

「Terria を用いた PLATEAU VIEW での経路検索 Web アプリの作り方」ハンズオン補足資料

目次

#	項目
(1)	概要
1)	概要の説明、チュートリアルで実施する概要説明
2)	SWアーキテクト及び各種SWの説明
3)	歩行空間ネットワークの説明
(2)	PLATEAU VIEW1.1の基本セットアップ
1)	はじめに
2)	Docker環境の構築
3)	PLATEAU VIEW1.1の動作確認
(3)	経路検索WebAPI構築
1)	経路検索WebAPI動作に必要なSWインストール
2)	歩行空間ネットワークデータの投入
3)	歩行空間ネットワークデータGIS配信設定
(4)	PLATEAU VIEW1.1連携方法説明
1)	連携を行う為のカスタマイズについて
2)	変更したソースデプロイ、連携設定方法
3)	歩行空間ネットワークデータGIS配信設定
4)	歩行空間WebAPIアプリソース・ビルト方法
5)	歩行空間WebAPI内部ロジックの説明
(5)	動作説明

(1) 概要

1) チュートリアルの概要

- 本チュートリアル、ハンズオンのテーマは、「Terria を用いた PLATEAU VIEW での経路検索 Web アプリの作り方」です。
- PLATEAU VIEW 1.1 と [uc22-028 「エリアマネジメントダッシュボード構築」](#) のユースケースにて Web アプリとして開発した経路検索機能について、TerriaJS をカスタマイズすることで連携させます。経路検索した結果を、PLATEAU VIEW 1.1 上で

表示するまでのハンズオンです。

- 実施する内容は、以下のとおりです。

①PLATEAU VIEW1.1の構築をしてみよう
→PLATEAU VIEWを構築し、3D都市モデルを表示

②経路検索Webアプリを構築しよう
→歩行空間ネットワークを利用して経路検索を行うWebアプリを構築実施

③経路検索WebアプリをPLATEAU VIEW1.1と連携してみよう
→PLATEAU VIEW1.1上で経路検索実施できる環境構築を実施

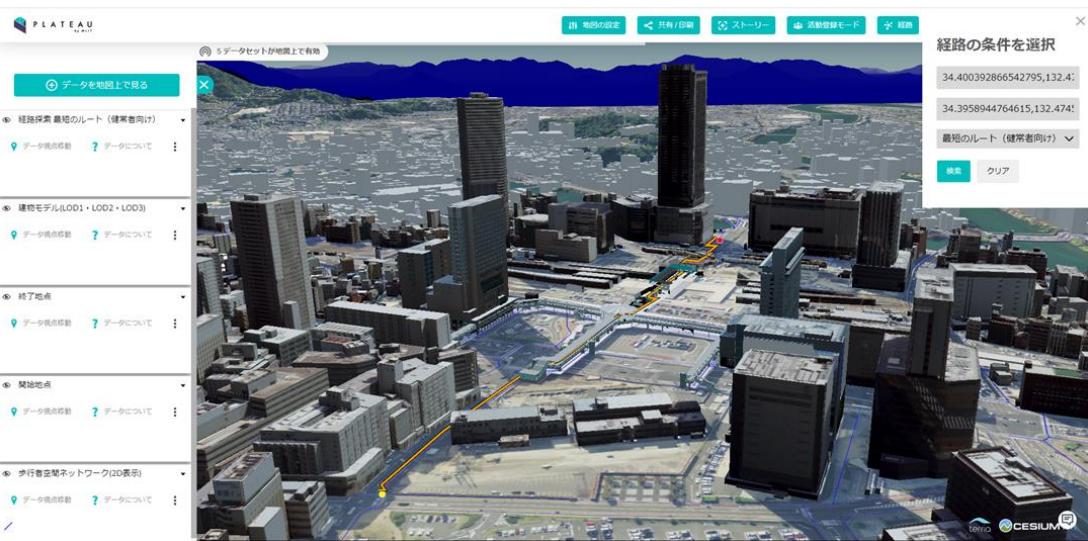
このハンズオンでカスタマイズした PLATEAU VIEW 1.1 の実行結果は、次の通りです。以下のことを実現します。

・経路の表示

GeoServer を使って、歩行空間ネットワークとして公開されている経路情報を配信し、それを PLATEAU VIEW 1.1 で重ね合わせて、車椅子利用者向け、高齢者・乳幼児連れ向け、視覚障害者向けなどに適した経路を表示できるようにします。

・経路の検索

3D 地図上でクリックして、「始点」と「終点」を選び、始点から終点に向けて、車椅子利用者向け、高齢者・乳幼児連れ向け、視覚障害者向けごとに、適した経路を検索して表示できるようにします。

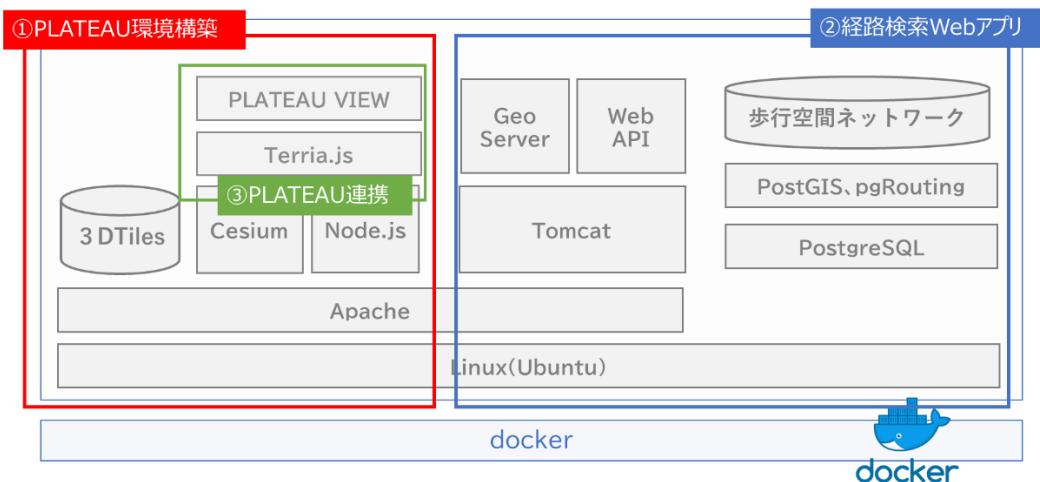


2) ソフトウェアアーキテクト及び各種ソフトウェアの説明

- アーキテクト構成図は下記の通りです。PLATEAU VIEW 1.1 環境と経路探索 Web アプリに大別されます。いずれも Linux(Ubuntu)上に構築します。今回のハンズオンでは、Docker で下図の環境をオールインワンで用意しています。これをベースに、

カスタマイズを加えていきます。

- PLATEAU VIEW 1.1 環境は、Node.js 上で稼働しています。Cesium と Terria.js を取り込んで構成されています。
- 経路探索 Web アプリは、アプリケーションとデータベースに分けられます。アプリケーションは Web コンテナである Tomcat 上で稼働しています。アプリケーションとして、OSS の GIS 配信サーバである GeoServer と、カスタム開発する WebAPI を作成します。WebAPI については、REST API を用意できれば形式は不問です。
- データベースは PostgreSQL を使用します。PostgreSQL に、位置情報を扱う拡張である PostGIS とルート検索を扱う拡張である pgRouting を追加します。PostgreSQL に歩行空間ネットワークデータを格納します。
- Web サーバとして、Apache を使用します。PLATEAU VIEW 1.1 と WebAPI、GeoServer へのリバースプロキシとして機能します。また、ドキュメントルート上に 3DTiles を格納することで、3DTiles 形式のデータを配信し、PLATEAU VIEW 1.1 で表示できるようにします。



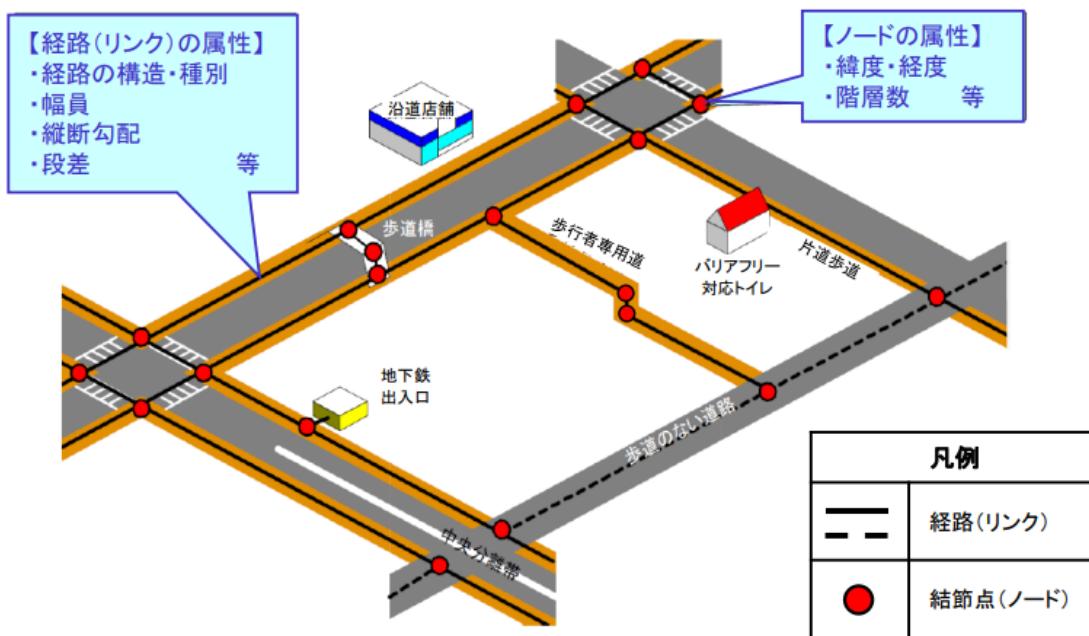
- 各ソフトウェアの詳細は、次表に記載の通りです。

ソフトウェア	バージョン	ライセンス	説明
apache	2.4.52	Apache License 2.0	Web アプリで配信を行うための Web サーバ
PLATEAU VIEW	1.1	Apache License 2.0	3D 都市モデルビューワ
Terria	3.3.4	Apache License 2.0	UI(ユーザーインターフェイス)の提供及び UI を介して Cesium の描画機能を制御するためのライブラリ
Cesium	-	Apache License 2.0	3D 都市モデルビューワ上にデータを描画するためのライブラリ
Node.js	16.16.0	MIT License	3D 都市モデルビューワの実行環境
GeoServer	2.20.4	GNU GENERAL PUBLIC LICENSE Version 2	各種データを WMS 及び WFS などで配信するための GIS サーバ
Tomcat	9.0.75.0	Apache License 2.0	GeoServer、metabase、カスタムアプリを起動する J2EE の SDK
Spring boot	-	Apache License 2.0	Java で利用可能な Web アプリのフレームワーク
PostgresSQL	14.8	PostgreSQL License	各種配信するデータを格納するデータベース
PostGIS	3.3	GNU General Public License	PostgreSQL で位置情報を扱うことを可能とする extention
pgRouting	3.5.0	GNU General Public License version 2	PostgreSQL でルート検索を可能とする extention
	3DTile 等配信データ		データベース以外で配信する 3D データ等

3) 歩行空間ネットワークの説明

①歩行空間ネットワークデータとは

- 国土交通省では、バリアフリー・ナビプロジェクトとしてICTを活用した歩行者移動支援サービスの普及を促進しており、同サービスに不可欠なバリアフリー情報(歩行空間ネットワークデータ)のオープンデータ化を推進しています。
- 歩行空間ネットワークデータは、段差や幅員などのバリアフリーに関連する情報を付与した「リンク」及びリンクの結節点を表す「ノード」で構成されています。
- 歩行空間ネットワークのイメージは、以下のとおりです。



- 参考

- https://www.mlit.go.jp/sogoseisaku/soukou/seisakutokatsu_soukou_tk_00026.html

②歩行空間ネットワークの利活用イメージ

- 歩行空間ネットワークデータを整備することにより、本データを活用したバリアフリーマップの作成、さらには、バリアフリールートのナビゲーションなどICTを活用した歩行者移動支援サービスを提供し、高齢者、障害者等の利便性向上を実現できます。
- 歩行空間ネットワークデータを活用したバリアフリールートの案内例は、以下の通りです。

階段、歩道の幅員や段差等のバリア情報を含む歩行空間ネットワークデータを用いることにより、バリアフリールートを探索することが可能となる。



● 参考

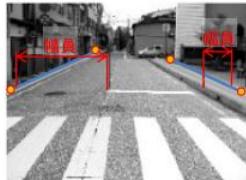
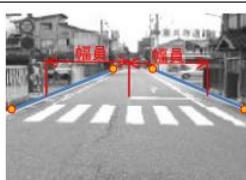
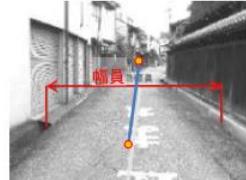
- https://www.mlit.go.jp/sogoseisaku/soukou/seisakutokatsu_soukou_tk_00026.html

③歩行空間ネットワークのデータ構成

- 歩行空間ネットワークデータについては、国土交通省より歩行空間ネットワークデータ等整備仕様が公開されています。
- 取得する属性の種類や、ノード、リンクの配置方法について定義されています。
- 歩行空間ネットワークデータのリンクの情報項目と属性情報の例を、以下に示します。

No	情報項目	フィールド名	形式	属性情報	第1層 (必須)	第2層 (任意)
1	リンク ID	link_id	文字列	リンクの ID	●	
2	起点ノード ID	start_id	文字列	起点のノード ID	●	
3	終点ノード ID	end_id	文字列	終点のノード ID	●	
4	リンク延長	distance	数値	リンクの延長を小数第1位で記入（単位はm）（経路の種類がエレベーターの場合には記入不要）	●	
5	経路の構造	rt_struct	コード	1：車道と歩道の物理的な分離あり、2：車道と歩道の物理的な分離なし、3：横断歩道、4：横断歩道の路面標示の無い道路の横断部、5：地下通路、6：歩道橋、7：施設内通路、8：その他の経路の構造、99：不明	●	
6	経路の種別	route_type	コード	1：対応する属性情報なし、2：動く歩道、3：踏切、4：エレベーター、5：エスカレーター、6：階段、7：スロープ、99：不明	●	
7	方向性	direction	コード	1：両方向、2：起点より終点方向、3：終点より起点方向、99：不明	●	
8	幅員	width	コード	1：1.0m未満、2：1.0m以上～2.0m未満、3：2.0m以上～3.0m未満、4：3.0m以上、99：不明	●	
9	縦断勾配	vtcl_slope	コード	1：5%以下、2：5%より大きい（起点より終点が高い）、3：5%より大きい（起点より終点が低い）、99：不明	●	
10	段差	lev_diff	コード	1：2cm以下、2：2cmより大きい、99：不明	●	
11	歩行者用信号機の有無	tfc_signal	コード	1：歩行者用信号機なし、2：歩車分離式信号機あり、3：押しボタン式信号機あり、4：これら以外の信号機、99：不明	●	
12	歩行者用信号機の種別	tfc_s_type	コード	1：音響設備なし、2：音響設備あり（音響用押しボタンなし）、3：音響設備あり（音響用押しボタンあり）、99：不明	●	
13	視覚障害者誘導用ブロック等の有無	brail_tile	コード	1：視覚障害者誘導用ブロック等なし、2：視覚障害者誘導用ブロック等あり、99：不明	●	
14	エレベーターの種別	elevator	コード	1：エレベーターなし、2：エレベーターあり（パリアフリー対応なし）、3：エレベーターあり（車いす使用者対応）、4：エレベーターあり（視覚障害者対応）、5：エレベーターあり（車いす使用者、視覚障害者対応）、99：不明	●	
15	屋根の有無	roof	コード	1：なし、2：あり、99：不明	●	

- 歩行空間ネットワークデータの歩道分類別のリンクの配置方法の事例については、以下のとおりです。

番号	分類			リンクの配置方法			
	歩道	中央線	概略図	リンク数	区分	配置位置	幅員
1	両側	有	—	2	歩道	歩道部	歩道幅員
2	両側	無	—	2	歩道	歩道部	歩道幅員
3	片側	有		1	歩道	歩道部	歩道幅員
				1	歩車共存	側端部	中央線～側端部
4	片側	無	—	1	歩道	歩道部	歩道幅員
5	無	有		2	歩車共存	側端部	中央線～側端部
6	無	無		1	歩車共存	道路中央付近	左端部～右端部

● 参考

➤ <https://www.mlit.go.jp/common/001244374.pdf>

④歩行空間ネットワークのオープンデータ

- 整備された歩行空間ネットワークデータについては、歩行者移動支援サービスに関するデータサイトにて公開されており、誰でもダウンロードできます。

データ名	組織	登録日	最終更新日	ダウンロード
歩行空間ネットワークデータ（神奈川県川崎市溝口駅周辺地区）	その他	2022/05/10	2022/05/10	<input type="checkbox"/> CSV <input type="checkbox"/> DBF <input type="checkbox"/> SHX <input type="checkbox"/> SHP <input type="checkbox"/> PRJ <input type="checkbox"/> PDF
歩行空間ネットワークデータ（福井県鯖江市）（2018年3月版適用）	その他	2021/03/31	2021/05/26	<input type="checkbox"/> CSV <input type="checkbox"/> DBF <input type="checkbox"/> PRJ <input type="checkbox"/> SHP <input type="checkbox"/> SHX <input type="checkbox"/> PDF
歩行空間ネットワークデータ（香川県高松市）（2018年3月版適用）	その他	2021/03/31	2021/05/26	<input type="checkbox"/> CSV <input type="checkbox"/> DBF <input type="checkbox"/> SHX <input type="checkbox"/> SHP <input type="checkbox"/> PRJ <input type="checkbox"/> PDF
歩行空間ネットワークデータ（東京都港区・お台場海浜公園周辺）（2018年3月版適用）	国土交通省	2021/03/31	2021/05/26	<input type="checkbox"/> CSV <input type="checkbox"/> DBF

● 参考

➤ <https://www.hokouukan.go.jp/top.html>

⑤歩行空間ネットワークの整備ツール

- 国土交通省では、歩行者のナビゲーションのために必要な歩行空間における段差や幅員、勾配等のバリアフリーに関する情報を入力し、データ化することが可能となる Web ツールを公開しています。
- 地図（地理院地図 2500 分の 1）上に、線データ「リンク」と点「ノード」を直接描画できます。視覚的に入力内容を確認しながら、データを作成することが可能となっています。
- 利用申請を行うことで、誰でもツールを利用できます。

- Web ツールの、利用、表示のイメージは以下のとおりです。



- 参考

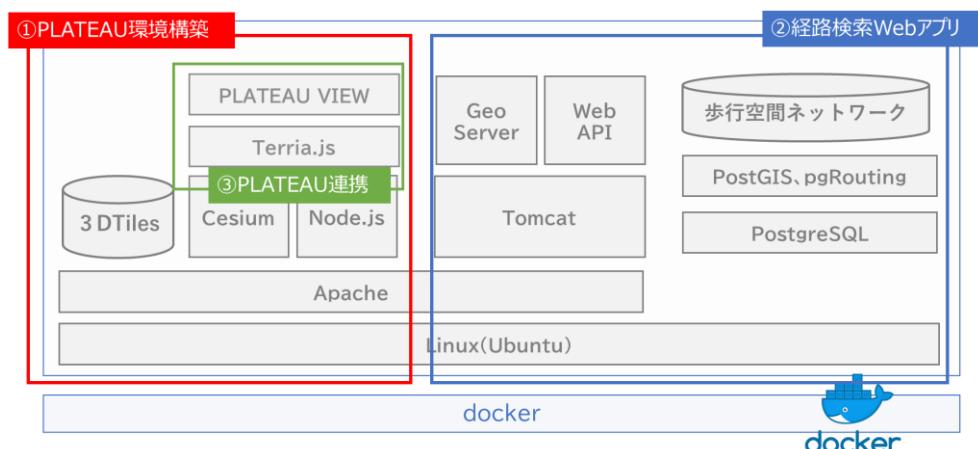
- https://www.mlit.go.jp/sogoseisaku/soukou/sogoseisaku_soukou_mn_00008.html

(2) PLATEAU VIEW 1.1 の基本セットアップ[°]

1) はじめに

- 今回は PLATEAU VIEW 1.1 と経路探索 Web アプリ環境をオールインワンで構築可能な Docker を用意しています。
 - サーバ上に構築する場合は、[稼動環境構築手順書 \(project-plateau.github.io\)](http://project-plateau.github.io)を参考にして構築を進めてください。
- 環境は一通り Docker で用意しておりますが、カスタムアプリとデータは別途ご用意いただく必要があります。

Dockerに含まれているもの	別途用意が必要なもの
<ul style="list-style-type: none"> Ubuntu Node.js Plateau View1.1(Cesium+TerriaJS) Apache Tomcat GeoServer PostgreSQL+PostGIS,pgRouting 	<ul style="list-style-type: none"> WebAPI 歩行空間ネットワークデータ



- 必要なファイル一式は以下のリポジトリから取得してください。
 - <https://github.com/Project-PLATEAU/PLATEAU-Terria-web-app.git>
- ファイル一式の構成は下記の通りです。

ファイルパス	内容	使用箇所
/Settings/docker	Docker環境構築ファイル一式	(2)-2)
/Settings/walk_space_network	歩行空間ネットワークデーター式 (リンク・ノード)	(3)-2)
/SRC/routesearchapi	経路探索Web API	(3)-3)
/SRC/TerriaMap	Plateau View 1.1 (経路探索カスタマイズ版)	(4)-1)
/SRC/terrajs	Plateau View 1.1 (経路探索カスタマイズ版)	(4)-1)

2) Docker 環境の準備

- WSL2 を有効化し、Docker Desktop をインストールしてください。
 - https://www.kagoya.jp/howto/cloud/container/wsl2_docker/

- /Settings/docker 以下のファイル一式を作業ディレクトリにコピーします。
- geoserver (2.20.4) の war ファイルを以下より入手します。
 - <https://sourceforge.net/projects/geoserver/files/GeoServer/2.20.4/geoserver-2.20.4-war.zip/download>
- zip ファイルがダウンロードされます。展開すると中に「geoserver.war」が入っているので、このファイルを作業ディレクトリ直下に¥webapps を作成し、その配下に格納します。
- Dockerfile 等が格納されたディレクトリをコマンドプロンプトで開いてください。

```
C:\$work\$handson_work\$PLATEAU-Terrria-web-app\$Settings\$docker>cd C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker>dir /B /s
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker\$compose.yaml
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker\$Dockerfile
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker\$mod_proxy_settings.txt
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker\$README.md
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker\$startup.sh
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker\$webapps
C:\$work\$handson_work\$2\$PLATEAU-Terrria-web-app\$Settings\$docker\$webapps\$geoserver.war
```

- 以下のファイルがダウンロードされていることを確認してください。
- Docker Desktop を起動し、コマンドプロンプトから以下コマンドを実行しイメージをダウンロードしてください。


```
docker pull ubuntu:22.04
```

 - 次に、以下コマンドを実行してください。


```
docker build -t badhbh:1.5 .
```

```
docker compose up
```

 - http://localhost:8080/geoserver/にアクセスできたら完了です。
 - イメージのビルドに約 15 分、サーバ立ち上げに 10 分ほど要します。
 - ※ホスト PC の容量が圧迫された場合


```
C:\$Users\$Username\$AppData\$Local\$Temp
```

 内の一時データを削除することで問題を解消できる可能性があります。

3) PLATEAU VIEW 1.1 の動作確認

- ブラウザから以下にアクセスし、PLATEAU VIEW 1.1 が起動することを確認してください。
 - http://localhost:8080/plateau/
- Add Data からデータを追加・表示できることを確認してください。
- 補足：Dockerfile では PLATEAU VIEW 1.1 の起動に必要な以下のコマンドも起動時に実行しているため、特に追加の操作をせず PLATEAU VIEW 1.1 を表示できます。


```
export NODE_OPTIONS=--max_old_space_size=4096
yarn
yarn gulp
yarn start
```

- 参考
 - [GitHub - Project-PLATEAU/PLATEAU-VIEW-1.1: Catalog-based web](https://github.com/Project-PLATEAU/PLATEAU-VIEW-1.1)

(3) 経路探索 WebAPI 構築

1) 経路探索 WebAPI 動作に必要なソフトウェアインストール

- 稼働環境に必要なミドルウェアとソフトウェアをインストールします。
 - OpenJDK
 - Tomcat9
 - GeoServer
 - PostgreSQL14 と PostGIS3

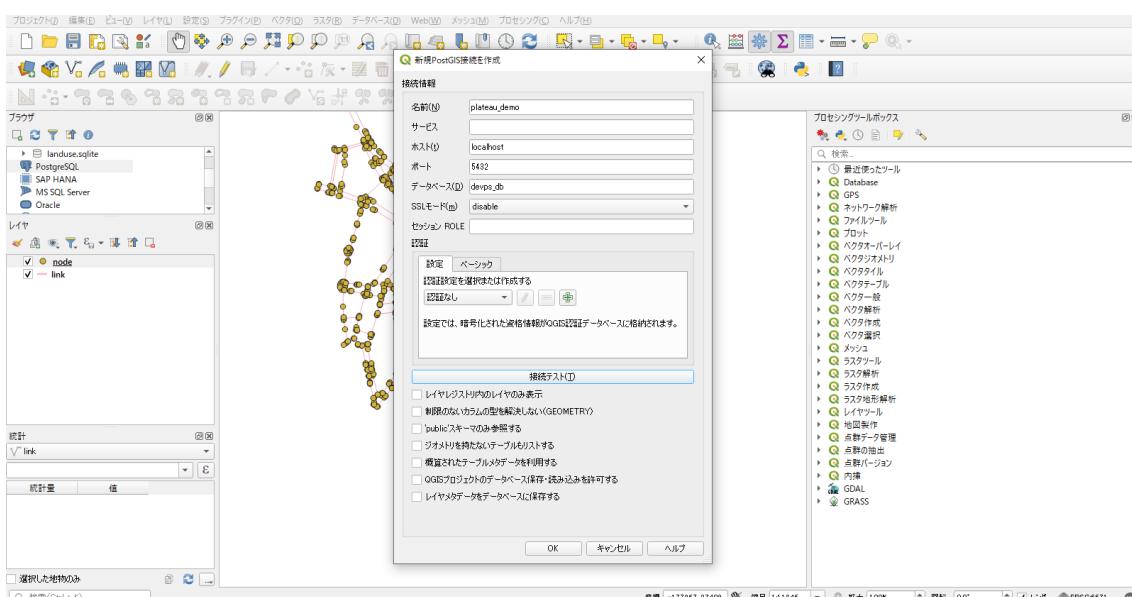
※ハンズオンでは、これらはすべて提供する Docker に含まれているため不要です。サーバに構築する場合、稼働環境構築手順書 (project-plateau.github.io)の手順を参考に構築を実施してください。
- 開発用ツールをローカルの WindowsPC にインストールします。
 - IDE (STS)
 - ✧ WebAPI (SpringBoot) の開発・ビルドに使用します。
 - ✧ 参考 Spring Tool Suite 4(STS)をインストールする手順 | ITSakura
 - QGIS
 - ✧ 歩行空間ネットワークデータの取込と、スタイルの作成に使用します。
 - ✧ 参考 index-10.pdf (maff.go.jp)
 - pgAdmin
 - ✧ データベースの閲覧・操作に使用します。
 - ✧ 参考 【PostgreSQL】pgAdmin4 のインストール手順 | 秋拓技術学院 (syutaku.blog)
 - Anaconda
 - ✧ 歩行空間ネットワークデータからコスト値を算出しセットするのに使用します。
 - ✧ 参考 <https://www.python.jp/install/anaconda/windows/install.html>

2) 歩行空間ネットワークデータ投入

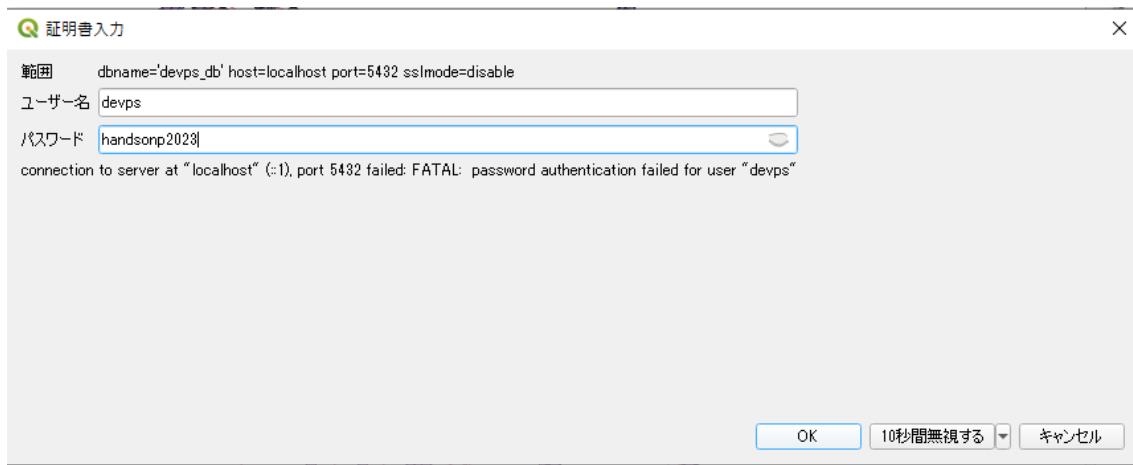
- QGIS からレイヤ情報の取り込みと定義するカラムを追加し、Anaconda からコストの計算を行います。
- コスト算出のロジック詳細については、3D 都市モデルを活用した民間サービス開発に向けたコンセプト策定及び案件組成業務 (mlit.go.jp)の「III-5 経路探索機能 | コスト算出」(p44-50) と同様ですので、こちらをご参照ください。
- QGIS の操作
 - plateau_demo コンテナと PostgreSQL が起動した状態で、QGIS からデータベースに接続します。

※コンテナ起動と同時に PostgreSQL は起動されます。

- 新規プロジェクトを立ち上げ、/Settings/walk_space_network/link.shp と/Settings/walk_space_network/node.shp をドラッグ&ドロップし画面に追加します。
- 「ブラウザ」ウィンドウより「PostgreSQL」を右クリックし、新規接続を押下します。データベースへの接続情報を入力し、接続テストを押下します。
 - ✧ 名前：任意
 - ✧ ホスト：localhost
 - ✧ ポート番号：5432
 - ✧ データベース名：devps_db



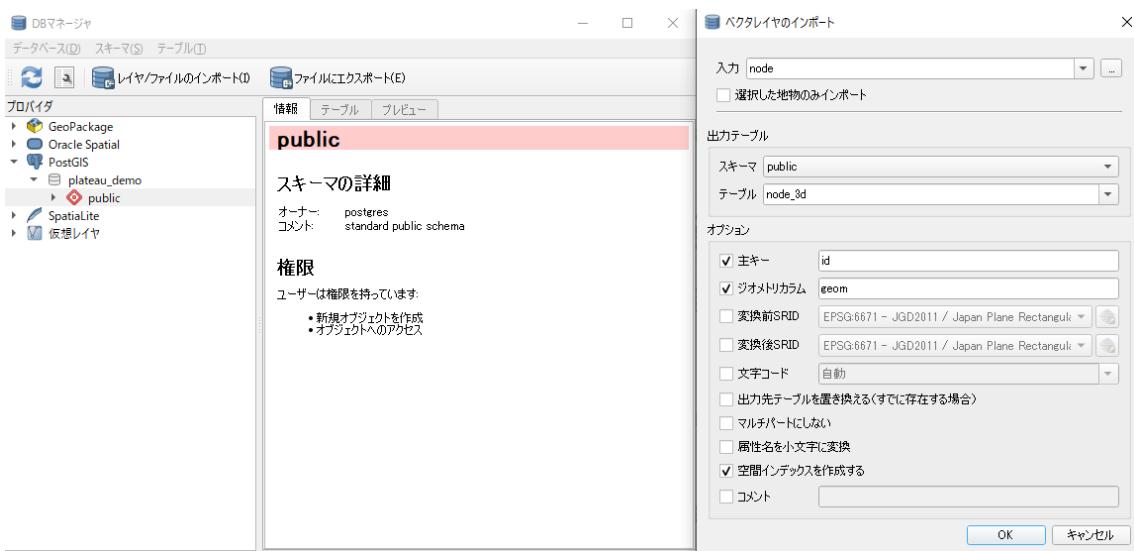
- ダイアログが開き、ユーザー名とパスワードを求められるため、以下のように入力します。
 - ✧ ユーザー名：devps
 - ✧ パスワード：handsonp2023



※ビルトの際、docker-compose.yaml ファイルによって PostgreSQL にはポート番号 5432 から接続できるように設定されています。

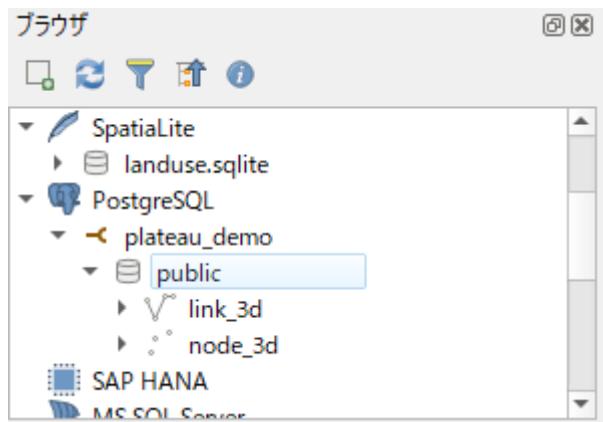
※データベース名とユーザー名、パスワードは Docker イメージ作成と同時に作成・設定されています。

- データベース-DB マネージャよりレイヤのインポートを行います。プロバイダより PostGIS-[設定した名前]-public を選択し、「レイヤ/ファイル のインポート」を押下し、以下のように入力します。
 - ✧ 入力：「ノードレイヤの名称」
 - ✧ 出力テーブル-テーブル：node_3d
 - ✧ 主キー：id
 - ✧ ジオメトリカラム：geom
- また、オプションより「空間インデックスを作成する」を有効にしてください。

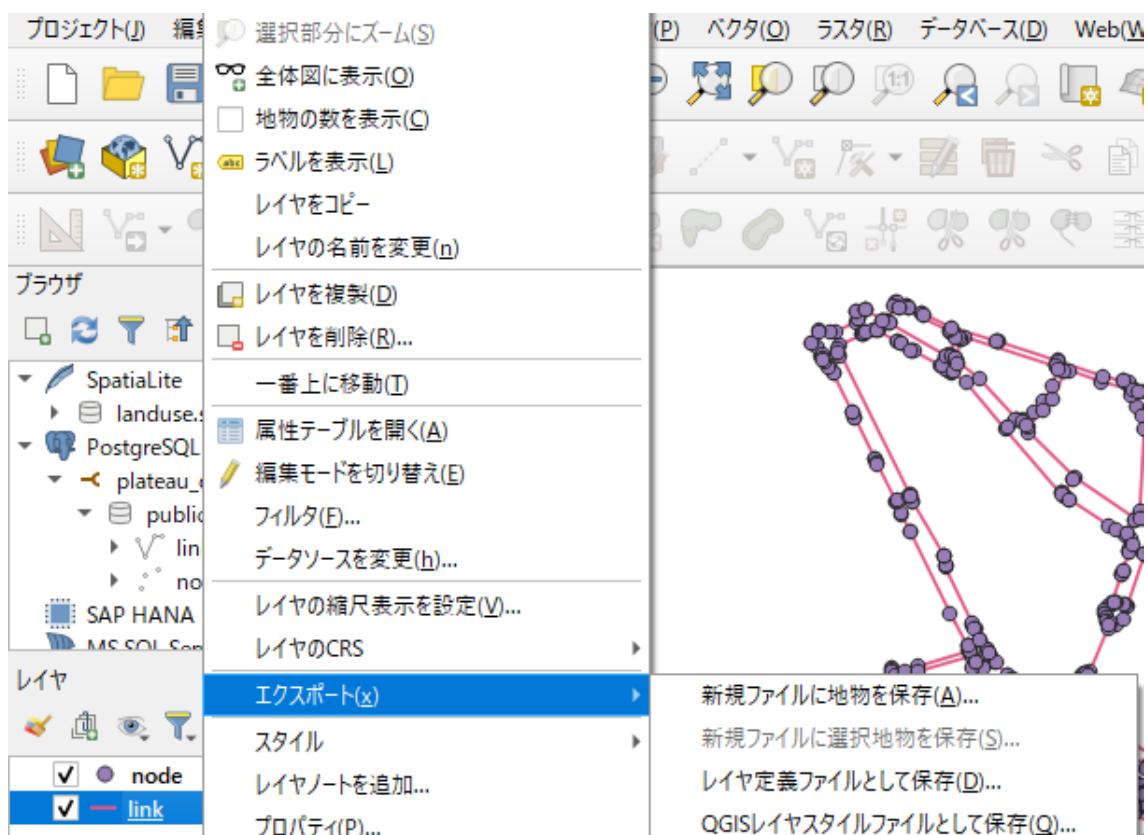


※今回のデモに使用しているデータは広島市の地図データです。SRID が「EPSG:6671」の「JGD2011 / Japan Plane Rectangular CS III」を座標参考系に選択しているか確認してください。

- 同様に入力レイヤを「リンクレイヤの名称」、出力テーブル-テーブルを link_3d にしてインポートします。
- 図のようになれば完了です。



- 続いてレイヤスタイルの登録を行います。スタイルを設定したらレイヤ名を右クリックし、「エクスポート-QGIS レイヤスタイルファイルとして保存」を選択します。

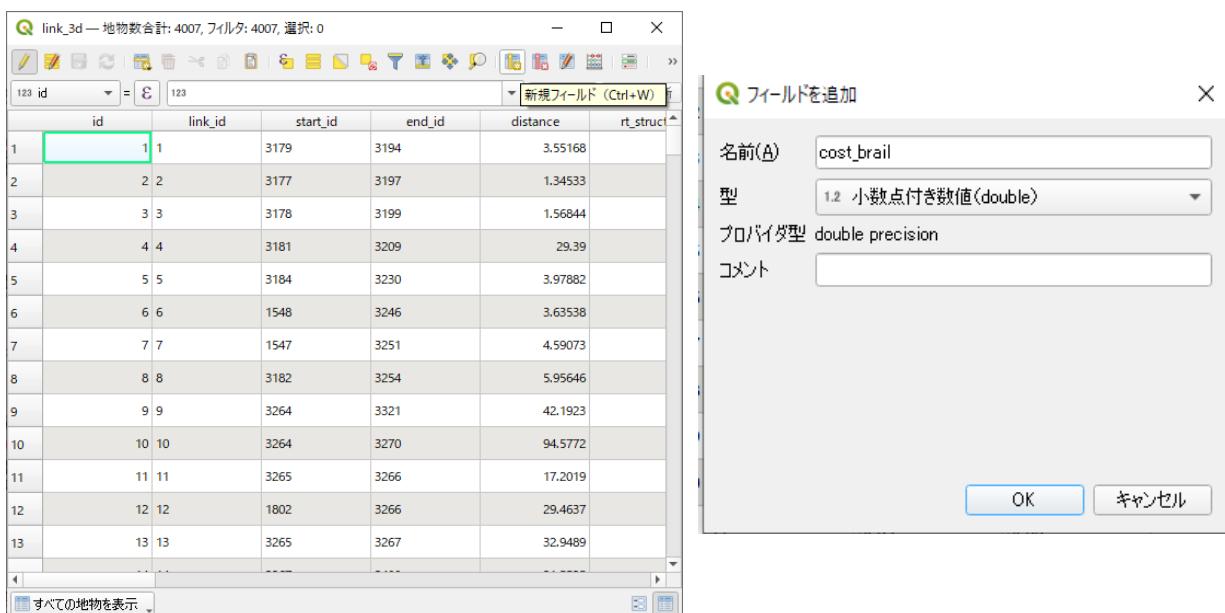


- 「スタイルを保存」プルダウンから「SLD スタイルファイル」を選択し、任意のディレクトリに保存してください。ファイル名は linkstyle.sld とします。

- 定義するカラムを追加します。内容は以下の通りです。

カラム名	型	備考
cost_wheelchair	double precision	車いす利用者向け経路探索用コスト
cost_elderly	double precision	高齢者・乳幼児向け経路探索用コスト
cost_brail	double precision	視覚障がい者向け経路探索用コスト

- 「ブラウザ」ウィンドウより link_3d をダブルクリックし、レイヤに追加してください。
- [F6] キーを押して属性テーブルを開き、編集を有効にして新規フィールドを押下し、上表の内容をダイアログに入力します。



- 編集内容を保存しテーブルを閉じます。

● Anaconda の操作

- /Settings/walk_space_network/calculate_and_update_cost.py を編集します。
- エディタでファイルを開き、228 行目からの os.environ で設定された行を以下のように編集し、保存します。

```

228     os.environ['RDS_HOST'] = 'localhost'
229     os.environ['RDS_PORT'] = '5432'
230     os.environ['RDS_DBNAME'] = 'devps_db'
231     os.environ['RDS_USER'] = 'devps'
232     os.environ['RDS_PASSWORD'] = 'handsomp2023'
233     # データ取得クエリ

```

- Anaconda Prompt を起動します。
- 以下コマンドを入力し、仮想環境を作成します。

```
conda create -n calculate-env python=3.9
```
- 以下コマンドを入力し、仮想環境を有効化します。

```
conda activate calculate-env
```
- 操作中の環境が以下のように(base)から(calculate-env)に切り替わります。

- PostgreSQL を操作するライブラリである psycopg2 をインストールします。

```
pip install psycopg2
```
- 作業ディレクトリに移動し、python ファイルを実行します。

```
python calculate_and_update_cost.py
```

実行が開始されると、コスト算出結果がコンソールに出力され続けます。
- QGIS を開き、link_3d テーブルの内容を確認します。新規に定義したフィールドに要素が代入されていたら、設定の完了です。

3) 歩行空間ネットワークデータ GIS 配信設定

- 作成したレイヤに GeoServer を通して PLATEAU VIEW 1.1 からアクセスできるようにします。
 - ユーザー名:admin パスワード:geoserver でログインします。
- ワークスペースとストアを登録する
 - 「ワークスペース新規作成」より、ワークスペースを作成し保存します。
 - Name と URI は任意ですが、今回は plateau-hands-on として登録します。



- 「ストア」より「ストア新規追加」を押下します。
- 新規データソースより PostGIS を選択します。
- 以下のように入力します。
 - ✧ ワークスペース : plateau-hands-on

- ❖ データソース名 : plateau-hands-on
- ❖ Host : localhost
- ❖ Port : 5432
- ❖ Database : devps_db
- ❖ Schema : public
- ❖ User : devps
- ❖ Passwd : handsonp2023

ストア基本情報

ワークスペース *

plateau-hands-on

データソース名 *

plateau-hands-on

解説

有効化

Auto disable on connection failure

パラメーターの接続

host *

localhost

port *

5432

database

devps_db

schema

public

user *

devps

passwd

ネームスペース *

plateau-hands-on

Expose primary keys

- 保存します。その際 node_3d と link_3d がテーブル一覧に表示されたら完了です。
- スタイルを登録する

- 「スタイル」より新規スタイルを登録します。以下のように入力し、Upload a style file [ファイルの選択]より、QGIS で作成したスタイルファイルを選択します。今回はユーザ名を linkstyle とします

新規スタイル

スタイルを作成するには、新しいSLD定義を作成、あるいは既存の定義をテンプレートとして作成、あるいはあなたのファイルシステム用の既存ファイルをアップロードしてください。編集画面では記法がハイライト表示され、フルスクリーンとなります。SLDドキュメントの記法に誤りがないかを確認するには“検証”ボタンをクリックしてください。

Data

Style Data

ユーザ名
linkstyle

ワークスペース
plateau-hands-on

データ形式
S...

Legend

Legend
Add legend

Preview legend

Style Content

Generate a default style
選んでください ▾ Generate ...

Copy from existing style
選んでください ▾ コピーしています

Upload a style file
ファイルの選択 linkstyle.sld
Upload ...

- Upload … を押下しファイルを読み込んだら、検証し保存します。

● レイヤを作成する

- 「レイヤ」よりリソースを新規に追加します。
- 「レイヤ追加元」プルダウンより plateau-hands-on:plateau-hands-on を選択してください。
- アクション列で「公開」または「再公開」リンクを選択し、レイヤを操作します。以下の操作を link_3d に対し行います。
- 下スクロールし、「範囲矩形」より
 - ✧ ネイティブの範囲矩形：「データを元に算出」をクリック
 - ✧ 緯度経度範囲矩形：「ネイティブの範囲を元に算出」をクリック

数字が表示されたら保存してください。

範囲矩形

ネイティブの範囲矩形

最小X	最小Y	最大X	最大Y
27,740.01953125	-179,016.953125	29,521.328125	-177,028.765625

データを元に算出

Compute from SRS bounds

緯度経度範囲矩形

最小X	最小Y	最大X	最大Y
132.4683332391434	34.385840151034756	132.48777276495088	34.40381395741041

ネイティブの範囲を元に算出

- link_3d では、公開タブからスタイルを変更します。WMS 設定の Layer Settings の「デフォルトスタイル」プルダウンより、登録したスタイルを選択し、保存します。

WMS設定

Layer Settings

- クエリ処理可
 Opaque

デフォルトスタイル

plateau-hands-on:links...
plateau-hands-on:linkstyle
ne:countries_mapcolor9
ne:disputed
ne:populated_places
poi
point
poly_landmarks
polygon
pophatch
population
rain
raster
restricted
simple_roads
simple_streams



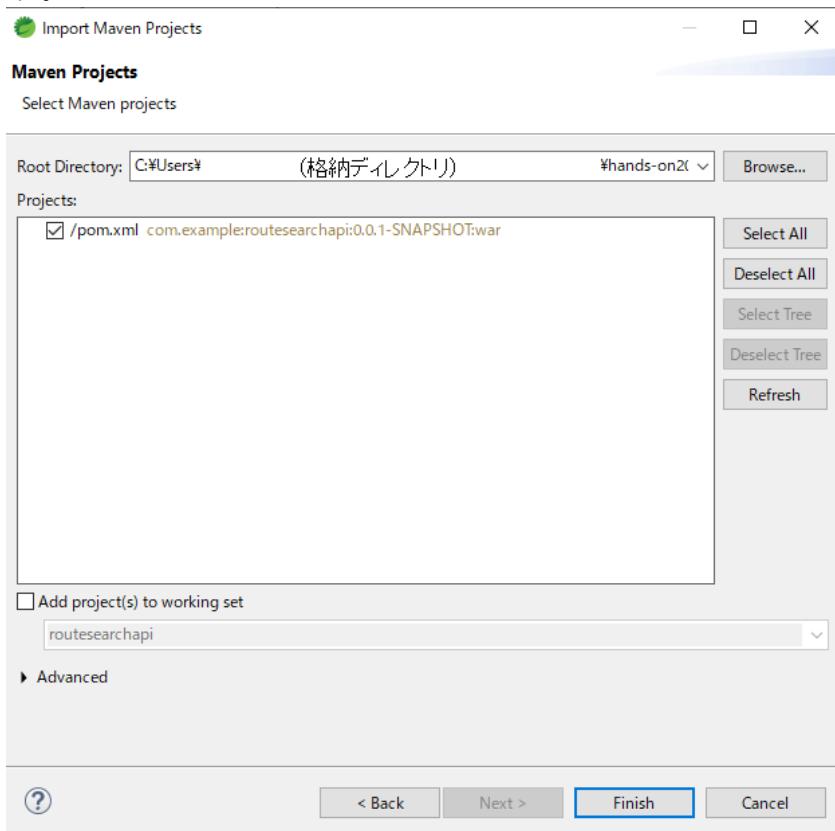
- レイヤープレビューにて link_3d の”OpenLayers”リンクをクリックし、データが表示されたら完了です。



- コンテナを停止します。
- 作業ディレクトリの webapps/geoserver.war を取り除きます。取り除かなかった場合、デプロイされた geoserver フォルダと追加したレイヤがコンテナ起動時に削除され、war ファイルとともに再デプロイされてしまう（追加した設定が上書きされ戻されてしまう）ので、注意してください。

4) 歩行空間 WebAPI アプリソース・ビルド方法

- /SRC/routesearchapi 以下のファイル一式を作業ディレクトリにコピーします。
- Spring Tool Suite 4 を起動し、hands-on2023 ディレクトリをローンチします。
- プロジェクトをインポートします。Import projects .. > Maven > Existing Maven Projects を選択してください。/pom.xml という routesearchapi の SNAPSHOT.war に関連のあるプロジェクトを指定して Finish を押下してください。



- Maven を更新します。プロジェクト右クリック> Maven > Update Project を選択し、OK を押下してください。
- src/main/resources に存在する application.properties を、環境に合わせて編集します。※Docker の設定をそのまま使用される場合は変更不要です。

```
application.properties X  
1 #DATABASE  
2 spring.jpa.database=POSTGRESQL  
3 spring.datasource.url=jdbc:postgresql://localhost:5432/devps_db  
4 spring.datasource.username=devps  
5 spring.datasource.password=hansongp2023  
6 spring.jpa.show-sql=true  
7  
8 #LOG  
9 logging.file.name=/var/lib/tomcat9/logs/routesearchapi.log  
10 logging.level.org.springframework.web=INFO  
11 logging.level.view3d=DEBUG  
12  
13 #PORT  
14 server.port=8082  
15 # ROUTE SEARCH EPSG  
16 route.epsg.view=4326  
17 route.epsg.data=6671  
18 #etc  
19 spring.mvc.pathmatch.matching-strategy = ANT_PATH_MATCHER
```

```

#DATABASE ⇒データベースの接続設定
spring.jpa.database=POSTGRESQL
spring.datasource.url=jdbc:postgresql://localhost:5432/devps_db
spring.datasource.username=devps
spring.datasource.password=handsopn2023
spring.jpa.show-sql=true

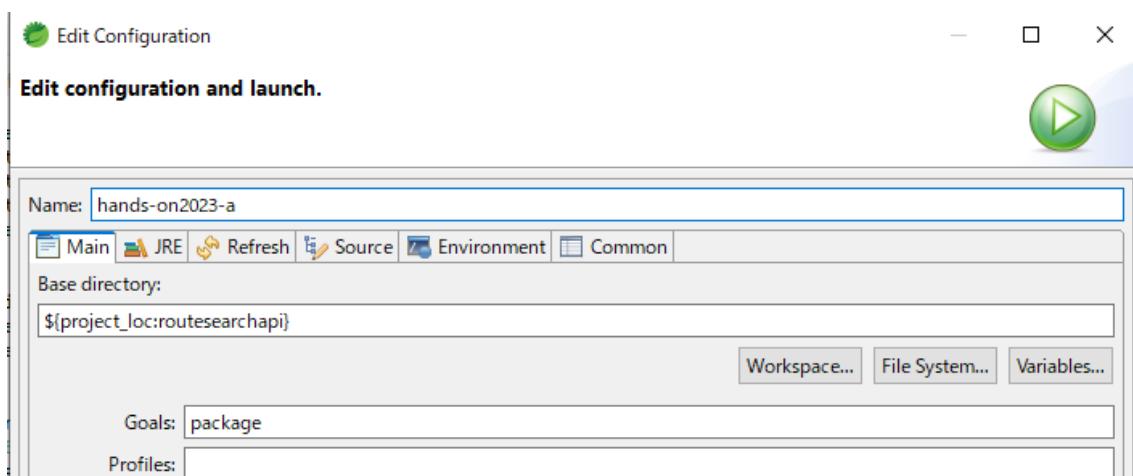
#LOG ⇒ログの出力先とログレベル
logging.file.name=/var/lib/tomcat9/logs/routesearchapi.log
logging.level.org.springframework.web=INFO
logging.level.view3d=DEBUG

#PORT ⇒ ポート番号※ 開発環境で実行する時のみ適用されます。warファイルには反映
されません。
server.port=8082

# ROUTE SEARCH EPSG ⇒ 表示時の座標系（view）と元データの座標系（data）
# ※元データの座標系は平面直角座標系を使用してください。
route.epsg.view=4326
route.epsg.data=6671
#etc
spring.mvc.pathmatch.matching-strategy = ANT_PATH_MATCHER

```

- war ファイルを作成します。プロジェクトを右クリックし、Run As > Maven build を選択してください。
- Goals に package を入力し、実行してください。



- コンソールに **BUILD SUCCESS** と表示され、target ディレクトリに routesearchapi-0.0.1-SNAPSHOT.war が作成されていれば完成です。
- 上記操作で作成した routesearchapi-0.0.1-SNAPSHOT.war を、webapps ディレクトリに移動させます。
- 移動後、routesearchapi-0.0.1-SNAPSHOT.war を routesearchapi.war に名前変更します。
※plateau-demo コンテナは webapps フォルダをマウントしているため、次回起動時に自動的にデプロイされます。

5) 歩行空間 WebAPI 内部ロジックの説明

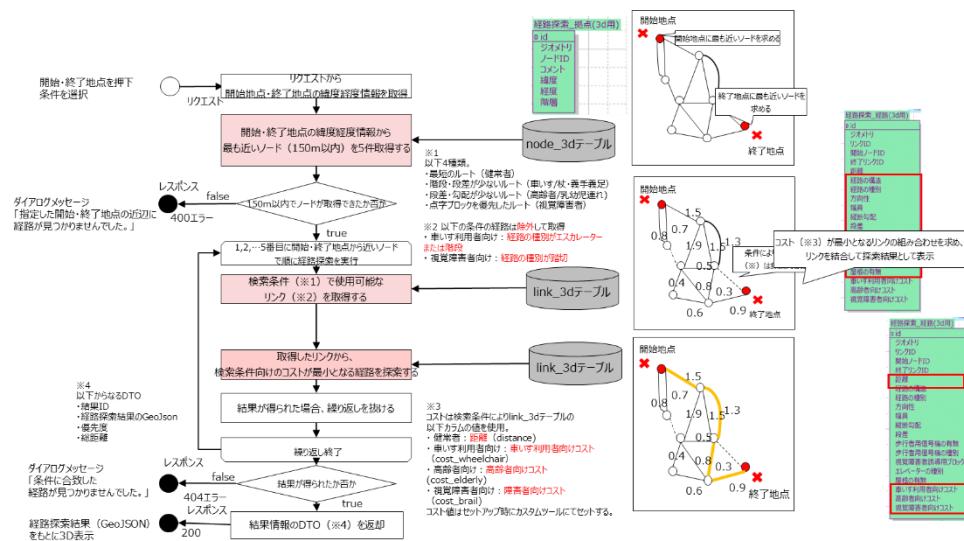
- 今回構築した WebAPI は Java のフレームワークである Spring Boot で実装されています。
 - <https://spring.io/projects/spring-boot/>
- 標準的な Web アプリケーションの 3 層構造に則り、外部からの入力を受け付ける Controller, 業務ロジックを実装する Service, データベースにアクセスする Dao の 3 階層に分けてアプリケーションを構成しています。以下では、3 層構造に基づいて、サンプルソースの内部ロジックを説明します。
 - 参考
 - <https://penguinlabo.hatenablog.com/entry/spring/web-layered>
- Controller クラス
 - /routesearchapi/src/main/java/view3d/controller/RouteSearchApiController.java
 - API の処理全体を routeSearch メソッドで実装しています。
 - API のパスやメソッドを @RequestMapping、レスポンス @ApiResponses アノテーションで定義しています。
 - メソッドの引数に @RequestParam アノテーションで API のリクエスト パラメータを定義しています。今回は以下の 3 種類をクエリパラメータで定義しています。
 - ❖ start: 経路探索開始地点緯度経度（カンマ区切り文字列）
 - ❖ end: 経路探索終了地点緯度経度（カンマ区切り文字列）
 - ❖ condition: 検索条件タイプ (1:健常者向け, 2:車いす利用者向け, 3:高齢者・乳幼児連れ向け, 4: 視覚障害者向け)

```
@RequestMapping(value = "/search", method = RequestMethod.GET)
@ApiOperation(value = "条件付き経路探索を実施します", notes = "条件付き経路探索を実施")
@RequestBody
@ApiResponses(value = {
    @ApiResponse(code = 400, message = "リクエストパラメータが不正の場合.または開始・終了ノードがリクエスト地点から一定範囲内に存在しない場合.", response = ResponseError.class),
    @ApiResponse(code = 404, message = "経路が見つからなかった場合", response = ResponseError.class),
    @ApiResponse(code = 500, message = "処理時にエラーが発生した場合", response = ResponseError.class)
})
public RouteSearchResultForm routeSearch(@RequestParam("start") String start, @RequestParam("end") String end, @RequestParam("condition") Integer condition) {
```

- Controller ではリクエストデータのチェックを実装しています (42-62 行目)。パラメータのフォーマット不正や、緯度経度の範囲不正はリクエスト不正としてレスポンスを返すように実装します。
- 64 行目で Service の処理を呼び出します。処理結果を受取り、レスポンスとして返します。

● Service クラス

- /routesearchapi/src/main/java/view3d/service/RouteSearchService.java
- 経路探索の内部処理を routeSearch サービスで実装しています。
- 内部処理の全体は下図の通りです。(ピンク色の項目の DB アクセス処理は Dao クラスで実施)



- 63-67 行目で NodeDao.getNearNode メソッドから開始地点、終了地点から近い順にノードを 5 件ずつ取得します。
- ノードが開始終了地点から 150m 以内で得られなかった場合、取得不正として結果を返します。 (105-108 行目)
- ノードが得られた場合、69-100 行目で近い順に開始ノードから終了ノードまでの経路探索を実施します。経路探索処理は 87 行目で RouteSearchDao.searchRouteWithCost メソッドを呼び出して実行します。経路探索結果が得られた場合、結果を返します。
- 結果が得られなかった場合、2 番目、3 番目、…に近いノードを順に使用して経路探索を実施し、結果を得次第結果を返します。結果が得られない場合、エラーを返します。

● Dao クラス (NodeDao)

- /routesearchapi/src/main/java/view3d/dao/NodeDao.java
- getNearNode メソッドで最も近いノードを順に取得します。
- 引数として以下を受け付け、SQL 文に組み込んでいます。
 - ❖ wkt: 開始（終了）地点の座標 (Well-known text 形式)
 - ❖ limit: 取得件数
 - ❖ viewEPSG: 画面表示で用いる座標系
 - ❖ dataEPSG: データの座標系
- 元の SQL 文は以下の通りです (: (コロン) +引数名の箇所は引数の値で置換)。postGIS の ST_Distance 関数を用いて指定した座標からノードの距離を求め、距離が近い順の取得を実現しています。
- そのほか postGIS の関数として、ST_GeomFromText 関数を wkt からジオメ

トリ型への変換に、ST_Transform 関数を座標系の変換に使用しています。

```
SELECT
    CAST(node_id AS integer) AS node_id
    , ST_Distance(
        t1.geom
        , ST_Transform(ST_GeomFromText(:wkt, :viewEPSG), :dataEPSG)) AS distance
FROM
    node_3d AS t1
ORDER BY distance asc Limit :limit
```

- Dao クラス (RouteSearchDao)
 - /routesearchapi/src/main/java/view3d/dao/RouteSearchDao.java
 - searchRouteWithCost メソッドで経路探索を実施します。
 - 引数として以下を受け付け、SQL 文に組み込んでいます。
 - ✧ startNodeId: 経路探索を開始するノード ID
 - ✧ endNodeId: 経路探索を終了するノード ID
 - ✧ condition: 検索条件タイプ (1:健常者向け, 2:車いす利用者向け, 3:高齢者・乳幼児連れ向け, 4: 視覚障害者向け)
 - ✧ resSRID: レスポンス (画面表示で用いる) 座標系
 - 79-92 行目で、検索条件タイプに応じてエッジ (経路探索対象の経路一覧) を取得する SQL を組み立てています。
 - ✧ エッジを取得する SQL の基本形 (34 行目の変数 route_search_edgge の内容) は以下の通りです。

```
SELECT id AS seq, cast(link_id AS integer) AS id, cast(start_id AS integer) AS source,
cast(end_id AS integer) AS target, $edge_column AS cost
FROM link_3d WHERE link_id is not NULL
```

- 後述の pgr_dijkstra 関数で用いるため、以下のカラムを含めます。
 - ✧ seq: シーケンス
 - ✧ id: 一意の ID
 - ✧ source: リンクの開始地点 ID
 - ✧ target: リンクの終了地点 ID
 - ✧ cost: 経路算出に用いるコスト。コストの値は検索条件別に異なるカラムに格納しているので、動的に取得先のカラムを変えるようにします。
 - ✧ 車いすや視覚障害者向けの検索の場合、段差や踏切などの経路を絞るため条件を追加しています。(36、38 行目の変数 condition_wheelchair、condition_brail)
- 引数の条件と、組み立ててエッジ SQL を組み込んで経路探索 SQL を実行し

ます。

❖ 実行する SQL は以下の通りです。流れを追って説明します。

```
SELECT ROW_NUMBER() OVER(ORDER BY united.geom ASC) AS result_id, :priority AS priority,  
-- ③②のジオメトリを GeoJSON 形式に変換し、総距離を求める  
ST_AsGeoJSON(ST_Transform(united.geom, :srid)) AS geojson, ST_Length(united.geom) AS  
distance FROM  
(  
    -- ② ①で取得したジオメトリを 1 つのジオメトリにまとめる  
    SELECT  
        ST_LineMerge(ST_UNION(res.geom)) AS geom  
    FROM  
    (  
        --- ① pgr_dijkstra 関数で最短経路となるリンクの組み合わせを取得  
        SELECT  
            t1.seq  
            , t1  
            , edge  
            , t2.geom as geom  
        FROM  
        pgr_dijkstra(  
            :edge_sql  
            , :start_node_id  
            , :end_node_id  
            , directed $$:= false  
        ) AS t1  
        INNER JOIN link_3d AS t2  
        ON t1.edge = cast(t2.link_id AS integer)  
    ) as res  
) AS united;
```

❖ ① まず、pgr_dijkstra 関数で最短経路となるリンクの組み合わせを取得します。pgr_dijkstra 関数の引数にはエッジ取得 SQL と、開始ノード ID、終了ノード ID、データの有向性の有無を指定しています。また、pgr_dijkstra 関数の取得結果は最短となるリンク ID (edge) の一覧となるため、内部結合でリンクのジオメトリも取得するようにします。

❖ ② 次に、①で取得したジオメトリはリンクごとに分かれているので 1 つの

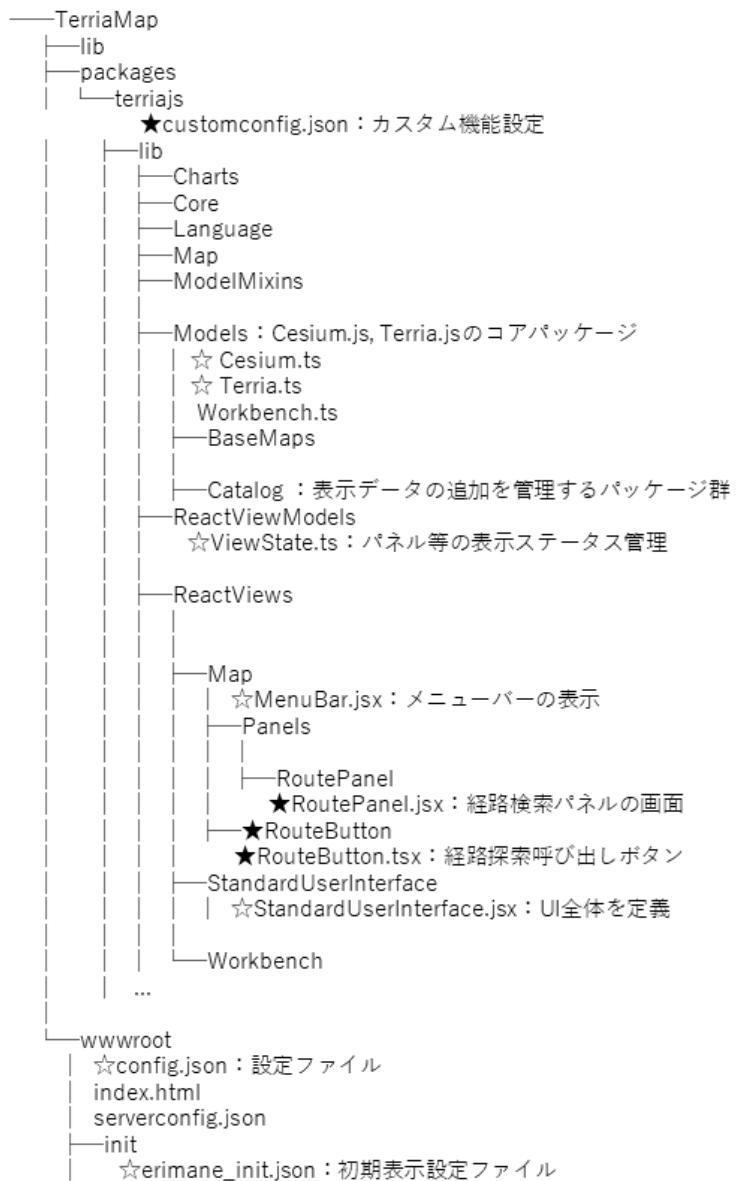
ジオメトリにまとめます。postGIS の ST_Union 関数と ST_LineMerge 関数を使用します。

- ❖ ③ 最後に、②で求めたジオメトリを GeoJSON 形式にまとめて、総距離を求めます。GeoJSON への変換は ST_AsGeoJSON 関数で行っており、距離の算出は ST_Length 関数で行っています。

(4) PLATEAU VIEW 1.1 連携方法の説明

1) 連携を行うためのカスタマイズについて

- PLATEAU VIEW 1.1 のソースコードをカスタマイズし、経路探索 WebAPI との連携を構築する方法を説明します。
- PLATEAU VIEW 1.1 のソースコードのパッケージ構成は下図の通りです。修正するファイルに☆を、追加するファイルに★をつけています。PLATEAU VIEW 1.1 は、JavaScript のフレームワークである React ベースになっているので、カスタマイズ箇所も React で実装します。
- カスタマイズしたソースコードは以下に格納しています。
 - TerriaMap 以下: /SRC/TerriaMap
 - TerriaMap¥packages¥terriajs 以下: /SRC/terriajs
 - ソースコードの配置・ビルド手順は PLATEAU VIEW 1.1 の README を参照してください。
 - ❖ <https://github.com/Project-PLATEAU/PLATEAU-VIEW-1.1#readme>



- 修正後版のフォルダから、上図の星印がついているファイルを Docker 上で公開した PLATEAU VIEW に反映し、ビルドしなおすことで修正が反映されます。

2) ソースコードの修正箇所と修正手順について

- 以下ではソースコードの修正箇所と修正手順を説明します。
- まず、経路探索の入力を受け付ける画面用のコンポーネントを作成します。
- 作成場所は下記のとおりです。Panels フォルダ内にヘルプパネルや共有パネルなどのパネルの jsx ファイルが格納されているので、同様に作成します。

- ❖ `TerriaMap¥packages¥terriajs¥lib¥ReactViews¥Map¥Panels¥RoutePanel¥RoutePanel.jsx`
- 以下、ソースコードの内容を示します。
- 36-51 行目の `constructor` ではコンポーネントが読み込まれた時の処理を定義します。親コンポーネントからの `props` の引継ぎと、自コンポーネント内の `state` の初期化をしています。
 - ❖ 参考
 - ❖ [React.Component – React \(reactjs.org\)](#)
 - ❖ [props と state のイメージをつかむ【はじめての React】 - Qiita](#)
- `viewState`、`terria`、`routeStart`、`routeEnd` は `props` から受け取っています。`viewState` は画面の表示状態を共通管理するオブジェクト、`terria` は PLATEAU VIEW 1.1 (TerriaMap ベース)の諸々の機能を司るオブジェクトです。`routeStart` と `routeEnd` は経路探索の開始地点・終了地点の座標を保持します。地図上でクリックした際に取得した座標を受け渡すために `props` を用います。
- 以下が `jsx` ファイル内で定義されている関数の一覧です。★をつけているものについて、順を追って説明します。

関数名	処理概要
<code>componentDidMount</code>	コンポーネント読み込み時処理
<code>★createCZMLTextAndWorkbenchAdd</code>	GeoJSON形式のデータからczml形式のデータを作成し、ワークベンチに追加 (=検索結果地物を表示) する。
<code>★render</code>	画面描画処理
<code>Clear</code>	クリアボタン押下時イベント。検索条件と検索結果をリセットする。
<code>★search</code>	検索ボタン押下時イベント。入力した条件でAPIリクエストし、結果を処理する。

- `render` では画面の描画処理を定義します。パネル全体の構成や、入力欄、検索・クリアボタンなどの要素、イベント処理の定義を行います。
- `search` では、検索ボタン押下時のイベントを定義します。緯度経度の値をチェックしたあとで、324 行目で WebAPIへのリクエストを `fetch` で行います。結果が正常の場合、`createCZMLTextAndWorkbenchAdd` 関数を呼び出します。結果が異常の場合、ステータスコードによってダイアログメッセージを出し分けます。
- `createCZMLTextAndWorkbenchAdd` では、経路探索結果として受け取った地物の表示処理を実装します。TerriaMap では、3D の GIS データを 3D 表示するためにはデータが CZML 形式である必要があります。WebAPI で受け取ったデータは GeoJSON 形式であるため、CZML 形式に変換します (67-149 行目)。ここでは座標情報のセットと表示のスタイル定義を行っています。
 - ❖ [CZML・GIS 実習オープン教材 \(gis-oer.github.io\)](#)

```

let czmlObj = [];

let positionsArrayArray = [];

const czmlTop = {
    id: "document",
    name: "line",
    version: "1.0"
};

czmlObj.push(czmlTop);

if (type == "MultiLineString") {
    // マルチラインの場合
    for (let i = 0; i < coordinates.length; i++) {
        let positionsArray = [];
        let coordArray = [];
        for (let j = 0; j < coordinates[i].length; j++) {
            coordArray.push(coordinates[i][j][0]);
            coordArray.push(coordinates[i][j][1]);
            positionsArray.push(Cartographic.fromDegrees(coordinates[i][j][0], coordinates[i][j][1]));
            //2D の JSON が来た場合高さ 0 でセットする。
            (coordinates[i][j].length == 3) ? coordArray.push(parseFloat(coordinates[i][j][2])) : coordArray.push(0);
        }
        let featureObj = {
            id: i + 1,
            name: "routeSearchResult",
            polyline: {
                "positions": {
                    "cartographicDegrees": coordArray,
                },
                "material": {
                    "polylineOutline": {
                        "color": {
                            "rgba": [255, 165, 0, 255],
                        },
                        "outlineColor": {

```

- 結果によって、ポリラインがマルチポリラインの場合とシングルポリラインの場合があるので、条件を分けて処理します。CZML データを生成後、155-165 行目で高さを調整します。これは、データの持つ高さ情報が Cesium の地形情報があつてないため必要な調整です。

✧ [「Cesium」を利用して、任意の座標から高度を取得する \(nnao-web.boo.jp\)](#)

```
sampleTerrainMostDetailed(terrainProvider, positions).then((updatedPositions) => {
  try{
    if (czmlObj[i + 1]?.polyline?.positions?.cartographicDegrees) {
      for (let j = 2; j < czmlObj[i + 1]?.polyline?.positions?.cartographicDegrees.length; j = j + 3) {
        let newHeight = parseFloat(updatedPositions[((j + 1) / 3) - 1].height);
        czmlObj[i + 1].polyline.positions.cartographicDegrees[j] = newHeight + (parseFloat(czmlObj[i + 1].polyline.positions.cartographicDegrees[j]));
      }
    }
  }
})
```

- 最後に 167-176 行目で生成したデータを地図に追加します。まずは 167-171 行目で既存の経路探索結果表示物を消去します。その後、CzmlCatalogItem オブジェクトを生成し、czmlString として、生成した CZML データをセットします。item.loadMapItems() 関数を呼び出してアイテムを追加し、terria.workbench.add(item) 関数を呼び出してワークベンチ上に経路探索結果を追加します。

```
item.setTrait(CommonStrata.user, "name", "経路探索 " + routeItems[searchType - 1]?.title);
item.setTrait(CommonStrata.definition, "czmlString", JSON.stringify(czmlObj));
item.loadMapItems();
this.state.terria.workbench.add(item);
```

- 次に、経路検索画面を呼び出すボタンを作成します。ヘルプボタンなど既存のボタンコンポーネントが格納されている場所に以下のファイルを作成します。
 - ✧ [TerriaMap¥packages¥terriaajs¥lib¥ReactViews¥Map¥RouteButton¥RouteButton.tsx](#)
- 要素の定義とクリックイベントの定義をします。
- 次は、既存ソースの修正です。
- まずは以下の場所にある viewState.ts を修正します。
 - ✧ [TerriaMap¥packages¥terriaajs¥lib¥ReactViewModels¥ViewState.ts](#)

- パネルの開閉状態を管理する、showRouteMenu と routePanelExpanded を observable として定義します（106-108 行目）。

```
// 経路探索

@observable showRouteMenu: boolean = false;

@observable routePanelExpanded: boolean = false;
```

- パネルの表示・非表示を切り替える以下の関数の定義を追加します（639-667 行目）。

```
//経路探索パネルの表示

[action]
showRoutePanel() {
    //this.terria.analytics?.logEvent(Category.route, CustomAction.panelOpened);

    this.showRouteMenu = true;
    this.routePanelExpanded = false;
    this.terria.setMode(2);
    this.terria.setRouteStart("");
    this.terria.setRouteEnd("");
    this.setTopElement("RoutePanel");
}

//経路探索パネルの初期化

[action]
clearRoutePanel() {
    this.terria.setMode(2);
    this.terria.setRouteStart("");
    this.terria.setRouteEnd("");
}

//経路探索パネルの非表示

[action]
hideRoutePanel() {
    this.terria.setMode(0);
    this.terria.setRouteStart("");
    this.terria.setRouteEnd("");
    this.showRouteMenu = false;
    this.setTopElement("");
}
```

- ✧ showRoutePanel
 - ✧ clearRoutePanel
 - ✧ hideRoutePanel
- TerriaMap¥packages¥terriajs¥lib¥Models¥Terria.ts に observable と setter を定義します。(553-608 行目)
 - mode: 操作モード
 - routeStart 開始地点
 - routeEnd 終了地点
 - clickLatLong クリック地点の緯度経度

```
/*
 * 経路検索におけるスタート地点の経度緯度
 * @type {string}
 */
@observable routeStart = "";

/**
 * 経路探索 開始地点の緯度経度をセット
 * @param {string}
 */
@action
setRouteStart(routeStart: string) {
  this.routeStart = routeStart;
}
```

- TerriaMap¥packages¥terriajs¥lib¥Models¥Cesium.ts にある、地図画面左クリックイベント定義を修正します。(307-381 行目)
 - ※以下を追加で import します。
 - import CommonStrata from "./Definition/CommonStrata";
 - import GeoJsonCatalogItem from "./Catalog/CatalogItems/GeoJsonCatalogItem";

```
inputHandler.setInputAction(e => {
  // ここにイベント処理を実装
}, ScreenSpaceEventType.LEFT_CLICK);
```

- イベント処理の実装内容を説明します。308-314 行目では、クリックしたピクセル位置から緯度経度を求めていきます。

```
const pickRay = this.scene.camera.getPickRay(e.position);
const pickPosition = this.scene.globe.pick(pickRay, this.scene);
const pickPositionCartographic =
  pickPosition && Ellipsoid.WGS84.cartesianToCartographic(pickPosition);
if (pickPositionCartographic) {
  const latitude = CesiumMath.toDegrees(pickPositionCartographic.latitude);
  const longitude = CesiumMath.toDegrees(pickPositionCartographic.longitude);
```

- 続いて、terria.mode の値によって処理を分岐しています。以下 mode=2 で経路探索の開始地点を追加する処理を説明します（321-348 行目）。
- 緯度経度の値を routeStart にセットし、開始地点の入力部に表示します。また、既存の開始地点と終了地点のレイヤを workbench から削除します。

```
//mode=2 の時は経路探索の開始地点を追加
} else if (this.terria.mode === 2) {
  this.terria.setRouteStart(latitude + "," + longitude);
  this.terria.setMode(3);
  const routeStart = document.getElementById('routeStart') as HTMLInputElement;
  routeStart.value = this.terria.routeStart;
  const items = this.terria.workbench.items;
  for (const aItem of items) {
    if (aItem.uniqueId === '開始地点' || aItem.uniqueId === '終了地点') {
      this.terria.workbench.remove(aItem);
    }
  }
}
```

- 次に、開始地点のポイントフィーチャレイヤを作成し、workbench に追加しています。

```

const pointItem = new GeoJsonCatalogItem("開始地点", this.terria);
pointItem.setTrait(CommonStrata.user, "geoJsonData", {
    type: "Feature",
    properties: {
        "marker-color": "#ffff00",
        "searchMode": "on",
        "経度": longitude,
        "緯度": latitude,
    },
    geometry: {
        type: "Point",
        coordinates: [longitude, latitude]
    }
});
pointItem.setTrait(CommonStrata.user,
"idealZoom", {targetLongitude:longitude,targetLatitude:latitude,targetHeight:100,heading:0 ,pitch:30 ,range:500});
pointItem.loadMapItems();
this.terria.workbench.add(pointItem);

```

- 続いて、新しく作成したコンポーネントを親コンポーネントに追加していきます。
- TerriaMap¥packages¥terriajs¥lib¥ReactViews¥StandardUserInterface¥StandardUserInterface.jsx を開きます。
- RoutePanel.jsx を import します。
 - import RoutePanel from "../Map/Panels/RoutePanel/RoutePanel";
- HelpPanel を読み込んでいる箇所の直下に RoutePanel のコンポーネントを追加します（479-481 行目）。

```

{this.props.viewState.showHelpMenu &&
  this.props.viewState.topElement === "HelpPanel" && (
    <HelpPanel terria={terria} viewState={this.props.viewState} />
  )}
{this.props.viewState.showRouteMenu &&
  <RoutePanel terria={terria} viewState={this.props.viewState} />
}

```

- `TerriaMap$packages$terriajslibReactViewsMapMenuBar.jsx` を開きます。
- `RouteButton.jsx` を Import します。
 - `import RouteButton from "./RouteButton/RouteButton";`
- `RouteButton` コンポーネントを追加します。

```
<ul className={classNames(Styles.menu)}>
  <li className={Styles.menuItem}>
    <RouteButton
      viewState={props.viewState}
    />
  </li>
</ul>
```

- カスタム機能向けに以下の設定ファイルを追加します。
 - `TerriaMap$packages$terriajs$customconfig.json`
- `Config.apiUrl` が API のトップ URL になるので、環境に合わせて修正してください。

```
{
  "config": {
    "apiUrl": "http://localhost/api"
  }
}
```

3) 変更したソースデプロイ、連携設定方法

- 設定ファイルを更新します。
- まず、初期表示設定ファイル (JSON) を `TerriaMap$wwwroot$init` 配下に作成します。名前は任意ですが、今回は `hands_on.json` とします。同じフォルダ内の `plateauview.json` をベースに修正しています。
- 修正する際は、下記 Terria の公式ドキュメントを参考にしてください。
 - <https://docs.terria.io/guide/customizinginitialization-files/>
 - https://www.mlit.go.jp/plateau/learning/tpc07-1/#p7_3_1
- `initialCamera` と `homeCamera` を修正し、初期表示位置 (緯度経度、高さ)・方向を変更します。`initialCamera` が初期表示位置、`homeCamera` がホームボタン押下時の方向です。

```
{
  "initialCamera": {
    "west": 132.44260414011882,
    "south": 34.37661020684377,
    "east": 132.52218201535285,
    "north": 34.4134506523117,
    "position": {
      "x": -3562077.3036708836,
      "y": 3889724.314190411,
      "z": 3579840.7448215405
    },
    "direction": {
      "x": 0.6019807825356082,
      "y": -0.6573521815742026,
      "z": 0.4533290712467638
    },
    "up": {
      "x": -0.30616195866755636,
      "y": 0.33432334932262747,
      "z": 0.8913432297171197
    }
  },
  "homeCamera": {
    "west": 132.3,
    "south": 34.25,
    "east": 132.6,
    "north": 34.55
  }
},
```

- 背景地図を修正する場合、baseMaps を修正します。

```
"baseMaps": {
  "items": [
    {
      "item": {
        "type": "composite",
        "id": "/basemap//全国最新写真（シームレス）",
        "label": "全国最新写真（シームレス）"
      }
    }
  ]
},
```

```

    "name": "全国最新写真アホ (シームレス)",
    "members": [
      {
        "type": "cesium-terrain",
        "id": "/basemap//全国最新写真 (シームレス)/terrain",
        "name": "tokyo-23ku-terrain",
        "ionAccessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJqdGkiOiI3NGI5ZDM0Mi1jZDIZLTrmMzEtOTkwYi0zzTk4Yzk3ODZlNzQiLCJpZCI6NDA2NDYsImhdCI6MTYwODk4MzAwOH0.3rc062ErML11TMSEflsMqeUTCDbIH6o4n415ssssuedE",
        "ionAssetId": 286503,
        "attribution": "地形データ: 基盤地図情報数値標高モデルから作成  
(測量法に基づく国土地理院長承認(使用)R 3JHs 259)"
      },
      {
        "type": "open-street-map",
        "opacity": 1,
        "id": "/basemap//航空写真/imagery",
        "name": "航空写真",
        "url": "https://gic-plateau.s3.ap-northeast-1.amazonaws.com/2020/ortho/tiles/",
        "fileExtension": "png",
        "attribution": "東京 23 区の地形と空中写真"
      },
    ],
  
```

- データを追加する場合、catalog を編集します。
 - id は一意の ID をセットしてください。
 - type:group を設定すると、階層構造を設定できます。子階層のデータは members に追加します。
 - 建物モデル (3DTiles) は type:3d-tiles を設定します。ソースの URL を url に設定します。type:composite とすることで、複数のソースのデータを一つのデータセットとして表示できます。
 - GeoServer のレイヤ (WMS) は type:wms を設定します。アクセス URL を url に設定します。アクセス URL のフォーマットは「http://[Web サーバの IP アドレス]/geoserver/[ワークスペース名]/wms」です。layers にはレイヤ名を設定します。フォーマットは「[ワークスペース名]:[レイヤタイトル]」です。パラメータがある場合、parameters で指定します。パラメータによってレイヤの表示を切り替えたい場合、以下参考にレイヤを作成し、viewparams でパラメータをコロン、セミコロン区切りの文字列で指定します。

✧ [SQL Views — GeoServer 2.23-SNAPSHOT User Manual](#)

```
"catalog": [
  {
    "id": "//データセット",
    "name": "データセット",
    "type": "group",
    "isOpen": true,
    "members": [
      {
        "id": "//データセット/建物モデル",
        "name": "建物モデル",
        "type": "composite",
        "hideSource": true,
        "members": [
          {
            "id": "//データセット/建物モデル(広島市中区)",
            "name": "建物モデル(広島市中区)",
            "type": "3d-tiles",
            "hideSource": true,
            "url":
              "https://assets.cms.plateau.reearth.io/assets/68/444ea3-37c5-451d-95c2-19ffc8828a70/34100_hirosima-shi_2022_3dtiles_1_op_bldg_34101_naka-ku_lod1/tileset.json"
          },
          {
            "id": "//データセット/建物モデル(広島市東区)",
            "name": "建物モデル(広島市東区)",
            "type": "3d-tiles",
            "hideSource": true,
            "url":
              "https://assets.cms.plateau.reearth.io/assets/74/8795a3-6d2e-4038-8a86-21d2b5e4da8e/34100_hirosima-shi_2022_3dtiles_1_op_bldg_34102_higashi-ku_lod1/tileset.json"
          }
        ]
      }
    ]
  }
]
```

```

        "id": "//データセット/建物モデル(広島市南区)",
        "name": "建物モデル(広島市南区)",
        "type": "3d-tiles",
        "hideSource": true,
        "url":

"https://assets.cms.plateau.reearth.io/assets/d9/610e68-ade4-4780-9ab2-
a9470335c760/34100_hirosima-shi_2022_3dtiles_1_op_bldg_34103_minami-
ku_lod1/tileset.json"
    }
]
},
{
        "id": "//データセット/歩行者空間ネットワーク",
        "name": "歩行者空間ネットワーク",
        "type": "wms",
        "url": "http://localhost:8083/geoserver/handson/wms",
        "layers": "handson:link_3d"
    }
]
}
],

```

- 初期表示するレイヤの ID を workbench に設定します。

```

"workbench": [
    "//データセット/建物モデル",
    "//データセット/歩行者空間ネットワーク"
]

```

- 続いて、TerriaMap¥wwwroot¥config.json を編集します。「initializationUrls」を読み込む初期表示設定ファイル（今回の場合「hands_on」）に変更します。

```

{
    /* Names of init files (in wwwroot/init), without the .json extension,
    to load by default */

    "initializationUrls": [ "hands_on" ],
    "parameters": {
        "googleUrlShortenerKey": null,
        "googleAnalyticsKey": null,

```

```
"googleAnalyticsOptions": null,
```

- 変更したソースを反映してデプロイします。
- 修正したファイル一式を Docker 上の PLATEAU VIEW にコピーします。
- docker cp コマンドでコピーします。TerriaMap、terriajs を格納しているディレクトリに移動し、以下のコマンドをコマンドプロンプトから実行します。

```
docker cp TerriaMap plateau_demo:/usr/local/work
```

```
docker cp terriajs plateau_demo:/usr/local/work/TerriaMap/packages/
```

- Docker のターミナルに入り、PLATEAU VIEW のディレクトリに移動して、以下コマンドを実行します。※ TerriaMap¥wwwroot 以下のファイルのみを修正した場合はコピーのみで、コマンド実行は不要です。

```
yarn gulp
```

```
yarn start
```

- PLATEAU VIEW の動作が重い場合、NODE_OPTIONS の環境変数が適切に設定されているか Docker のターミナルから確認してください。設定されていない場合は、適切に設定し、再度 yarn gulp から実行してください。

```
export -p
```

```
export NODE_OPTIONS=--max_old_space_size=4096
```

- PLATEAU VIEW の初期画面を開いたままロードが進まない場合、Tomcat に割り当てられた JVM メモリサイズが足りていないため GeoServer が正常に動作しておらず、初期読み込みのレイヤが読み込めていないことが影響している場合があります。その場合は Docker のターミナルに入り、JVM メモリサイズを引き上げてください

```
vi $CATALINA_HOME/bin/setenv.sh
```

```
# 編集モードで以下の内容を記述し保存します。
```

```
export CATALINA_OPTS='-Xms512m -Xmx1024m -Xss1024k -XX:PermSize=512m -XX:MaxPermSize=1024m -XX:+UseConcMarkSweepGC -XX:+CMSClassUnloadingEnabled'
```

```
# tomcat を再起動します。
```

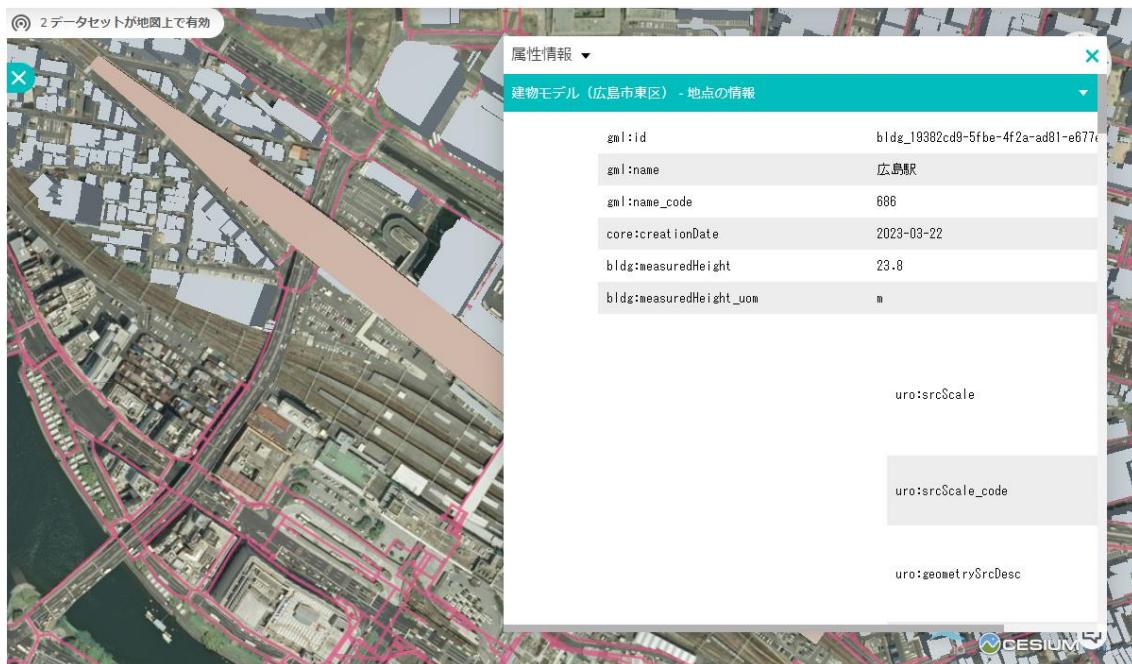
```
exec $CATALINA_HOME/bin/catalina.sh run
```

(5) 動作説明

- <http://localhost:8080/plateau/> にアクセスすると、前回アクセス時と異なり、広島市にズームされ、歩行空間ネットワークが表示された地図が表示されます。

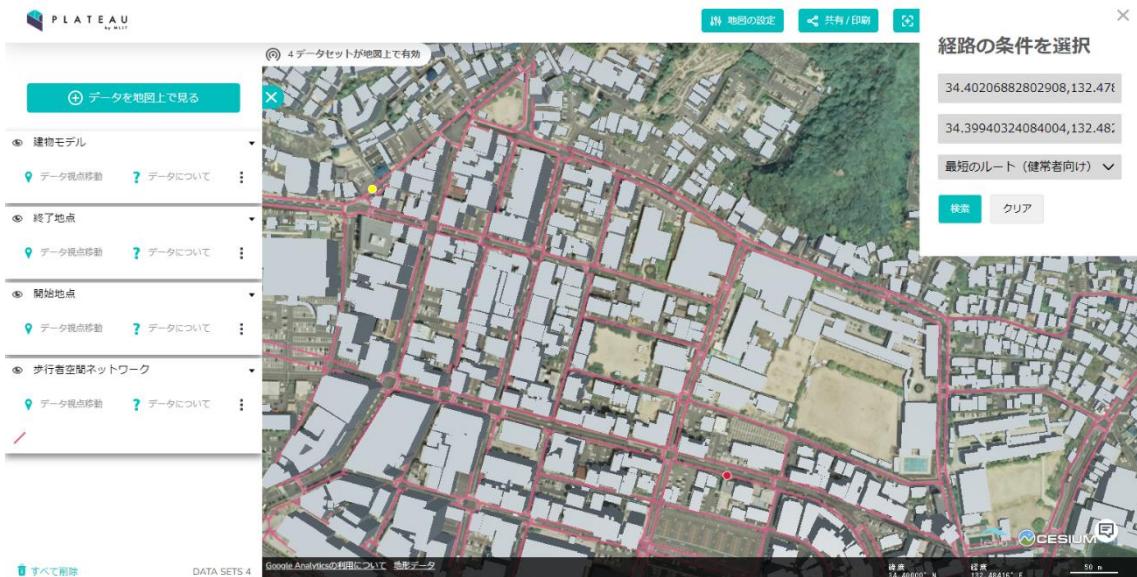


- 地図はホールドすることで移動できます。
- 建造物や歩行空間ネットワークをクリックすると、そのオブジェクトの属性情報が表示されます。

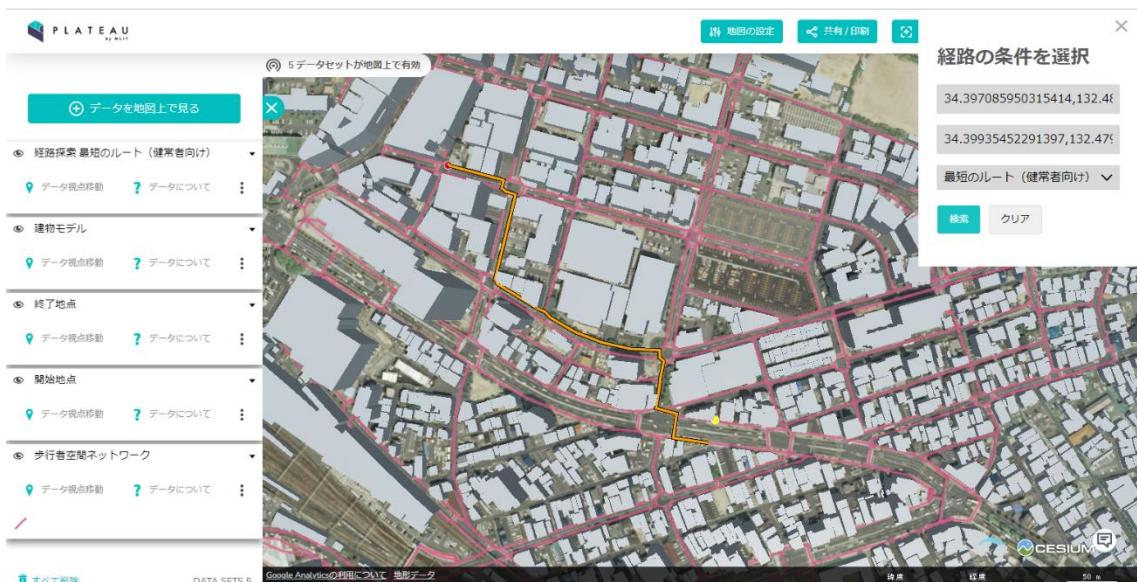




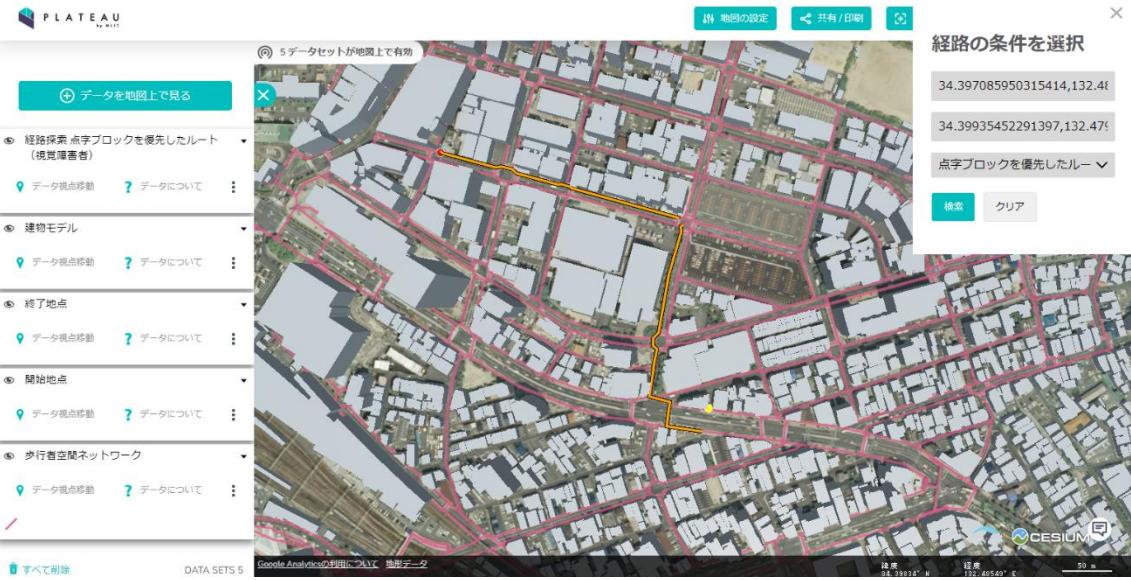
- 経路探索
 - 右上の「経路」ボタンを押下します。
 - 地図上で歩行者空間ネットワークの開始地点と終了地点をクリックします。地図上では開始地点が黄色いポイント、終了地点は赤いポイント表示されます。左の表示レイヤで「開始地点」と「終了地点」が追加され、右上の経路探索設定画面では座標が設定されていることを確認してください。



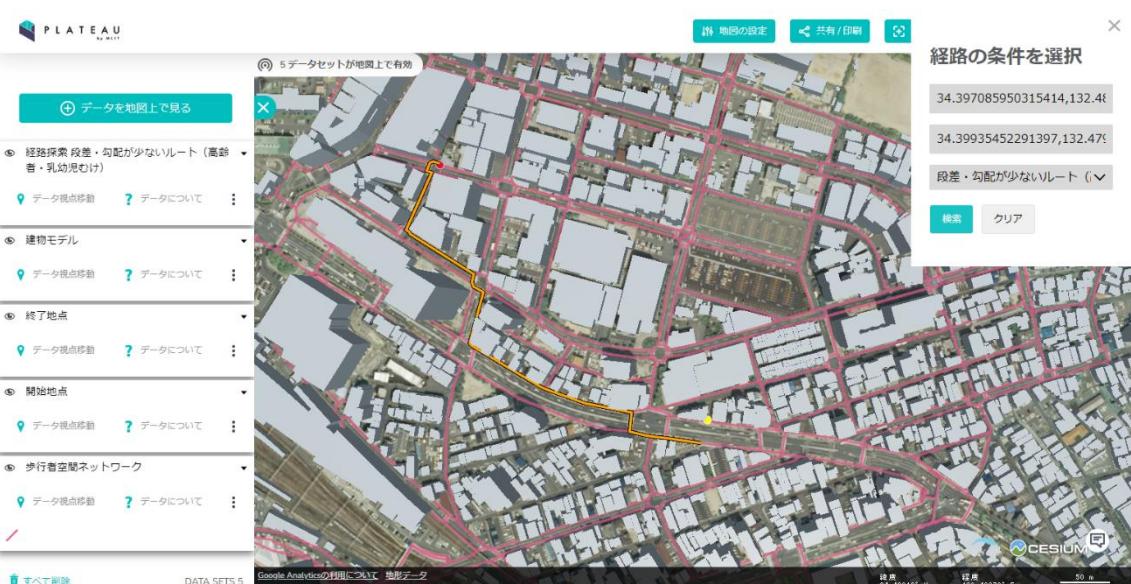
- 入力しなおす場合は「クリア」を押下します。
- プルダウンから条件を設定できます。「検索」ボタンを押下し結果を表示します。



・最短のルートの表示



・点字ブロックを優先したルートの表示



・段差・勾配が少ないルートの表示