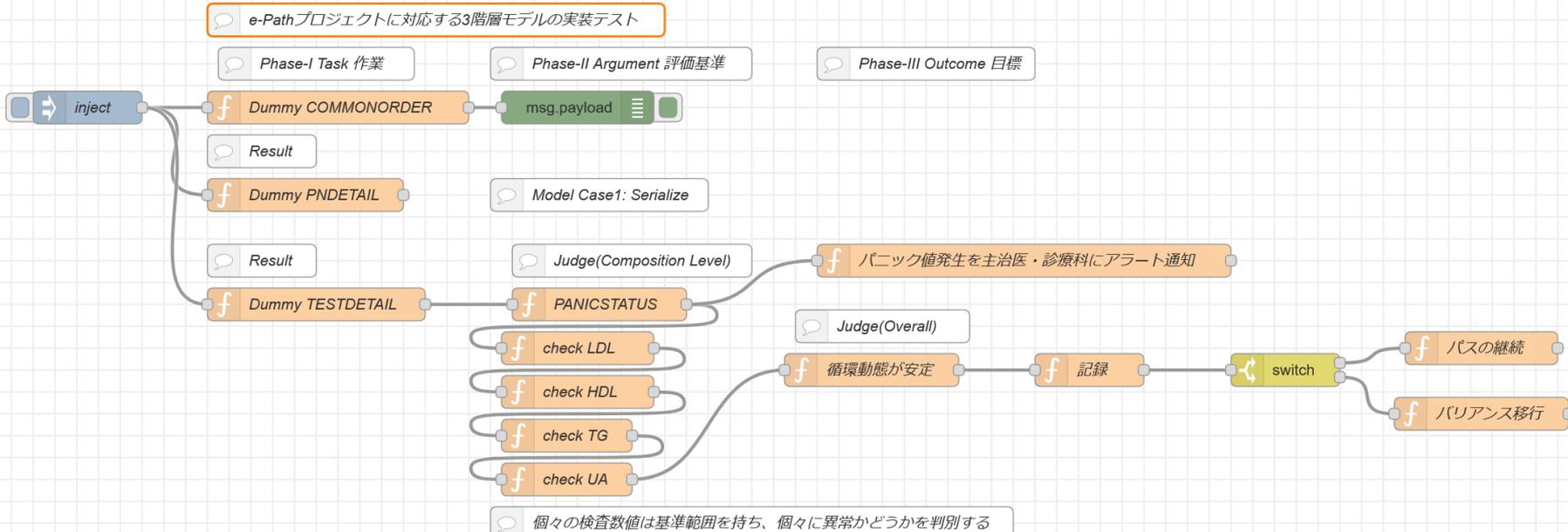
```
function ノードを編集
  削除 中止 完了
  プロパティ
  名前 Dダイマー
  初期化処理 コード 終了処理
  1 var result = msg.resdetail.filter(function(item,index){
  2   var str = item.TESTITEMCODE;
  3   if(str.indexOf('227300') > -1){return true;}
  4   //以下条件を書き連ねてよい(OR)
  5 });
  6 msg.out=result;
  7 return msg;
```

# 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

フロー 1

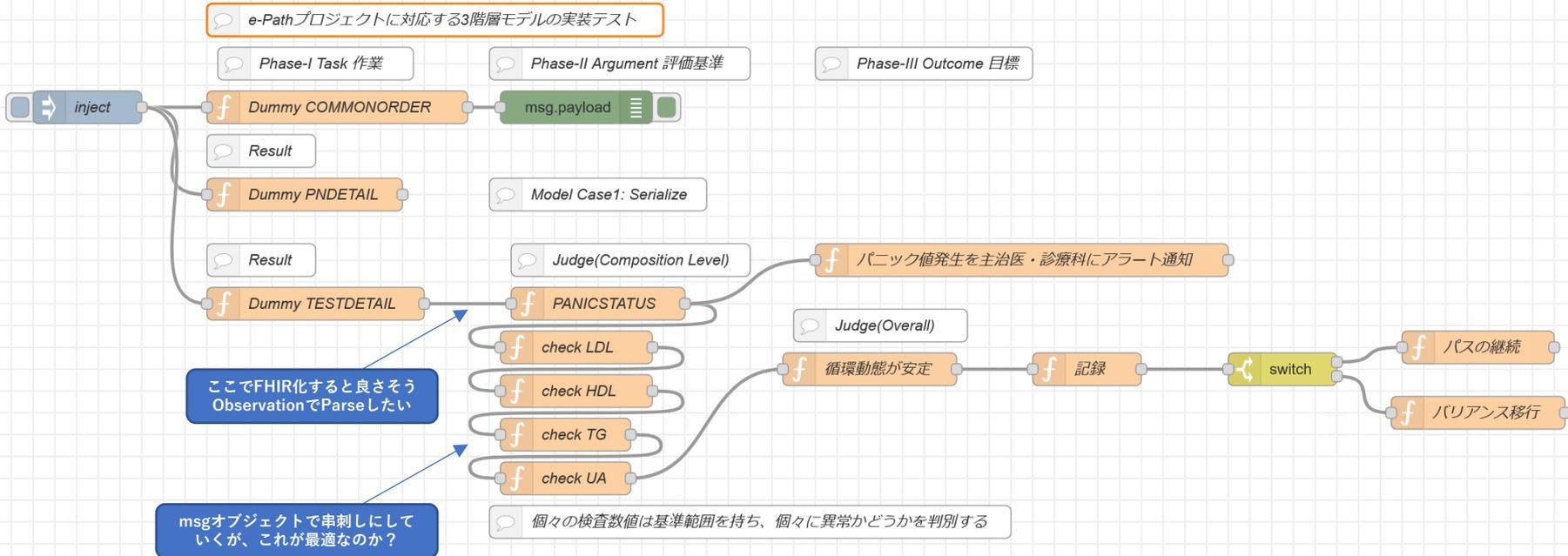


# IPCIアーキテクチャによるe-Path3階層の実装テスト



## NEC MegaOAKHRとのオーダリングDB接続を実現

フロー 1



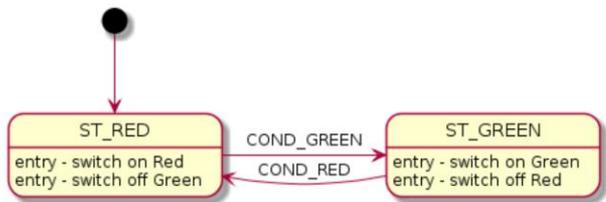
# Node-REDとUMLに特徴的なFinite State Model(FSM)の実装性



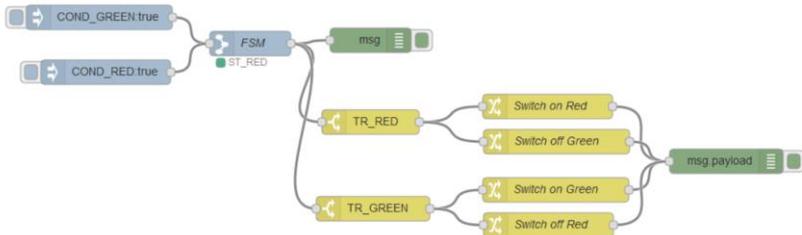
## Example

### 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 startup. The transitions between the two states are triggered by COND\_GREEN and COND\_RED respectively.



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.

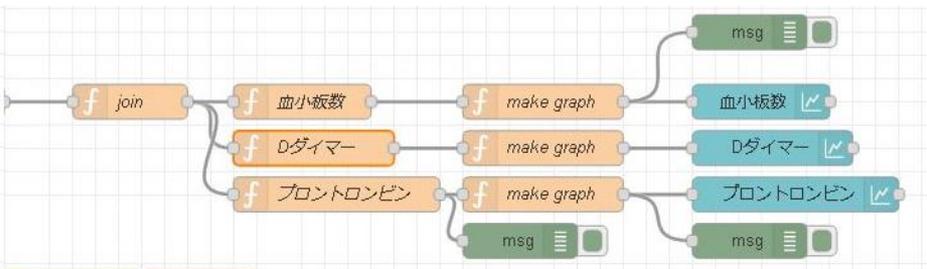
| Name     | Source   | Destination | Condition  | Cond. State |
|----------|----------|-------------|------------|-------------|
| TR_GREEN | ST_RED   | ST_GREEN    | COND_GREEN | true        |
| TR_RED   | ST_GREEN | ST_RED      | COND_RED   | true        |

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

```
1 var temp = [{
2   "series": ["prontronbin"],
3   "data": [[]],
4   "labels": [""]
5 }];
6 for(var i = 0; i < msg.out.length; i++){
7   var plotdata = {"x" : msg.out[i].UPDATEDATE, "y" : parseFloat(msg.out[i].TESTRESULT)};
8   temp[0].data[0][i] = plotdata;
9 }
10
11 msg.payload = {};
12 msg.payload = temp;
13 return msg;
```

## DVT検査指標

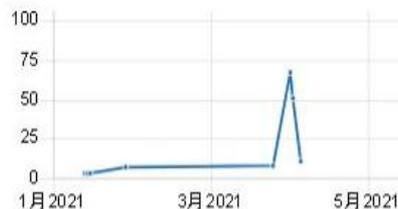
プロトロンビン

プロトロンビン



Dダイマー

Dダイマー



血小板数

血小板数

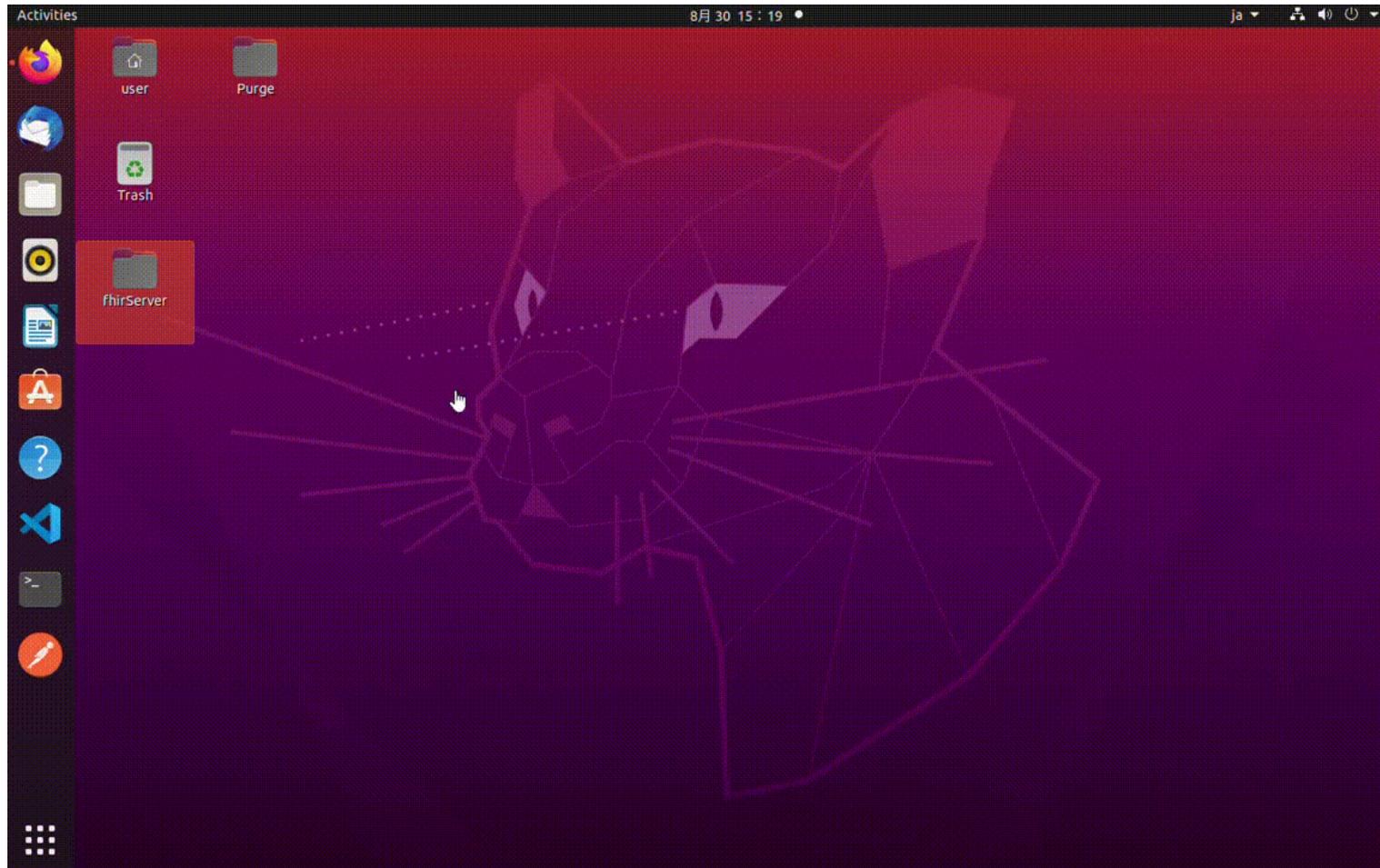




# IPCI-FHIRサーバーの起動と利用

---

# 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

---





route一覧

The screenshot shows the Visual Studio Code editor with two files open. The left file, `app.js`, contains Express.js route definitions for various endpoints, such as `var createError = require('http-errors');` and `var irisouttestRouter = require('./routes/irisouttest');`. The right file, `simplesqlsontofile.js`, shows the implementation of the `router.get('/', function(req, res) {` endpoint, which logs the start time and executes a SQL query to retrieve data from a table named `DISPLAYTEXT`. The interface includes the Explorer sidebar on the left, the main editor area, and the Output/Terminal panel at the bottom.

app.js

simplesqlsontofile.js



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<br>アプリケーション<br>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

ユーザーインターフェース  
(テキスト/ボタン)

医療ロジックの記述

OS依存開発環境  
(例: MS.NET)

OS依存ドライバ

OS

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)

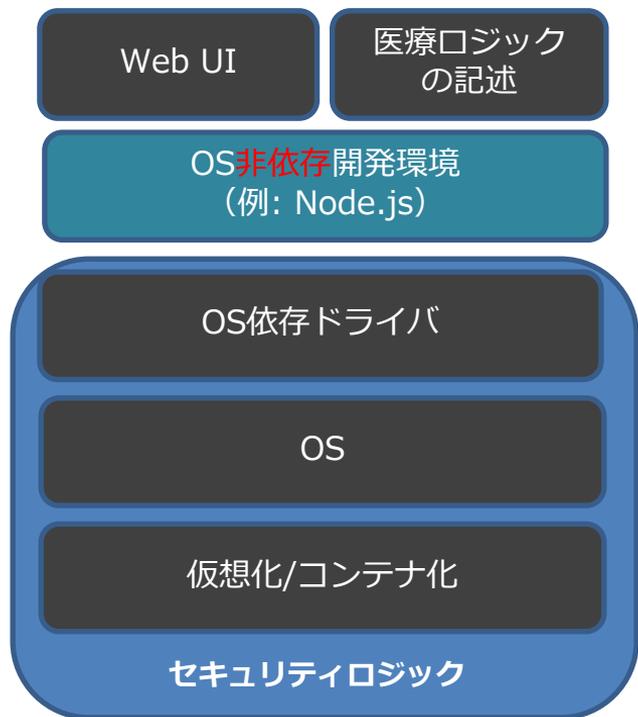
また現場での「迅速な」実装を求められた際、Doc-Viewはリファクタが困難であった→オブジェクト指向自身の性質に由来

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)

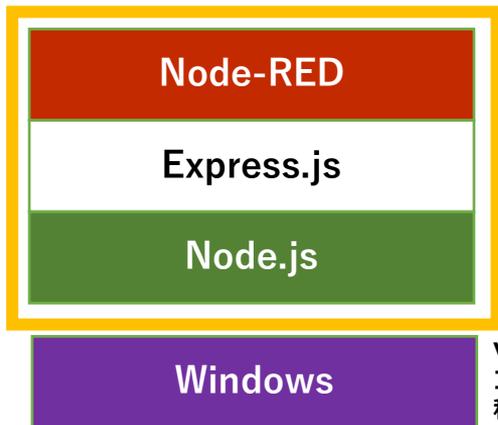
# OSS(特にOpenJS系)の特徴と開発方針



- ・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
- ・言語仕様がかわらない、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-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

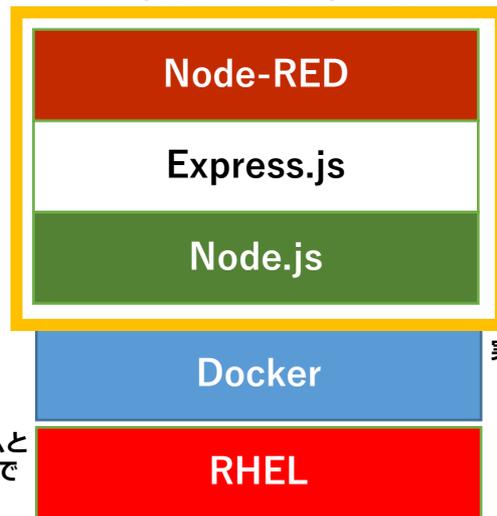
RESTサービス全体（サーバ間通信設定）まで保存したい場合に  
Docker（必要に応じてK8）を使う→プログラムの永続性



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



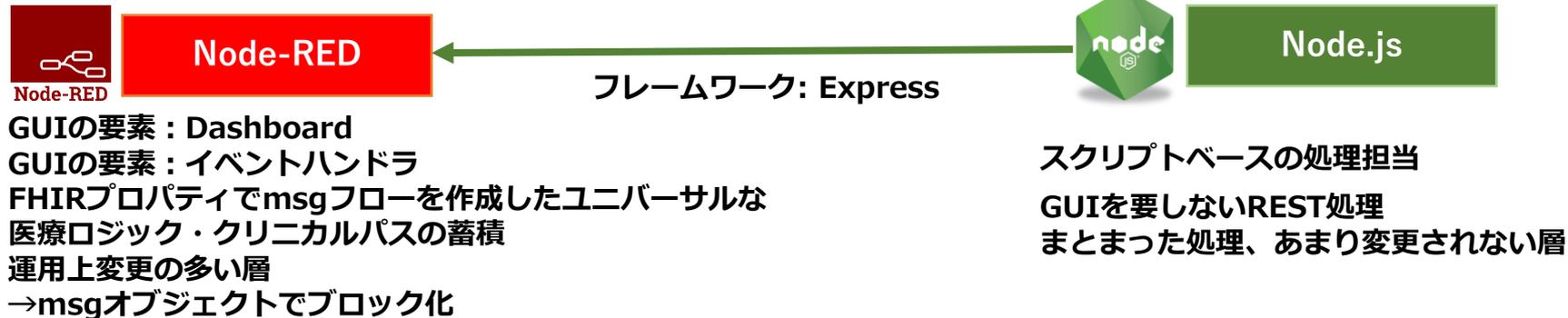
実行速度の低下が少ない  
(有償版ではVMのハイパーバイザ)

**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



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 :

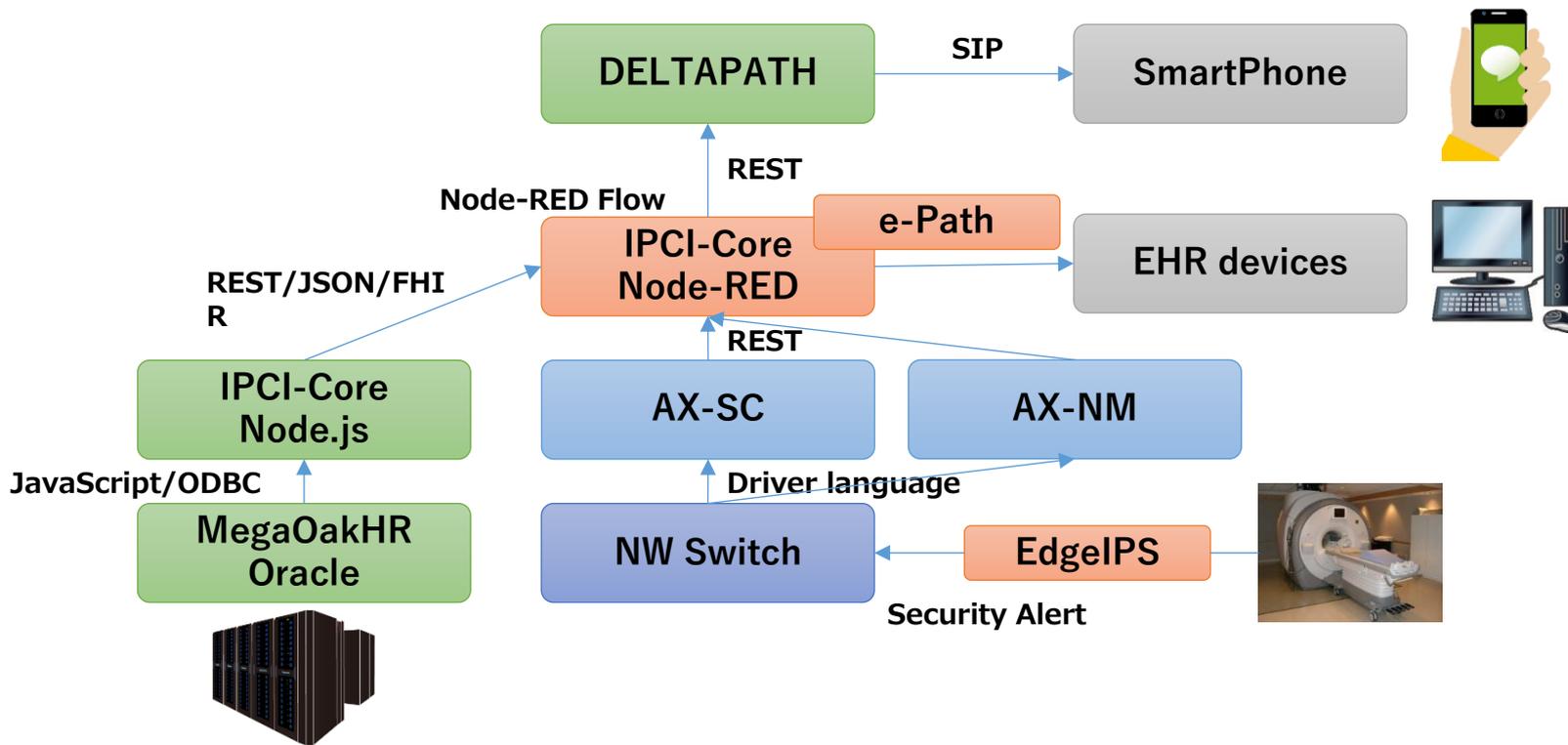
PythonではJSONネイティブな書き方が存在するが、UIの生成については別途HTML/CSS/JavaScriptを組むことになる→処理文法が増える欠点あり

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



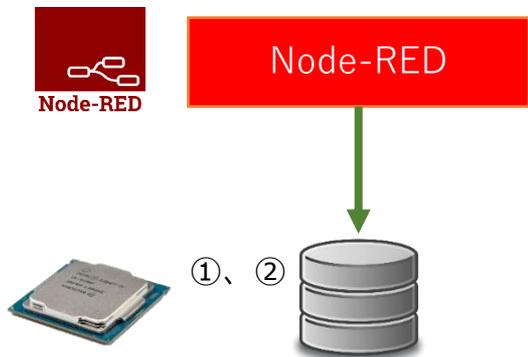
# 実装上の工夫



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)

Node-REDサーバは一つとは限らない

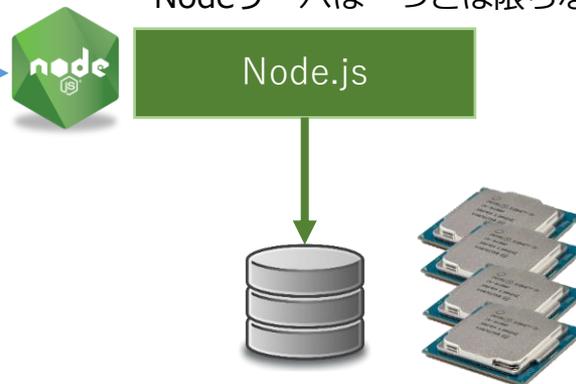


①②は高レスポンスを強く求める場合

③

ネットワーク性能が決定的に  
レスポンスに影響する  
→Webプログラミングがもつ  
普遍的性質

Nodeサーバは一つとは限らない



③は処理の重いタスクを  
分担させるのに向いている



# 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

Windows UIや  
既存オーダーの仕組みのみ  
再利用

電子カルテ



IPCI-Node  
Node-RED



DELTAPATH frSIP  
RESTでメッセージ生成、通話  
音声認識インターフェース  
スマートフォン通知



ロードバランサーとしても機能する

Node-RED



Node-RED



Node-RED



例：医師がスマートフォンで承認する際  
には、操作者の権限とともに、  
なりすまし防止のトラストエンジン  
(操作権限) チェックが同時に必要

位置情報をRESTで収集



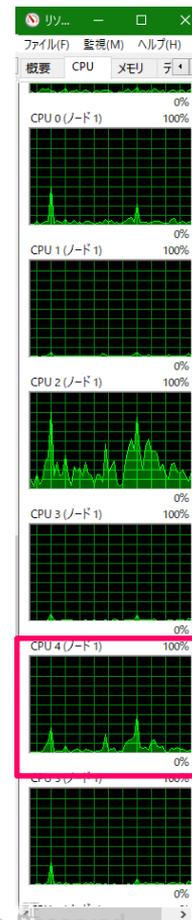
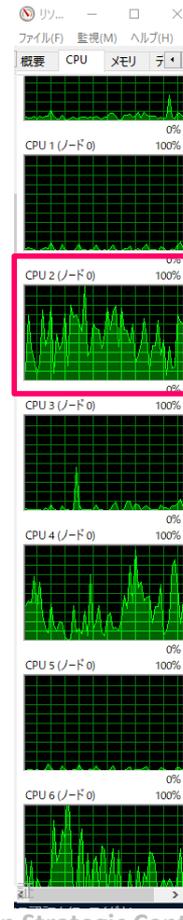
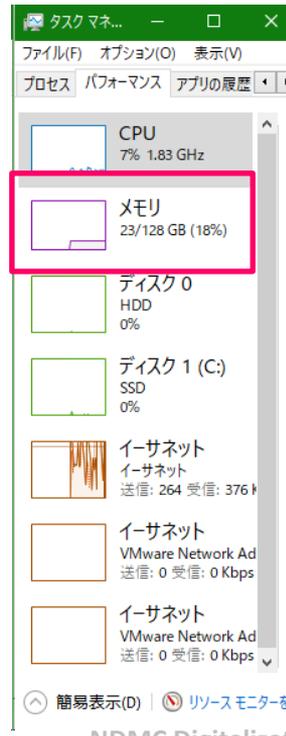
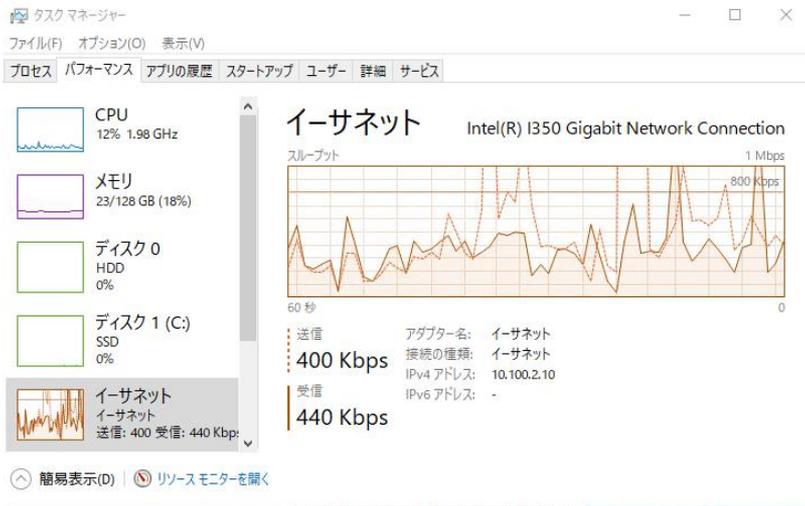
Alaxala



# 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++<br>bash          | Windows Form<br>マウス操作/RPA | LabView<br>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<br>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<br>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

f function    python shell

|            | 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に近い)         | ×                                  | ×        | (さまざま)          |
| 中核サーバ      | Express                            | http.server<br>Flask   | ×                  | ×                                  | ×        | (さまざま)          |
| 単体アプリケーション | Electron<br>Angular                | Kivy<br>Tkinter<br>PyQt<br>wxPython<br>PySimpleGUI<br>→PyInstaller | .bat<br>MS-DOS窓で可能 | VBへの移植以外不可<br>(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



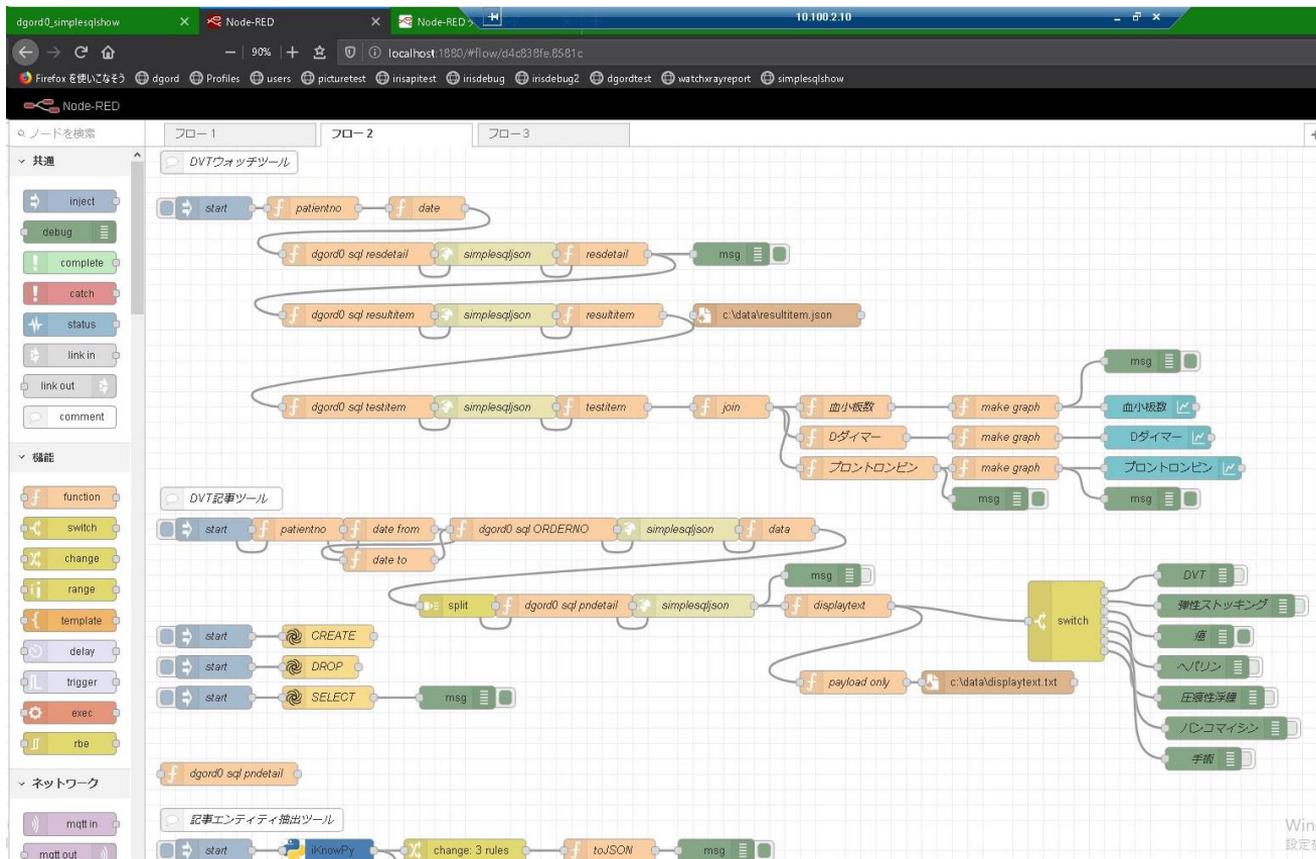
InterSystemsがGitで公開しているiKnowPyを呼び出し（ファイルで渡し、結果はmsg.payloadで戻せる）  
将来的な機械学習機能の実装にも対応できる

The screenshot displays the Node-RED web interface. The main workspace shows a workflow with several nodes. A node labeled 'iKnowPy' is highlighted with a red box. The right-hand panel, titled 'python-shell ノードを編集', shows the Python code for this node:

```
1 import iknowpy
2 import sys
3 import codecs
4 import json
5 engine = iknowpy.iKnowEngine()
6
7 #sys.stdout = codecs.getwriter('utf-8')(sys.stdout)
8 #reload(sys)
9 #sys.setdefaultencoding('cp932')
10
11 #index text
12 text = '新型コロナウイルスの121件の感染拡大に伴い、'
13 f = codecs.open('C:\data\displaytext.txt', 'r', 'utf-8')
14 text = f.read()
15 engine.index(text, 'ja')
16 f.close()
17 text2 = json.dumps(engine.m_index)
18
19 print(engine.m_index)
```

# 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



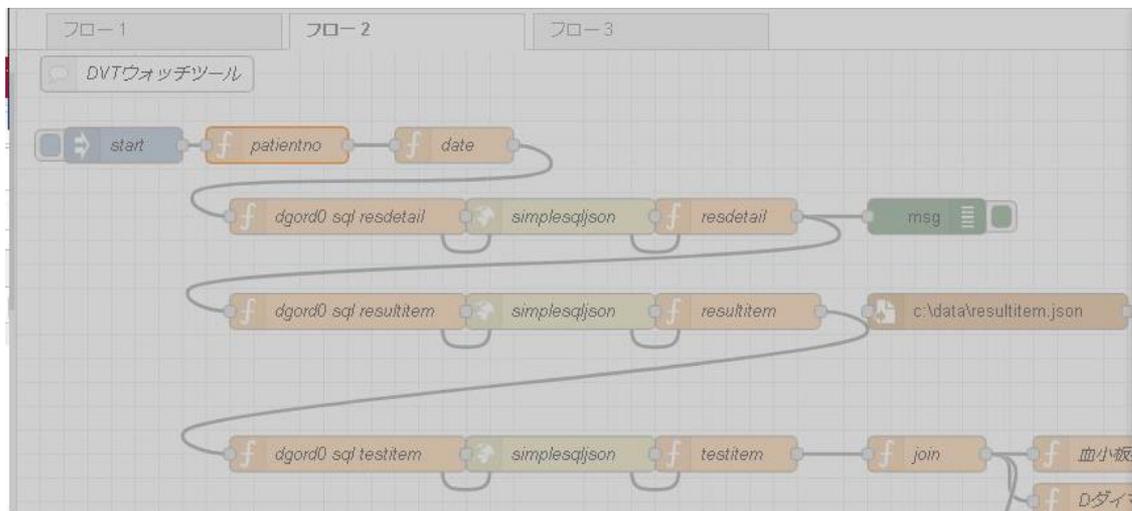
# プロトタイプオブジェクト指向の特長



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



function ノードを編集

削除 [中止] [完]

プロパティ

名前 patientno

初期化処理 コード 終了処理

```
1 msg.pid = '06219282';
2 return msg;
```

**患者ID用にmsg.pidを動的に指定  
日付はmsg.dateで指定している**

# SQLのみをロジック化する手法

msg.payload中にSQL構文を記述



```
function ノードを編集
  削除
  中止
  完了
  プロパティ
  名前: dgord0 sql resdetail
  初期化処理
  コード
  終了処理
  1 sqlstr = `
  2 SELECT
  3 *
  4 FROM RESDETAIL
  5 WHERE
  6 PATIENTNO = '' + msg.pid + ''
  7 AND
  8 UPDATEDATE > TO_DATE('' + msg.date + '', 'YYYY-MM-DD')
  9 ;
  10 `;
  11
  12 msg.payload = { "word": sqlstr };
  13 return msg;
```

# 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



ODBC Connector

http request ノードを編集

URL: http://localhost:3000/simplesqjson

出力形式: JSONオブジェクト

名前: simplesqjson

# msg.resdetailの生成



msg.resdetailにオブジェクトを載せ替え

The image shows a workflow editor interface with a grid of process blocks. The workflow starts with a 'start' block, followed by 'patientno' and 'date' filters. It then branches into three parallel paths: 'dgord0 sql resdetail', 'dgord0 sql resultitem', and 'dgord0 sql testitem'. Each path uses a 'simplesqljson' block to process data. The 'resdetail' path leads to a 'resdetail' block, which is highlighted with a red box in the code editor. The 'resultitem' path leads to a 'resultitem' block, and the 'testitem' path leads to a 'testitem' block. The workflow concludes with a 'join' block and a 'msg' block.

The code editor on the right shows the following code for the 'resdetail' function:

```
1 msg.resdetail = msg.payload;  
2 return msg;
```

# 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

The screenshot shows the Node-RED web interface. The main workspace contains a flow with the following nodes: `start`, `patientno`, `date`, `dgord0 sql resdetail`, `simplesqljson`, `resdetail`, `msg`, `dgord0 sql resultitem`, `simplesqljson`, `resultitem`, `c:\data\resultitem.json`, `dgord0 sql testitem`, `simplesqljson`, `testitem`, and `join`. The `join` node is connected to three output nodes: `血小板`, `Dダイマー`, and `フロン`. The right-hand panel, titled "function ノードを編集", shows the code for the selected function node:

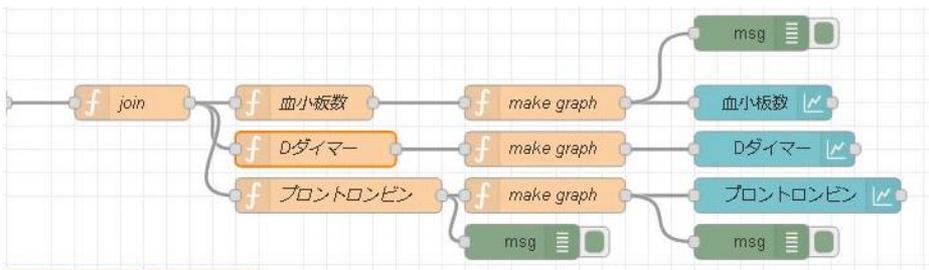
```
function ノードを編集
  1 var result = msg.resdetail.filter(function(item,index){
  2   var str = item.TESTITEMCODE;
  3   if(str.indexOf('227300') > -1){return true;}
  4   //以下条件を書き連ねてよい(OR)
  5 });
  6 msg.out=result;
  7 return msg;
```

# 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
1 var temp = [{
2   "series": ["prontronbin"],
3   "data": [],
4   "labels": [""]
5 }];
6 for(var i = 0; i < msg.out.length; i++){
7   var plotdata = {"x" : msg.out[i].UPDATEDATE, "y" : parseFloat(msg.out[i].TESTRESULT)};
8   temp[0].data[0][i] = plotdata;
9 }
10
11 msg.payload = {};
12 msg.payload = temp;
13 return msg;
```

## DVT検査指標

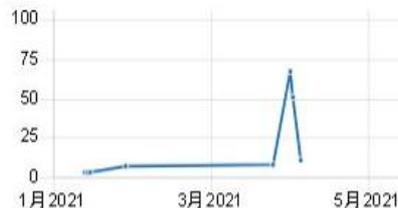
プロトロンビン

プロトロンビン



Dダイマー

Dダイマー



血小板数

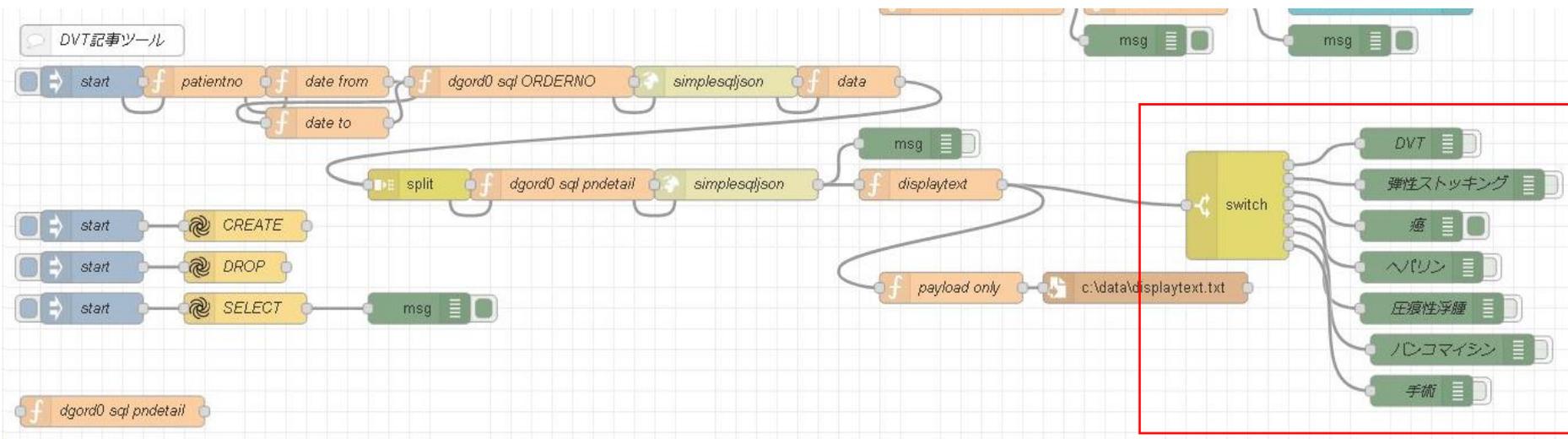
血小板数



# 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

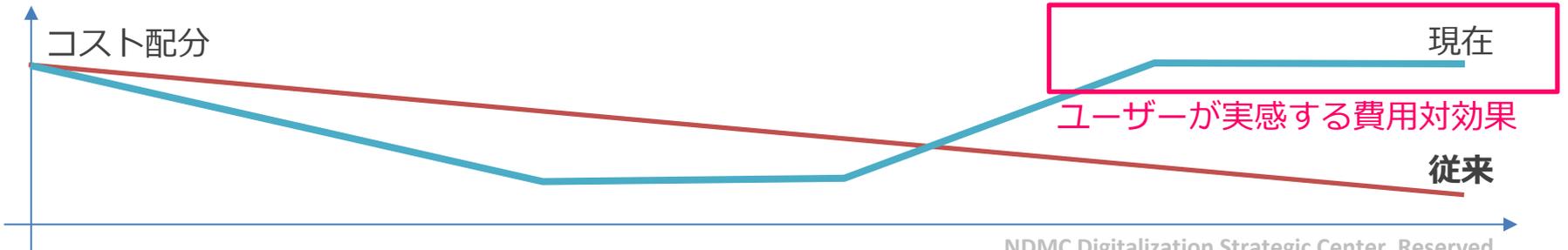
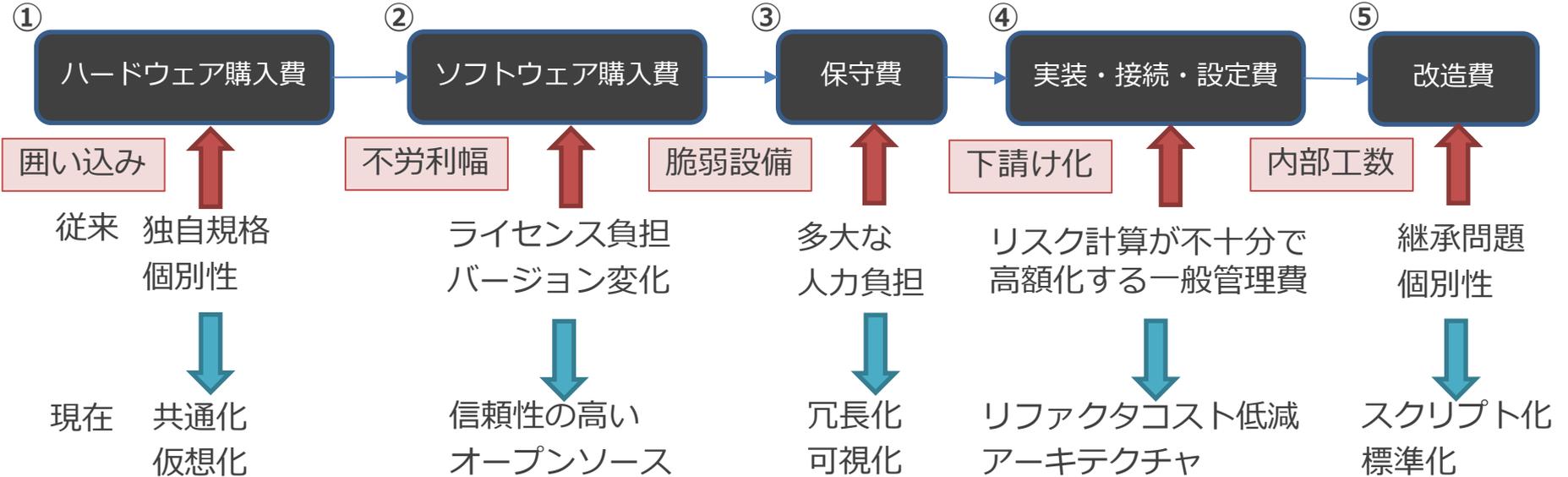
The screenshot displays the Node-RED web interface. The main workspace shows a workflow with several nodes. A node labeled 'iKnowPy' is highlighted with a red box. The right-hand panel, titled 'python-shell ノードを編集', shows the Python code for this node:

```
1 import iknowpy
2 import sys
3 import codecs
4 import json
5 engine = iknowpy.iKnowEngine()
6
7 #sys.stdout = codecs.getwriter('utf-8')(sys.stdout)
8 #reload(sys)
9 #sys.setdefaultencoding('cp932')
10
11 #index text
12 text = '新型コロナウイルスの121件の感染拡大に伴い、'
13 f = codecs.open('C:\data\displaytext.txt', 'r', 'utf-8')
14 text = f.read()
15 engine.index(text, 'ja')
16 f.close()
17 text2 = json.dumps(engine.m_index)
18
19 print(engine.m_index)
```

# 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



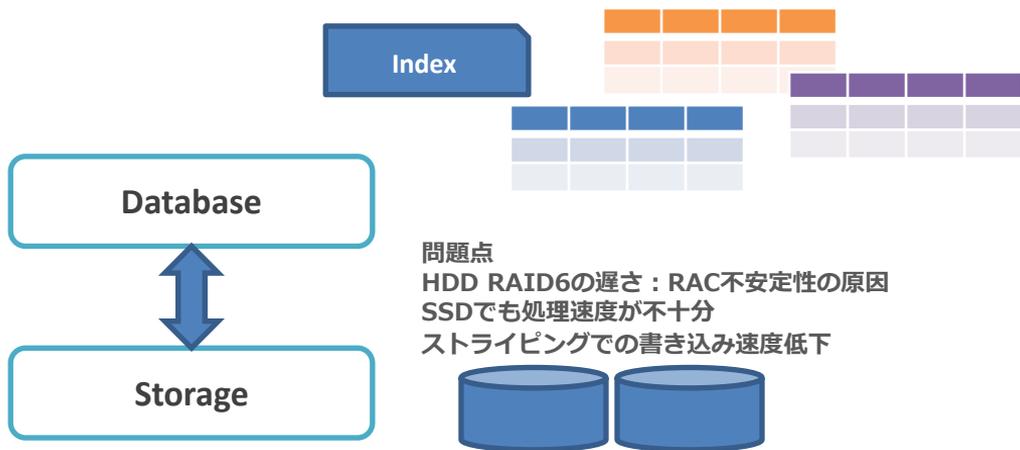
# DBのindex構築の属人性、医療データベースが抱える問題、根本改善につ



## 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

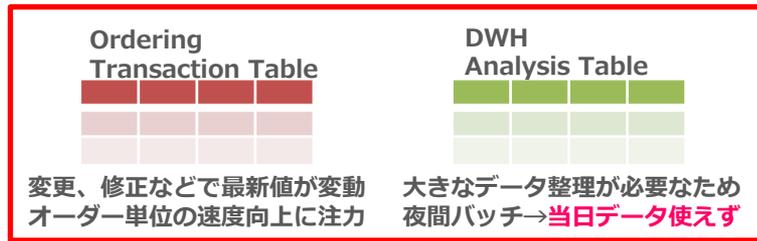
群大病院としての  
中期的データ保持形態案

大幅な改造の停止  
改造工数抑制

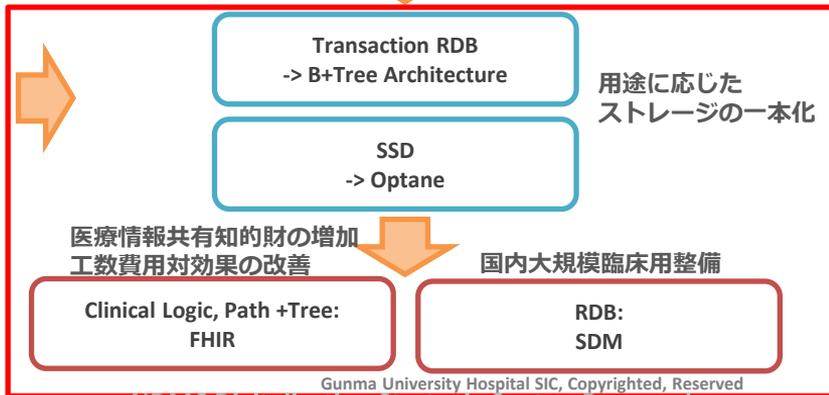
GUI/Transaction  
Vender-Made DB

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





The Tutorial is Completed  
お疲れ様でした