OGC® Routing Pilot ER

Table of Contents

1. Subject . . . 4
2. Executive Summary . . . 5
  2.1. Document contributor contact points . . . 7
  2.2. Foreword . . . 7
3. References . . . 8
4. Terms and definitions . . . 9
  4.1. Abbreviated terms . . . 9
5. Overview . . . 10
6. Background . . . 11
  6.1. The OGC API - Processes specification . . . 11
  6.2. Overview of Pilot architecture . . . 12
7. Pilot Technical Architecture . . . 13
  7.1. Scenarios . . . 13
    7.1.1. Online Scenario . . . 13
    7.1.2. Intermittent Scenario . . . 14
    7.1.3. Offline Scenario . . . 15
  7.2. Client Execution Patterns . . . 15
    7.2.1. Synchronous . . . 15
    7.2.2. Asynchronous Polling . . . 16
    7.2.3. Callback . . . 17
    7.2.4. Server-sent Events . . . 17
  7.3. API Pattern Options . . . 18
  7.4. Input Datasets . . . 19
8. Pilot Component Implementations . . . 21
  8.1. Client Implementations . . . 21
    8.1.1. Helyx QGIS Client . . . 21
    8.1.2. GIS-FCU Client . . . 32
    8.1.3. Ecere Client . . . 35
  8.2. OGC API - Processes Profile Implementations (WPSs) . . . 37
    8.2.1. 52North WPS . . . 37
    8.2.2. GIS-FCU WPS . . . 47
    8.2.3. Skymantics WPS . . . 50
    8.2.4. Helyx WPS . . . 54
  8.3. Routing Engine Implementations . . . 58
    8.3.1. Skymantics Routing Engine . . . 58
    8.3.2. Ecere Routing Engine . . . 71
9. Technology Integration Experiments (TIEs) . . . 78
10. Pilot Recommendations . . . 85

10.1. OGC Web Interface Recommendations . . . 85
10.2. OGC Routing Recommendations . . . 85
10.2.1. Skymantics Recommendations . . . 85
10.2.2. Helyx Recommendations . . . 86
10.2.3. Ecere Recommendations . . . 86
11. Conclusions . . . 88
Appendix A: Routing Exchange Model . . . 89
A.1. Overview . . . 89
A.2. Requirements class "Route Exchange Model (core)" . . . 90
A.3. Requirements class "Route Exchange Model (full)" . . . 90
A.4. Requirements class "Route Exchange Model (overview)" . . . 94
A.5. Requirements class "Route Exchange Model (segment)" . . . 95
A.6. Requirements class "Route Exchange Model (segment with links)" . . . 96
Appendix B: Revision History . . . 98

Publication Date: 2020-01-08

Approval Date: 2019-11-22

Submission Date: 2019-09-08

Reference number of this document: OGC 19-041r3

Reference URL for this document: http://www.opengis.net/doc/PER/routing-pilot-er

Category: OGC Public Engineering Report

Editor: Sam Meek, Theo Brown, Clemens Portele

Title: OGC® Routing Pilot ER

OGC Public Engineering Report

COPYRIGHT

Copyright © 2019 Open Geospatial Consortium. To obtain additional rights of use, visit http://www.opengeospatial.org/

WARNING

This document is not an OGC Standard. This document is an OGC Public Engineering Report created as a deliverable in an OGC Interoperability Initiative and is not an official position of the OGC membership. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an OGC Standard. Further, any OGC Public Engineering Report should not be referenced as required or mandatory technology in procurements. However, the discussions in this document could very well lead to the definition of an OGC Standard.

LICENSE AGREEMENT

Permission is hereby granted by the Open Geospatial Consortium, ("Licensor"), free of charge and subject to the terms set forth below, to any person obtaining a copy of this Intellectual Property and any associated documentation, to deal in the Intellectual Property without restriction (except as set forth below), including without limitation the rights to implement, use, copy, modify, merge, publish, distribute, and/or sublicense copies of the Intellectual Property, and to permit persons to whom the Intellectual Property is furnished to do so, provided that all copyright notices on the intellectual property are retained intact and that each person to whom the Intellectual Property is furnished agrees to the terms of this Agreement.

If you modify the Intellectual Property, all copies of the modified Intellectual Property must include, in addition to the above copyright notice, a notice that the Intellectual Property includes modifications that have not been approved or adopted by LICENSOR.

THIS LICENSE IS A COPYRIGHT LICENSE ONLY, AND DOES NOT CONVEY ANY RIGHTS UNDER ANY PATENTS THAT MAY BE IN FORCE ANYWHERE IN THE WORLD. THE INTELLECTUAL PROPERTY IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT OF THIRD PARTY RIGHTS. THE COPYRIGHT HOLDER OR HOLDERS INCLUDED IN THIS NOTICE DO NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE INTELLECTUAL PROPERTY WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE INTELLECTUAL PROPERTY WILL BE UNINTERRUPTED OR ERROR FREE. ANY USE OF THE INTELLECTUAL PROPERTY SHALL BE MADE ENTIRELY AT THE USER’S OWN RISK. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR ANY CONTRIBUTOR OF INTELLECTUAL PROPERTY RIGHTS TO THE INTELLECTUAL PROPERTY BE LIABLE FOR ANY CLAIM, OR ANY DIRECT, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES, OR ANY DAMAGES WHATSOEVER RESULTING FROM ANY ALLEGED INFRINGEMENT OR ANY LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR UNDER ANY OTHER LEGAL THEORY, ARISING OUT OF OR IN CONNECTION WITH THE IMPLEMENTATION, USE, COMMERCIALIZATION OR PERFORMANCE OF THIS INTELLECTUAL PROPERTY.

This license is effective until terminated. You may terminate it at any time by destroying the Intellectual Property together with all copies in any form. The license will also terminate if you fail to comply with any term or condition of this Agreement. Except as provided in the following sentence, no such termination of this license shall require the termination of any third party end-user sublicense to the Intellectual Property which is in force as of the date of notice of such termination. In addition, should the Intellectual Property, or the operation of the Intellectual Property, infringe, or in LICENSOR’s sole opinion be likely to infringe, any patent, copyright, trademark or other right of a third party, you agree that LICENSOR, in its sole discretion, may terminate this license without any compensation or liability to you, your licensees or any other party. You agree upon termination of any kind to destroy or cause to be destroyed the Intellectual Property together with all copies in any form, whether held by you or by any third party.

Except as contained in this notice, the name of LICENSOR or of any other holder of a copyright in all or part of the Intellectual Property shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Intellectual Property without prior written authorization of LICENSOR or such copyright holder. LICENSOR is and shall at all times be the sole entity that may authorize you or any third party to use certification marks, trademarks or other special designations to

indicate compliance with any LICENSOR standards or specifications.

This Agreement is governed by the laws of the Commonwealth of Massachusetts. The application to this Agreement of the United Nations Convention on Contracts for the International Sale of Goods is hereby expressly excluded. In the event any provision of this Agreement shall be deemed unenforceable, void or invalid, such provision shall be modified so as to make it valid and enforceable, and as so modified the entire Agreement shall remain in full force and effect. No decision, action or inaction by LICENSOR shall be construed to be a waiver of any rights or remedies available to it.

None of the Intellectual Property or underlying information or technology may be downloaded or otherwise exported or reexported in violation of U.S. export laws and regulations. In addition, you are responsible for complying with any local laws in your jurisdiction which may impact your right to import, export or use the Intellectual Property, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

Chapter 1. Subject

The goal of this OGC Routing Pilot Engineering Report (ER) is to document the proof of concept of an Application Programming Interface (API) conforming to a profile of the draft OGC API - Processes specification that allows implementation of vector routing across one or more routing engines. The components implemented in the OGC Open Routing API Pilot 2019 included two clients, interfacing with three implementations of the draft OGC API - Processes specification that in turn communicated with three routing engines. This work resulted in the definition of a proposed common interface and data exchange model supported by all components for requesting, generating and returning routes.

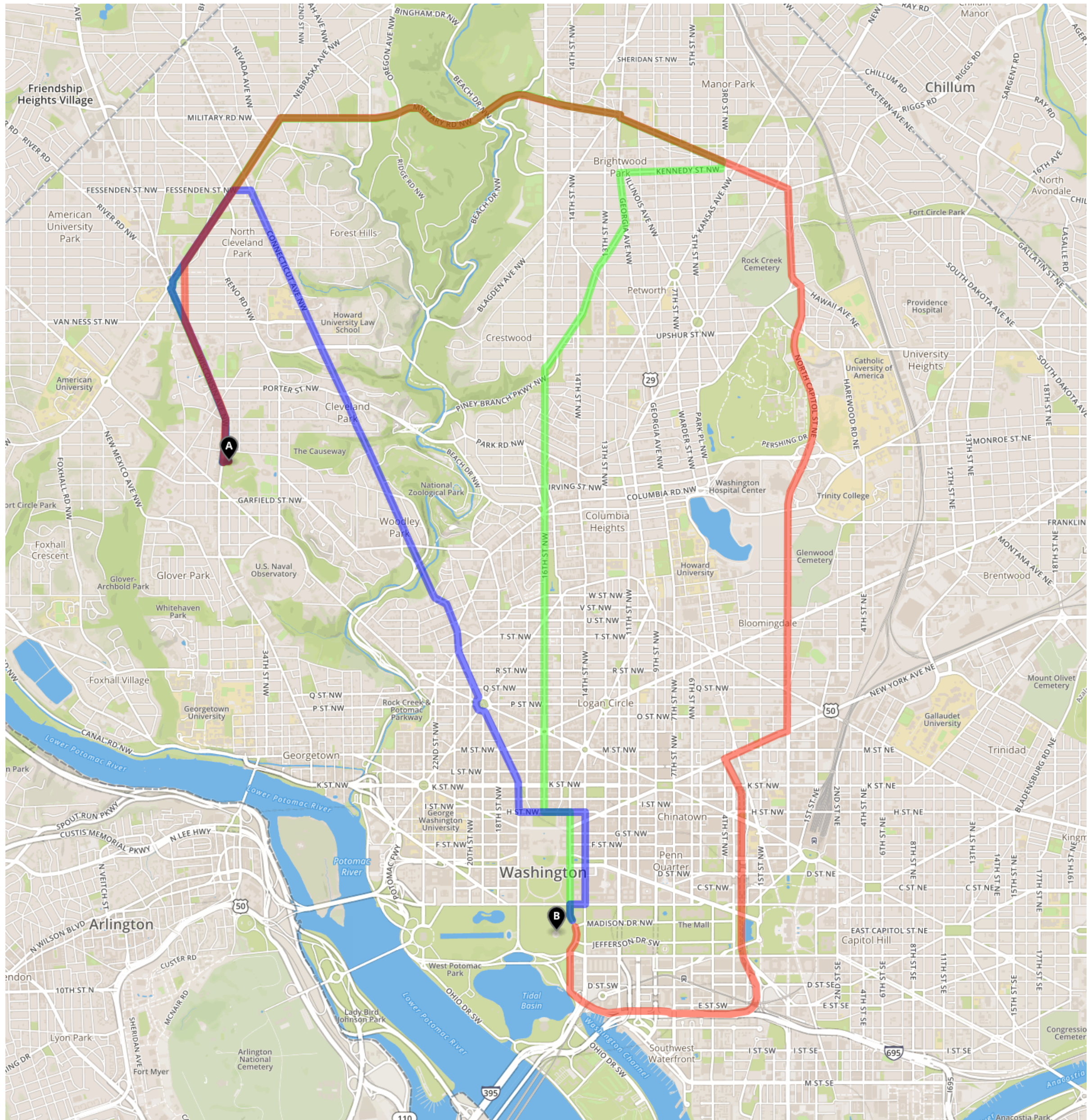

Figure 1. Route Examples

Chapter 2. Executive Summary

This document describes the implementations to support the OGC Open Routing API Pilot 2019. The purpose of the work was to utilize new and emerging OGC standards to support routing applications in denied, degraded, intermittent or low bandwidth (DDIL) environments as these are of particular importance to the sponsor. A secondary requirement was to define and implement an exchange model for routing information. The requirements for this model were: That the model uses GeoJSON and that it was lightweight enough to support exchange in DDIL environments. Contextually, DDIL has been a concern within the OGC for a number of years. This Pilot was a further exploration of strategies to support the environment.

The DDIL environment requirement was expressed as three scenarios:

- Online - where the client has full connectivity with the server components.
- Offline - the client has no connectivity and will not have connectivity.
- Intermittent - one of the clients has connectivity, but the rest do not.

The scenarios created requirements for the component operations. The online scenario required interoperability between all clients, all OGC APIs, all routing engines (HERE, Skymantics and Ecere) that utilize all datasets. The Offline scenario required all components to be co-located in order to maintain functionality whilst removing the reliance on connectivity. This resulted in a sub-set of options made available to the user. The intermittent scenario required a single client to have access to online routing capabilities and the route be shared among the clients using the Pilot routing exchange model encapsulated in a GeoPackage. The components utilized in the Pilot include:

- Web clients:
  - Ecere
  - GIS-FCU
  - Skymantics
- Desktop client:
  - Helyx SIS - QGIS
- 3D client
  - Helyx SIS - Cesium
- OGC API - Processing instances
  - 52 Degrees North
  - Helyx SIS
  - Skymantics
  - GIS-FCU
- Routing engines:
  - HERE
  - Skymantics

    - Ecere
    - OSRM
- Datasets for routing:
    - HERE
    - NSG
    - OSM

The experiments involved components from different participants, combining the client capabilities with a variety of processing services end-points implementing the Open Routing API (Ecere, Skymantics, Helyx, 52° North and GIS-FCU), three different routing engines used for computing the routes (Ecere, Skymantics and HERE), and different source data for the roads network used by those calculations (OpenStreetMap, HERE and NSG). The interoperability between the clients, OGC APIs, routing engines and data sources has shown that the new OGC APIs are fit for purpose in DDIL environments given the sponsor scenarios. Additionally, the distribution of routing data between the clients in the intermittent scenario provided a mechanism for utilizing networks in sub-optimal connectivity scenarios, it is likely that this approach to architecture and development can be followed in future innovation endeavors.

The outputs of the Pilot consisted of several components with multiple implementations of each type of component. The component types included; clients for web and desktop use cases, routing engines to do the routing work, datasets to support the routing engines and OGC APIs to mediate the communication between the clients and the routing engines in a standardized manner. The data exchange was performed using the Routing Exchange Model, which is a lightweight version of GeoJSON and described fully in Annex A.

The Pilot included an exploration of transport mechanisms for data over the web. Typical requests to web services are either synchronous or asynchronous. The synchronous request was useful for small routes as they can be generated and the routes returned in a timely manner without causing the client or server to visibly lock. The asynchronous methods included:

- Polling - checking the server at a defined interval to see whether a result has been generated.
- Callback - providing the server with a URL on the client side to post the result to.
- Server Sent Events (SSE) - a one-way web socket approach.

The different methods are available in the QGIS client as selectable parameters for demonstration and performance testing purposes.

The OGC API is an emerging standard and at time of writing more of a set of practices. The approach adopted in this Pilot was to implement *conformance classes*, these are then used to build the clients automatically to expose the features of the API, in this case, the routing engines. It is recommended that this approach to APIs be adopted across the OGC suite as it provides an efficient method of delivering interoperable capability without having to heavily specify clients.

Future work for the Pilot includes development of the OGC interfaces to support further routing constraints such as elevation, restricted maneuvers, speed limits, street hierarchies and transportation methods. Uplifting the routing engines would enable further testing with eventual convergence on real-world requirements. Additionally, OGC APIs should be explored further by

understanding API efficiency, structure and potential bottlenecks.

2.1. Document contributor contact points

All questions regarding this document should be directed to the editor or the contributors:

Contacts

Name	Organization	Role
Sam Meek	Helyx SIS	Editor
Theo Brown	Helyx SIS	Editor
Clemens Portele	interactive instruments GmbH	Editor
Christian Autermann	52°North GmbH	Contributor
Ricky Lin	GIS.FCU	Contributor
Nacho Correas	Skymantics	Contributor
Josh Lieberman	Open Geospatial Consortium	Contributor
Donovan Dall	Helyx SIS	Contributor
Jérôme Jacovella-St-Louis	Ecere	Contributor

2.2. Foreword

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

Chapter 3. References

The following normative documents are referenced in this document.

- OGC: OGC 06-121r9, OGC® Web Services Common Standard (2015) [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2]
- OGC: OGC 14-065r2, OGC® WPS 2.0.2 Interface Standard Corrigendum 2 (2018) [http://docs.opengeospatial.org/is/14-065/14-065.html]
- OGC: OGC 09-025r2, OGC Web Feature Service 2.0 Interface Standard – With Corrigendum (2014) [http://docs.opengeospatial.org/is/09-025r2/09-025r2.html]
- OGC: OGC 17-069r3, OGC API - Features - Part 1: Core (2019) [http://docs.opengeospatial.org/is/17-069r3/17-069r3.html]
- IETF: RFC 7946 - The GeoJSON Format [https://tools.ietf.org/html/rfc7946]

Chapter 4. Terms and definitions

For the purposes of this report, the definitions specified in Clause 4 of the OWS Common Implementation Standard OGC 06-121r9 [https://portal.opengeospatial.org/files/?artifact_id=38867&version=2] shall apply. In addition, the following terms and definitions apply.

- **application programming interface**

  standard set of documented and supported functions and procedures that expose the capabilities or data of an operating system, application or service to other applications (adapted from ISO/IEC TR 13066-2:2016)

- **feature**

  abstraction of real-world phenomena (source: ISO 19101-1:2014)

- **OpenAPI definition | OpenAPI document**

  a document (or set of documents) that defines or describes an API and conforms to the OpenAPI Specification [derived from the OpenAPI Specification]

- **Web API**

  API using an architectural style that is founded on the technologies of the Web [derived from the W3C Data on the Web Best Practices]

NOTE Best Practice 24: Use Web Standards as the foundation of APIs [https://www.w3.org/TR/dwbp/#APIHttpVerbs] in the W3C Data on the Web Best Practices provides more detail.

4.1. Abbreviated terms

- API Application Programming Interface
- GML Geography Markup Language
- HATEOAS Hypermedia As The Engine Of Application State
- JSON JavaScript Object Notation
- WFS Web Feature Service
- WPS Web Processing Service
- REST Representational State Transfer

Chapter 5. Overview

Section 6 introduces the motivation for conducting the Pilot. It describes in brief the ongoing OGC discussion regarding the modernization of OGC web interfaces and how this has been accounted for during the pilot. This section also discusses the requirements set by the sponsors in the form of three scenarios.

Section 7 discusses the overarching architecture of the pilot, taking into account the interactions between the various components. The delivery of the sponsor requirements is explained and the commonalities across the various pilot implementations is also outlined.

Section 8 contains the details of each participant implementation, providing insight into the specific design of implementations, the challenges encountered across the pilot and examples of how to implement the API and exchange model.

Section 9 documents the Technology Integration Experiments.

Section 10 discusses recommendations generated during the pilot.

Section 11 contains the conclusions of the pilot work.

Annex A Details the Routing Exchange Model developed during the Pilot.

Chapter 6. Background

This OGC Engineering Report (ER) describes the work done in the OGC Open Routing API Pilot 2019. The two major concepts in this Pilot, routing capabilities and OGC API approaches, have been previously explored in depth by a number of OGC endeavors. The purpose of this section is to draw upon this previous work to provide a context for the OGC Routing API Pilot 2019.

The previous work for this ER includes, but is not limited to work described in the following documents:

Next generation APIs:

- 18-045 OGC Testbed-14: Next Generation Web APIs - WFS 3.0 Engineering Report - At the time of writing this report represents the most current documented approach to the OGC API.
- 18-021 OGC Testbed-14: Next Generation APIs - Complex Feature Handling Engineering Report - Although the Routing Pilot does not deal with complex features, this report is useful for understanding the modular approach of the OGC API, specifically the topic of extensions to the API.

Routing Capabilities:

- 16-029r1 Testbed-12 GeoPackage Routing and Symbology Engineering Report - This report provides insight into how a routing GeoPackage may be used in a mobile application, providing context for the browser and desktop clients documented in this ER.
- 07-074 OpenGIS Location Service (OpenLS) Implementation Specification: Core Services – An OGC standard consisting of the composite set of basic services comprising the OpenLS Platform.
- 08-028r7 OpenGIS Location Services (OpenLS): Part 6 - Navigation Service (1.0.0) - the corresponding navigation capability for the OpenLS Interface Standard.

6.1. The OGC API - Processes specification

NOTE	Prior to the OGC API – Processes naming convention, the specification was referred to as WPS 3.0. The official name of the specification is now OGC API – Processes. This ER therefore, at times, acceptably refers to implementations of OGC API – Processes as WPS.

- 18-036 OGC Testbed-14: WPS-T Engineering Report - Provides useful insight into some of the more recent WPS 2.0 work, as context for moving the processing capability to an OGC API.

The recent advances to the Web Feature Service (WFS) and Web Processing Service (WPS) standards to support a modern API approach have been significant. This work draws upon elements of the OpenAPI specification and Representational State Transfer (REST) principles to provide a modular *building block* approach to web interfaces. It is an ongoing process to reach a consensus in the OGC for the next generation standards. The emerging OGC API – Processes draft specification (which is based on WPS) should support a routing capability that conforms to this modular approach. While this routing work was being carried out, Testbed-15 was running in parallel. There was some overlap between both work efforts and coordination between these

efforts enabled a joined-up approach by sharing interim findings.

The *OGC API – Processes* draft specification adopts a resource-based approach, which is along the same lines as the OGC API – Features specification (formerly named WFS 3.0) except it is focused on processing services. The OGC API – Processes draft specification has not yet been approved as a standard and is some way from ratification, therefore implementations in this Pilot were experimental and contributed to the discourse on OGC APIs. A point of contention at time of writing was the structure and nature of OGC APIs for processing - the concept of OGC API is superseding WPS as OGC's processing API. There were broadly two schools of thought regarding OGC API architecture and design:

1. A very lightweight raw OpenAPI version of processes, for example in this Pilot "/routes/"
2. A somewhat lightweight version of OpenAPI that conforms broadly to WPS2.0 calls, for example "/processes/<processName>/jobs"

A convention of this ER is that these two options are referred to *Option 1* and *Option 2* throughout the text. This is a reflection of the naming conventions from the Pilot work. It is suggested that these options are named formally in a TC working group should the TC choose to standardize both approaches.

One argument for choosing Option 1 and essentially dropping the WPS construct is that unlike the other OGC services such as WFS, WCS and others, there is nothing inherently *geospatial* about WPS and it has always been a somewhat artificial construct. This is likely to be an on-going discussion and was not resolved by the end of the Pilot. Therefore, the clients and WPS implementations supported both options as a demonstration exercise.

6.2. Overview of Pilot architecture

The Pilot architecture comprised of two clients, three implementations of OGC API - Processes profiles, and three routing engines. These combine to make a variety of interoperable, end-to-end routing capabilities. All clients interface with all implementations of the OGC API, which in turn interface with all routing engines. Therefore, every client has the ability to select any combination of OGC API implementations and routing engines.

The supplied source datasets from OpenStreetMap (OSM), HERE and the US National System for Geospatial Intelligence (NSG) were converted to GeoPackages and used by the technical components as inputs into the routing engines. The WPS Routing API ER (a sister document to this ER) documents the technical discussions and findings relating to all API considerations, which is not covered in this scope of this ER. The Route Exchange Model Annex details the GeoJSON encoding used to transfer content between the technical components of the Pilot architecture. At this time, there was no conceptual model for route exchange, although such conceptual models could be created as part of a future pilot or as work within the working groups of the Technical Committee (TC).

The Pilot considered three specific use cases aimed at tackling real-life routing challenges. These three use cases assume the possibility of DDIL environments and the need to support a variety of route choices. This environment lends itself to three distinct scenarios, online, offline, or hybrid, where a client has connectivity to get capability and data, but it is lost over time.

Chapter 7. Pilot Technical Architecture

The Pilot architecture includes a variety of implementations all supporting the Routing API and the Routing Exchange Model. The Routing API is discussed in brief here, further detail can be found in the WPS Routing API ER [1]. In addition, the Routing Exchange Model is outlined, for further detail refer to Annex A Routing Exchange Model.

7.1. Scenarios

The Pilot architecture consisted of three scenarios influenced by the DDIL environment use cases. These were:

- Online - fully connected with stability
- Intermittent - unreliable connection.
- Offline - no connectivity

7.1.1. Online Scenario

In the Online Scenario, the operator uses one of the clients to request a route from any of the implementations of OGC API - Processes profiles. The chosen profile implementation then passes this request on to one of the routing engines using the routing exchange model. The engine is tasked to complete a route using a variety of parameters that are described later in this document.

Once computed by one of the engines the route is returned to the profile implementation using the exchange model. This profile implementation then passes the route back to the client to be visualized and shared by the operator.

The Online Scenario components must support the ability to choose which OGC API - Processes profile to use, which routing engine to use and which routing algorithm to use. In addition, the ability to allow extra criteria must be supported, such as maximum height of vehicle, obstructions or areas to avoid, and arrival or depart time. Finally, the choice of the fastest route or shortest route should also be available.

The online scenario is where the components in the Pilot have consistent connections between them and out to the wider internet.

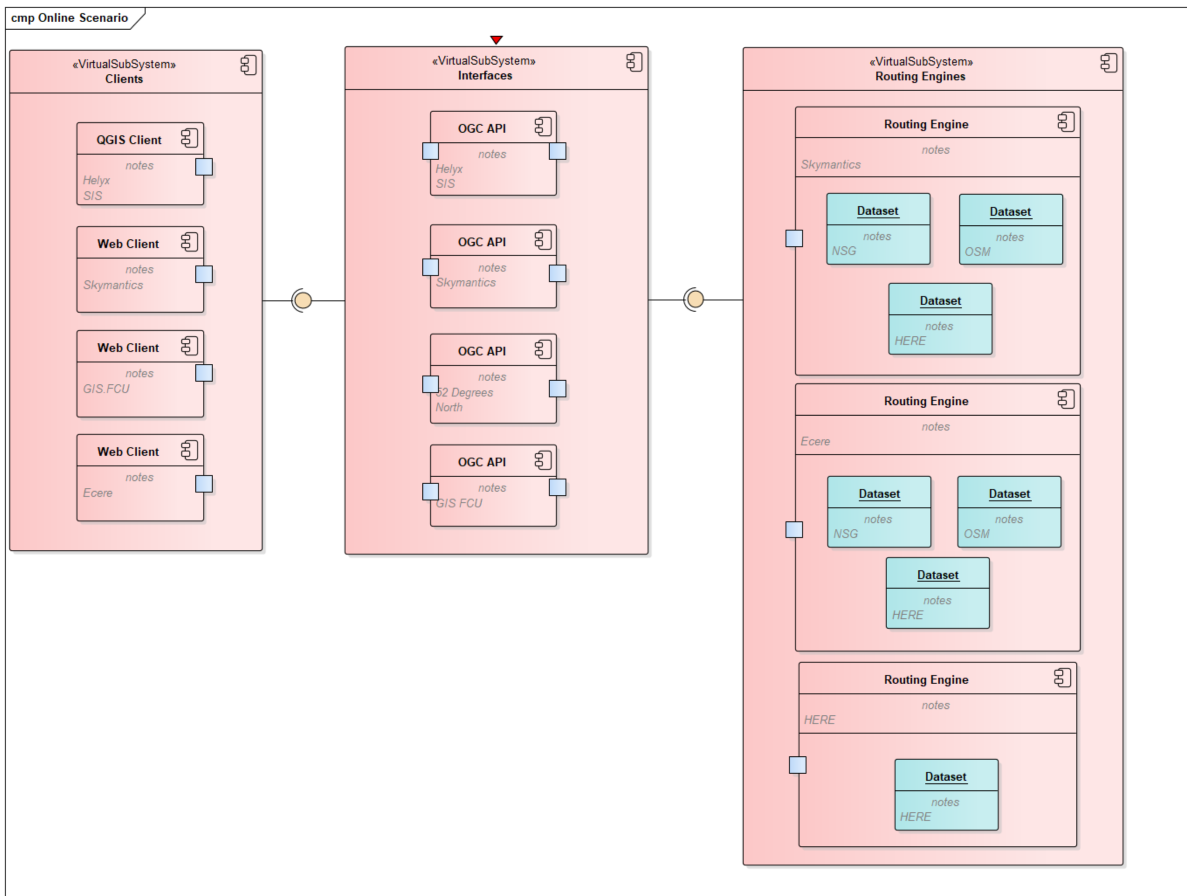

Figure 2. Online Scenario Architecture

Figure 2 describes the architecture in the online scenario. Essentially there are three virtual subsystems (that is, individual components not co-located performing the same function) that can communicate with all of the components within the upstream and downstream virtual subsystems. For example, the Ecere client was able to configure and execute the HERE routing engine via the Helyx OGC API.

This architecture uses the Routing API established as part of this pilot for all request and response handling between the client and OGC API Process Profiles.

7.1.2. Intermittent Scenario

The intermittent scenario is where the components have connectivity, but it is not necessarily:

- Consistent
- Stable
- Reliable
- High-speed

Therefore, the network cannot be relied upon to provide connectivity on demand and compensation actions are likely when connectivity is not available. Intermittent connectivity is unpredictable and it maybe that in the real-world, decisions are made to treat intermittent connectivity as no connectivity, as it is the only sensible course of action, especially if the scenario involves threat to life.

In the Routing Pilot API, the intermittent scenario was described as a situation where only one of the clients had access to a routing engine. Therefore, the connected client had the ability to create routes, but none of the other clients did. Additionally, there were situations where the clients are able to communicate with each other via some other means (Bluetooth or some other peer-to-peer communication, for example). This scenario could also be a situation where a client had connectivity to a routing engine, but has lost it due to other reasons.

The solution proposed to address intermittent connectivity is to enable the clients to share pre-defined routes, that is, the routing operation has been completed when a connection to the routing engine was established, but has now been lost. Route sharing between clients is facilitated by the JSON encoded *Route Exchange Model*, described in the annexes.

7.1.3. Offline Scenario

The third scenario assumed that there was no connectivity outside of a device's local network, this could be a desktop computer, mobile device or a mesh. In the real-world the scenario is modeling an instance where there is no connectivity and there is not going to be any connectivity for the duration of an operation. To mitigate the lack of connectivity, the routing engines, the WPS and the clients are packaged on the same machine. Therefore, the routing engine implementers made the routing engine available for installation on the client machine.

In the Offline scenario the operator uses the routing functionality provided by the client to create a route. The operator then shares this route with other clients using the route exchange model. To enable the required functionality, all of the capability has to be tightly coupled in a single location. Practically, this involves installing all of the components on the same machine to remove communication dependencies with the wider network.

7.2. Client Execution Patterns

There are several clients that execute the OGC APIs to task the routing engines. The description of each of these clients is covered in the relevant components section. This section describes the generic client-server communication patterns that are utilized by the clients, these were:

- Synchronous
- Asynchronous
- Callback
- Server-sent events (SSE)

7.2.1. Synchronous

The synchronous interaction pattern is typical of short-running processes. The client configures the routing engine parameters and sends the execute request, the client then holds the connection open to the server until it receives a response. This model is simple and efficient for small requests and responses, but it can cause time out exceptions and cause the client to hang until a response or time out is received.

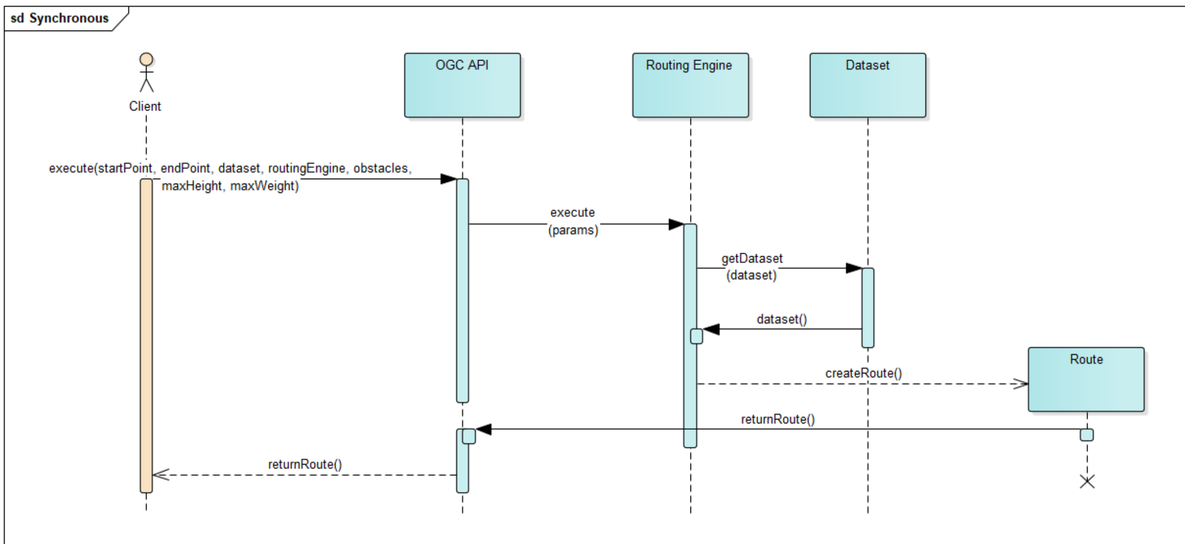

Figure 3. Synchronous Client execution pattern

7.2.2. Asynchronous Polling

Asynchronous execution is when the client sends an execution command to the server, but does not wait for a result. There are then several options for *checking* whether the result has been generated. In this client, the method of checking for a result is called *polling*, that is, checking back with the server at a defined time for a result. This is slightly more efficient that synchronous as it releases the thread post execution request. However, it can be inefficient as the client is constantly checking the server and the server has to respond each time, which can affect network and computational performance.

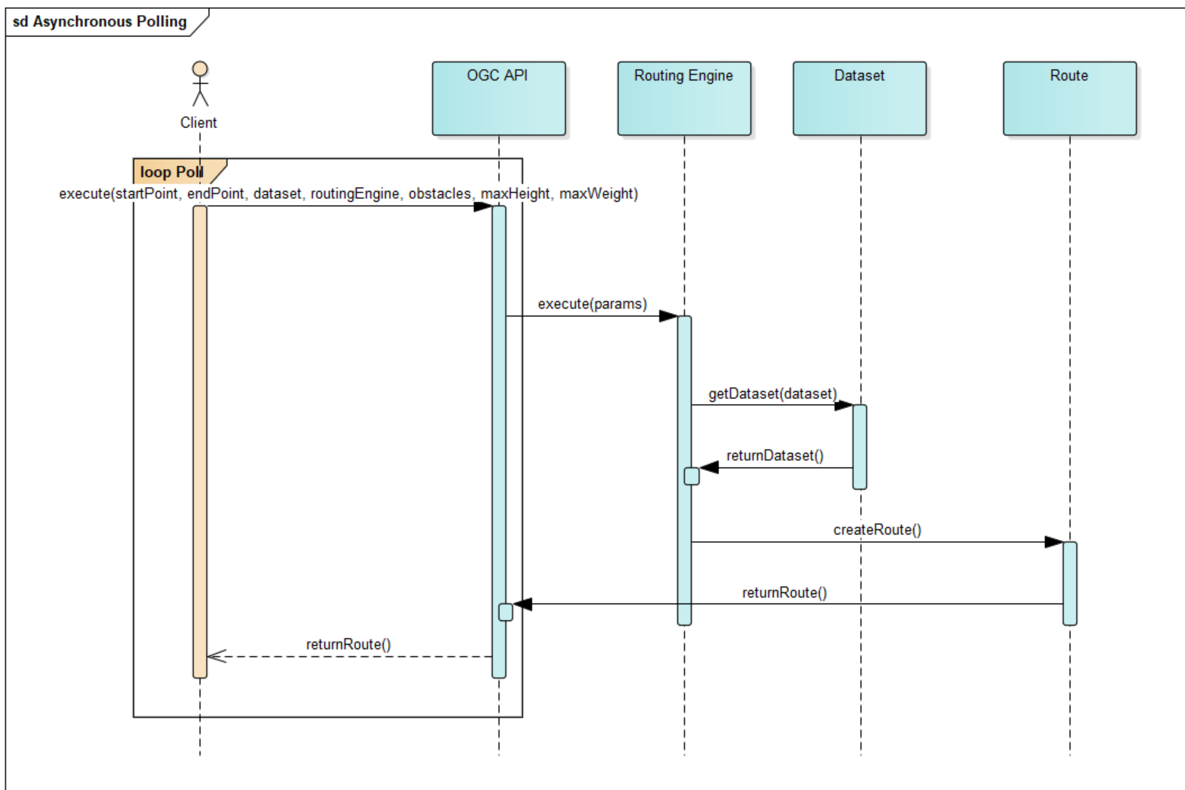

Figure 4. Asynchronous Polling Client execution pattern

7.2.3. Callback

Callback has a slightly different approach to asynchronous as the client sets up a miniserver in the background upon opening. This server stays active whilst the client is active and the client provides the miniserver address to the OGC API endpoint. The OGC API then uses a POST request to send the result to the miniserver. This has the advantage of removing the requirement to polling, but it does make the client a little more heavyweight as there is a second server involved. This approach is particularly relevant to many Desktop clients that rely on threading such as QGIS. The miniserver is started on a separate thread to stop machine lockup.

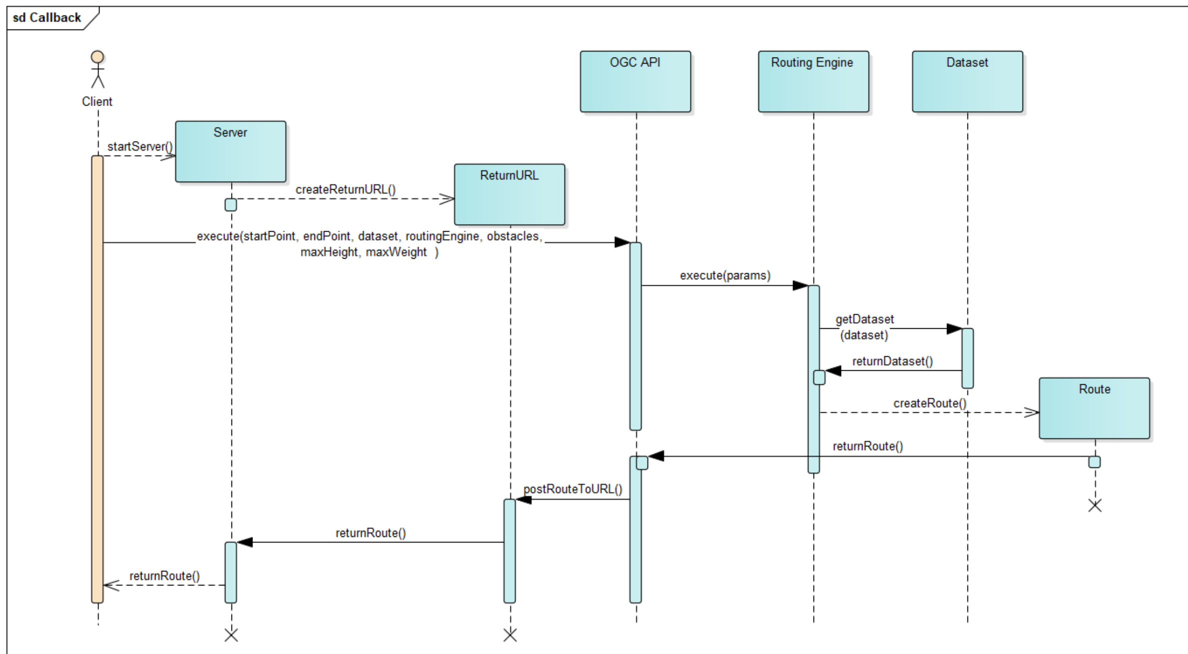

Figure 5. Callback execution pattern

7.2.4. Server-sent Events

SSE is a compromise between synchronous and asynchronous as it utilizes both approaches in its cycle. When the client creates a POST request to execute the routing engine through the OGC API, it also creates an SSE request. The routing engine produces the route and then sends a handshake to the client using the SSE endpoint, the data are then streamed to the client.

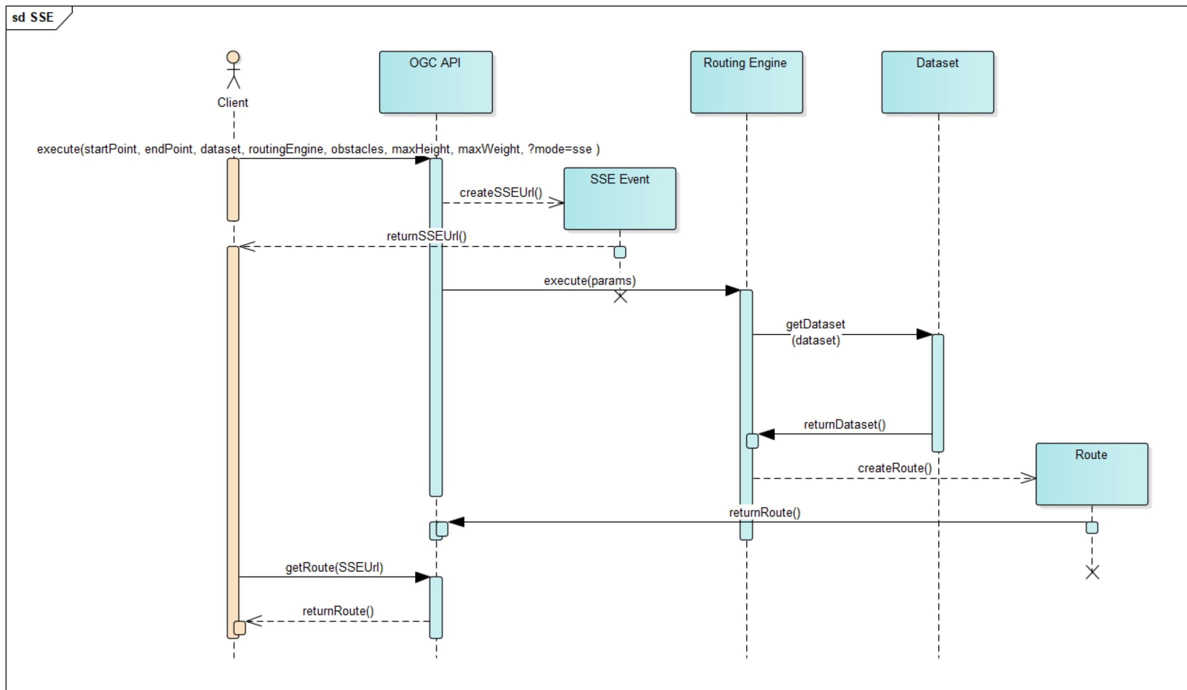

Figure 6. SSE execution pattern

7.3. API Pattern Options

The design of the API for Routing is discussed at length in the WPS Routing API ER, however, the design of the API has ramifications for design of servers, clients and the appropriate architecture pattern for the different options. The two options are broadly as follows:

1. Option 1 - Using *Routes* as a resource.
2. Option 2 - Including a WPS facade in the API path.

Option 1 is the most simplistic and does not require any artificial inclusion of prefixes in the paths. The OpenAPI standard has the vocabulary to define processes, and unlike WFS, which has a lot of geospatial specific guidance, WPS has always been designed to provide processing capability regardless of any geospatial constraints.

Option 2 respects the current WPS 2.0 path parameters and includes a /processes/ prefix to indicate it is a WPS. Including the WPS pattern as part of the OGC API for processing enables continuity between the old WPS and the new WPS, however it introduces complexity and rigidity.

A wider discussion regarding the API pattern design and its relevance to WPS is to understand where data requests end and processing begins. When a user requests a dataset using WFS and parameters are applied to the request, then some data processing has to take place to fulfill the request. For example, when a user requests a dataset subset bounded by coordinates, then the operation on the server goes beyond simply returning the dataset as is and processing takes place. However, this is not considered a processing service and the line between data return and processing services is not well defined. For the Pilot, anything that created a new resource was considered a *processing service*, however this is a mutable claim, as one could simply create a new resource with a dataset subset.

For the scope of this ER all APIs support at least one of these Options in order to allow for the

Scenarios to be tested.

For further information on the APIs used during this Pilot see the OGC Routing API ER [1].

7.4. Input Datasets

The input data used by the Routing Engines to produce the routes is key to the routing capability efforts. Not only does this determine the structure of the routes produced, but also how they are displayed by the clients using the Routing Exchange Model.

The three datasets used are listed below, with a brief description of each. These datasets are discussed in more detail in the Routing Engine implementation sections.

- NSG: The National System for Geospatial Intelligence (NSG) Entity Catalog (NEC) specifies an NSG-wide semantic model for geospatial data. This semantic model includes: feature information concepts with their allowed geometric representations and related constraints, attributes with their domain types, associations with their roles, and accompanying metadata. The NEC specifies the domain data model for feature-based geospatial intelligence that determines the common semantic content of the NSG despite varying physical realizations across DoD/IC systems (i.e., regardless of whether geospatial features are represented as an image, a multi-dimensional grid of values, or a set of one or more vector shapes). Derivative models like Topographic Data Store (TDS) and US Army's Ground-Warfighter Geospatial Data Model (GGDM) identify specific content of the NEC that shall be obligatory for geospatial intelligence producers using this specification, and specifies the conditions under which this geospatial intelligence shall be collected by producers for use in net-centric data exchange with other NSG participants. The data sample provided in this pilot is a provisional NEC-based Model for military focused routing. This dataset is a profiled approach to the NEC that extracts key pieces of information conducive to military focused routing which includes Military Load classification, Overhead Clearances. It consists 3 core components, Nodes, Edges, and Network with an auxiliary association table liking NAS features with the edges of the graph. Nodes model the origin and terminus of each relationship (Edge) and NAS entities are associated to create a contextual or subject based network for analysis. These relationships are modelled through the node/edge graph structure and can be associated with geophysical phenomena such transportation entities or abstract relationships such as political, social, and economic matters. For this pilot, the NAS transportation entities Road, Bridge, and Tunnel were associated to the edges and provide the cost components of the graph. The baseline network was abstracted from OpenStreetMap data of Washington, D.C. However, the attributes in this sample are assigned for Test and Evaluation use only and do not represent actual geophysical phenomena in some cases (e.g. LC1-4) are grossly inaccurate. This data is fabricated to provide a controlled set of information to generate specific routes. The area is:

  ```
  {"type":"Polygon","coordinates":[[[-77.1176857,38.7924115],[-
  77.1176857,38.9952444],[-76.8773984,38.9952444],[-76.8773984,38.7924115],[-
  77.1176857,38.7924115]]]}
  ```

- OSM: The OpenStreetMap (OSM) project is an initiative that provides user-generated street maps for the entire world. OSM follows a peer-production and peer-review model similar to

that adopted by Wikipedia. The project aims to create map data that is available for anyone to use or edit for free. As a crowd-sourcing initiative, OSM comes with varying levels of data quality and as such has been the subject of several studies reviewing the quality of its data.

- HERE: NAVSTREETS is a commercial data product by HERE that provides road network data that includes functional classifications of roads, and geographic representations of features such as airports, aircraft roads, cemeteries, golf courses, hospitals, military bases, parks, national monuments, pedestrian zones, shopping centers, sports complexes, undefined traffic areas, university/colleges, and woodlands. NAVSTREETS provides users with information such as access to expressway ramps, connectivity of roadways, one-way streets, turn restrictions, construction projects, and lane dividers.

Chapter 8. Pilot Component Implementations

This section describes the implementations from each of the vendors involved in the Pilot. It is split into *clients*, *WPSs* and *routing engines*.

8.1. Client Implementations

8.1.1. Helyx QGIS Client

8.1.1.1. Design Considerations

The client is a desktop based QGIS plugin to support network routing. The goal was to provide the ability to easily compute and display new routes, fetch existing routes and share routes with other clients in line with the sponsor requirements. These routes are converted to QGIS layers for display and to allow users to input them into native QGIS functions, the data exchange was performed using the Route Exchange Model and the exchange format was GeoPackage.

QGIS was also chosen due to its extensive plugin customization and by request of the sponsor. The majority of this plugin capability uses PyQT, a python module for developing interactive interfaces. Once the PyQT capabilities were evaluated it was clear that this combination of PyQT and QGIS would provide the necessary functionality to produce the desired interface.

The design of the UI is broken down into five QGIS windows. The initial window (see Figure 7) allows the user to choose an API landing page URL to utilize. In order to account for both Option 1 and Option 2 of the Routing API this window provides the ability to choose which option to use when requesting routes.

The initial window also allows the user to choose offline routing or to open the sharing window. The offline routing option supports the *Offline Routing Scenario* described in previous sections.

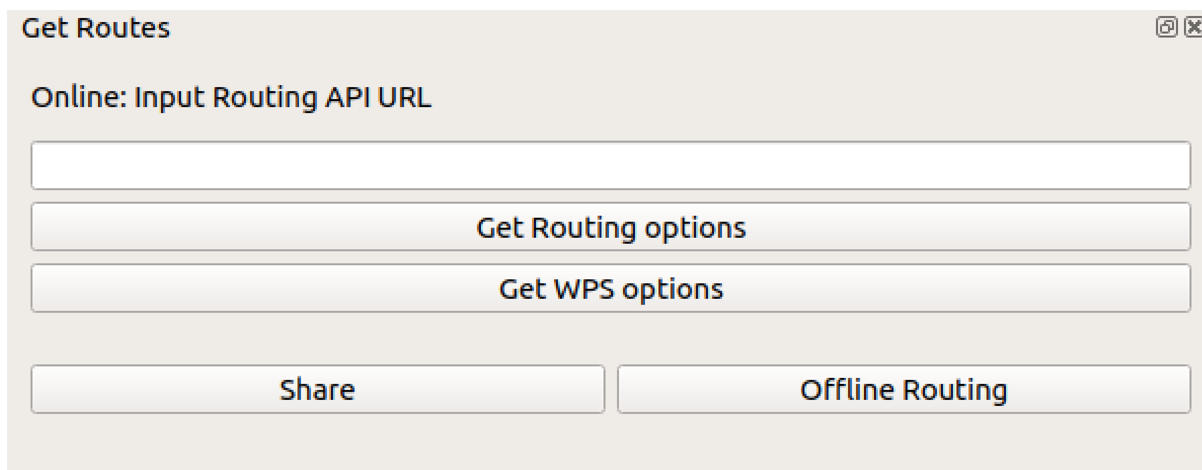

Figure 7. QGIS Plugin Opening Window

If Option 1 or Option 2 are chosen, then a corresponding window is displayed depending on the choice.

The Option 1 choice opens a comprehensive input window (see Figure 8) for the user to input route options and request a route.

Get Routes

WPS is: http://localhost:5600

Get New Route — Set the inputs below to create a new route using the API

Lat Lon

Start Point

End Point

Lat Lon

Waypoints

Delete Waypoint

Preference: Fastest

Source Dataset: HERE

Routing Engine: HERE-routing-engine

Routing Algorithm

Height Restriction

Max Load Restriction

Name of new Route

Time: No Time Constraint

00:00

Obstructions: No Layer Selected

Result Type: Full Route

FIND ROUTE poll

Get Existing Route — Choose Existing Route To Fetch

Refresh Route 5ags3

Share FETCH ROUTE

Figure 8. QGIS Plugin Option 1 Input Window

The Option 2 choice opens a window displaying all processes exposed at the API's processing endpoint (see Figure 9). The user then selects a process, sees the description and choose to continue with the chosen process.

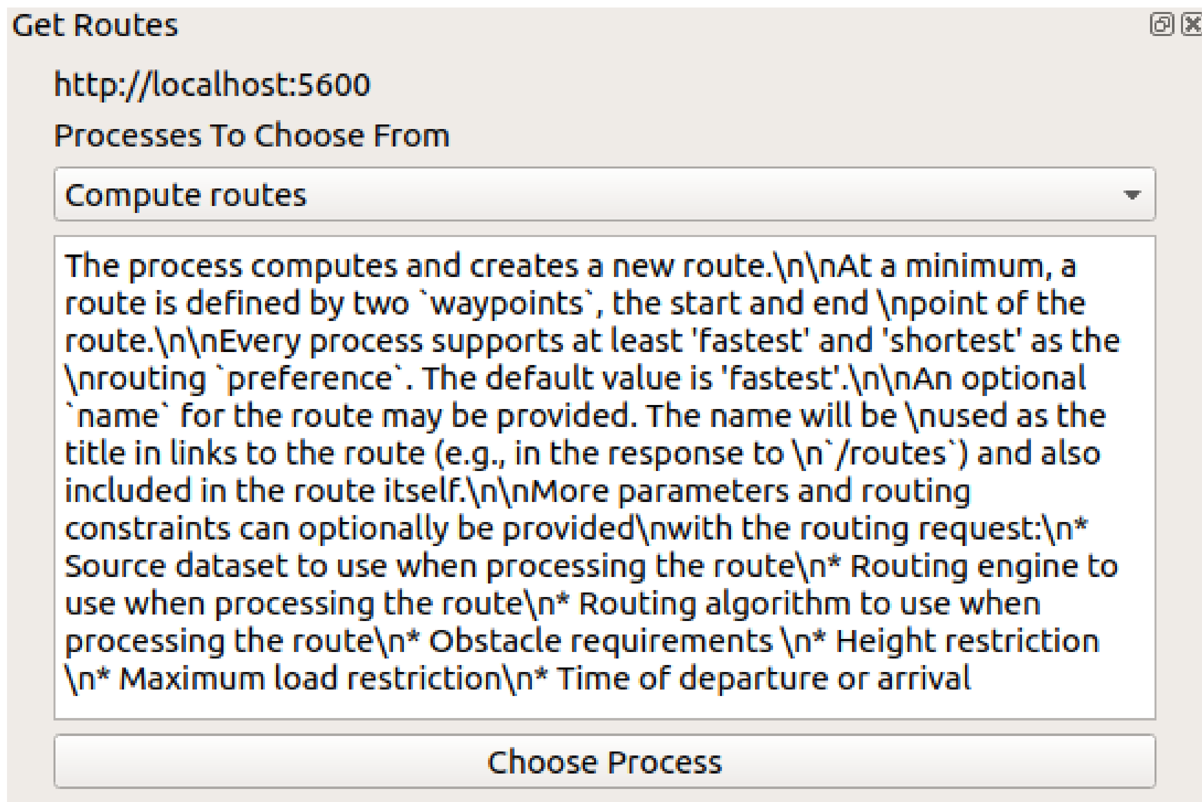

Figure 9. QGIS Plugin Option 2 Processes Window

The previous step opens a third window which is dynamically constructed to provide the input options and request options for the chosen process (see Figure 10).

Figure 10. QGIS Plugin Option 2 Input Window

Both the Opening window and the Option 1 Input window provide the ability to open the Sharing Window (see Figure 11). Once the sharing window is opened the user can select layers in the QGIS panel or GeoJSON files to share. The routes can be shared by sending them directly to other clients using the QGIS plugin or exported as a GeoPackage for offline sharing.

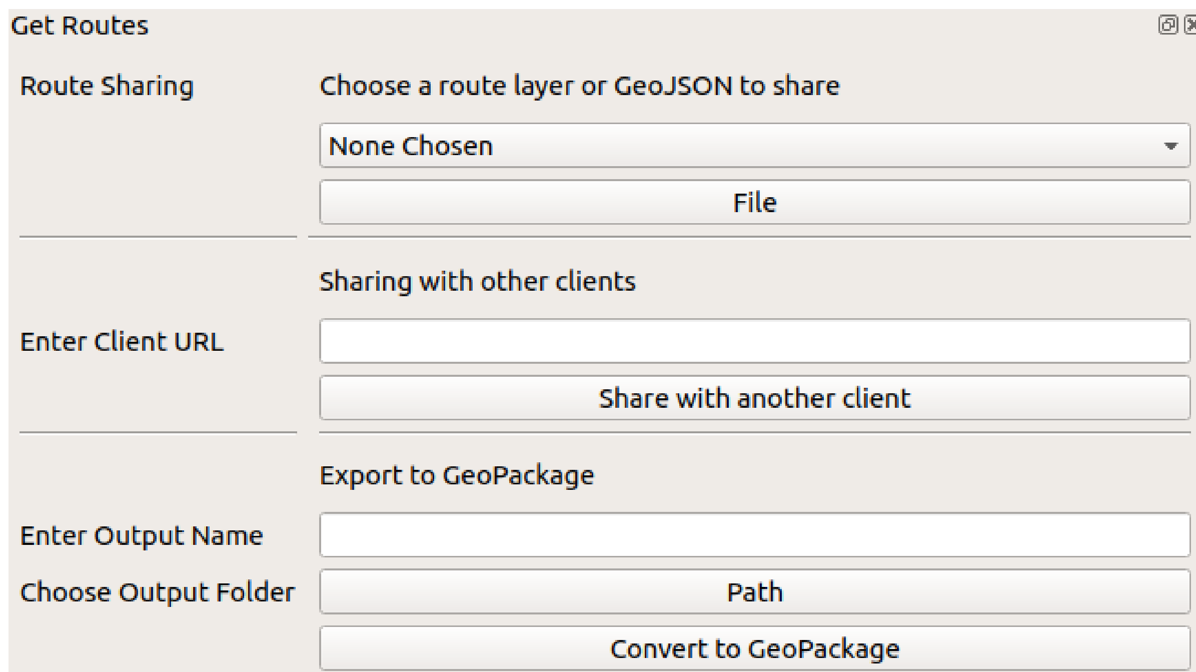

Figure 11. QGIS Plugin Sharing Window

This design provides the core functionality to create and visualize routes. It is expected that any extensions could easily be added by including an additional window, for example to create/edit routes manually.

In addition, the design ensures the functionality for Option 1 and Option 2 are distinct from one another, meaning the two capabilities could be de-coupled in the future if required.

8.1.1.2. Encountered Challenges and Solutions

The following challenges were encountered during the development of the client. Some of these challenges are implementation specific whilst others relate directly to the Routing Exchange Model and Routing API.

8.1.1.2.1. Multi-Threading to Support Asynchronous Requests

The Routing API exposes a number of endpoints to support asynchronous requests. These endpoints allow the client to request a route via a post request and later fetch the created route. The basic method for this asynchronous request is polling, where the client requests a route and uses the returned *Location* header to find the resource. This resource endpoint is then polled until a *finished* status is returned by the resource endpoint. In contrast the more efficient webhook (user defined HTTP callbacks) asynchronous method requires the client to provide a client-side *subscriber* URL in their route request. This URL is then used by the WPS to return the route once complete; the WPS sends the route data to the subscriber URL once the route is finished. This means only two requests are made, one Post request from the client to the server and one Post request from the server to the client. This is in contrast to polling where requests are made from the client to the server at regular intervals until the server reports a status change (or times out).

In order for QGIS to handle webhook functionality and expose a subscriber URL to the WPS, a server is required to run in the background of the application. This proved to be a challenge as QGIS runs on the main thread on the machine, meaning any other process, such as a server, running on the same thread causes QGIS to crash.

The solution to this challenge is to launch a separate thread at the point the Option 1 Input window opens in QGIS. The server is then started on this second thread and waits for any Post requests from APIs which have been sent subscriber URLs. The second thread is stopped when the QGIS plugin window is closed, meaning the cleanup for the thread has to be robust enough to allow the widget to be opened and closed repeatedly without conflicting threads being created.

8.1.1.2.2. Conformance classes and Dynamic GUIs

The Routing API provides a list of conformance classes via a conformance endpoint allowing clients to identify what functionality the API supports. Option 1 of the API exposes a route definition schema and Option 2 exposes the standard WPS process input definitions. Both of these approaches list all potential inputs to the route request. Out-of-Band information is required to know what functionality each conformance class relates to and therefore, which inputs are supported by the API. Given this conformance class and input definition dichotomy, two differing methods were devised for client creation. This is intended to show the benefits and constraints of the two API elements when creating clients.

The first approach focuses on supporting Option 1 of the API and relies on out of band knowledge of the conformance classes. If a conformance class exists, such as *obstruction*, then the client exposes input capability to send an obstruction as part of the routing request. The client cannot programmatically determine this input information from the conformance class, out-of-band knowledge is used to decide what sort of input to expose for each conformance class.

The second approach focuses on supporting Option 2 of the API and relies purely upon the input parameter definitions exposed by the process endpoint, which requires no out-of-band information. The client programmatically parses the input definitions and exposes appropriate inputs for the user.

The first approach exposes an *easy-to-use* client with obvious inputs and appropriate support for all request types, such as callback support and synchronous support. However, the first approach has proven to be inflexible when new conformance classes are introduced, as the client then requires an update to support this new functionality.

The second approach exposes a flexible dynamic front end which can account for additional inputs provided they conform to the Option 2 API schema. However, as the client parses the inputs in order, and displays them as they are parsed the order of the inputs, the resultant Graphical User Interface (GUI) can appear illogical to the user. Additionally, the API lists all possible inputs, not specifically the inputs supported by the various routing engines. Therefore, the user can send an input which is listed in the input list, such as an obstruction, which may not be supported by the routing engine. This makes for a quick-to-implement and flexible GUI interface useful for exploring conformant APIs, but which may supply unavailable options and therefore invalid requests.

The ideal approach is a combination of the two; conformance classes to define extensions to the API, where each class link returns a subset of JSON input definitions. The client creator can then decide, using out-of-band information, to order and define the display of each set of conformance class inputs. This allows for the dynamic creation of client elements, such as the input elements for the conformance class obstruction.

8.1.1.2.3. QGIS and Heterogeneous Feature Collections

The Routing Exchange Model uses the GeoJSON Feature Collection schema to return the start, end and segment nodes as well as the overview line, that is, the unsegmented route from start to finish. These are all stored as MultiPoint, Point and LineString types in a single Feature Collection. QGIS does not programmatically support Feature Collections containing more than one type of feature (sometimes called *complex features*, although there are plugins available to support GML application schemas).

Therefore, the client solution receives the single Feature Collection from the API and stores this original Feature Collection on file, to be shared later on. The contents of this JSON file is then copied and separated out into a Node Feature Collection JSON file and an Overview Feature Collection JSON file. These are then all loaded into the QGIS map project to be visualized. This satisfies the requirement of the client needing to load the Route Exchange Model GeoJSON data directly. When added to the QGIS Layer List the two layers are grouped inside a group layer. This group layer provides the mechanism by which to select an entire route when using the sharing window.

8.1.1.3. Final Component Description

The QGIS Plugin supports an assumed user workflow; initially the user opens the plugin initial window Figure 7 and decides whether they wish to interact with an API or conduct offline routing or sharing. The sharing button opens the sharing window, the *Offline Routing* button opens the *Offline Routing* window, which leverages the routing engines, including the Open Source Routing Machine and the Ecere Routing Engine. If the user wishes to use an online routing API then the landing page URL is specified in the text input box. Following these steps the user can either choose to use the Option 1 API and click the 'Get Routing options' button or the Option 2 API and click the *Get WPS options*.

Once the Option 1 *Get Routing Options* button is clicked the plugin conducts the following steps:

- The landing page is requested along with the routes endpoint to return any existing routes. If the landing page cannot be reached, then the window does not open, and an error is returned to the user.
- The conformance endpoint is checked. All conformance classes are returned, and corresponding functionality is flagged to be enabled in the input window based on which conformance classes are listed.
- A python server is launched on a separate machine thread to support callback requests (see Multi-Threading to Support Asynchronous Requests)
- The Input Window for Option 1 is displayed (see Figure 8), including populated drop down lists to reflect available inputs, and a list of existing routes which can be fetched from the server.

The user then chooses the *start* and *endpoint* of their desired route by clicking in the map view using the Start Point and End Point input check boxes. During the Pilot, no address geocoding was implemented as it was out of scope. From here the user can choose to accept the faults or they can alter the other inputs to reflect their requirements. In the case of the *obstruction* input, when the user clicks the Obstructions button the current selected layer in the QGIS layer list is validated and chosen for input.

The method by which a route is requested is available in the drop-down list adjacent to the FIND

ROUTE icon. This lists the supported options, based on the APIs conformance classes, which the user can choose to request a route. The default is polling Figure 4, other optional inputs include the API supported sync Figure 3 and callback Figure 5, as well as an additional method SSE (Server Send Events) Figure 6.

Once the user is content with their inputs they can click 'FIND ROUTE' to request the route via the proposed OGC Routing API. The plugin then conducts the following steps:

- The request is constructed, and serialized into JSON. Any polygon layers chosen for the Obstruction input are converted from QGIS layers to GeoJSON MultiPolygons.
- If the user chose the poll, sync or SSE method then the requested route is returned using the corresponding method.
- If the user chose the callback method for the request then the subscriber URL is included in the request, which is the URL of the python server on the second thread. This URL tells the API server where to send the route once it has finished computing. Once the route is computed the API then POSTs the route back to the subscriber URL. The python server receives the route and passes it back to the main thread.

When received, the plugin converts the route from a GeoJSON Feature Collection to QGIS Vector Layers (see QGIS and Heterogeneous Feature Collections), adds the Vector layers in a group to the layer list and displays them on the map.

If the user instead chooses to click the Option 2 *Get WPS options* button the plugin conducts the following steps:

- The landing page is requested along with the processes endpoint to return all existing process. If the landing page cannot be reached, then the window does not open and an error is returned to the user.
- The Processes window is displayed listing all available processes found at the processes endpoint along with a description for each (see Figure 9).

The user then chooses the 'Compute routes' process and clicks 'Choose Process'. The plugin then conducts the following steps:

- A GET request is sent to the *Compute Routes* process link, which returns back the Compute Routes process page, including a list of all supported inputs.
- For each input, an input class is constructed at runtime for the QGIS GUI and a request property is established in a request object.
- Once all inputs are processed all the input classes are exposed to the user in the Option 2 Input window (see Figure 10).

The user then chooses the start and endpoint of their desired route by clicking in the map view using the way points checkbox. If the API supports other inputs the corresponding input elements can be changed by the user in the GUI to reflect their requirements.

Once the user is content with their inputs they can click 'POST to Process' to request the route from the OGC Routing API. The plugin then conducts the following steps:

- The inputs are stored in their corresponding request property in the request object, which was created in when input window was initialized.
- The request object is then converted to JSON and sent using HTTP POST to the API server to request a route.
- The requested route is returned using the polling method.

When received, the plugin converts the route from a GeoJSON Feature Collection to QGIS Vector Layers (see QGIS and Heterogeneous Feature Collections), adds the Vector layers in a group to the layer list and displays them on the map.

The display of routes is customizable. At present the route and node features reference two styles on disk which the user can choose to update or change. Below in Figure 12 can be seen the full QGIS application, with the plugin loaded. In this image the route used as part of the TIE tests can be seen between the National Cathedral and the Washington Monument.

The corresponding inputs used to create the route can be seen in the plugin input box on the right hand side. The Nodes are represented by red points and the edges by red lines.

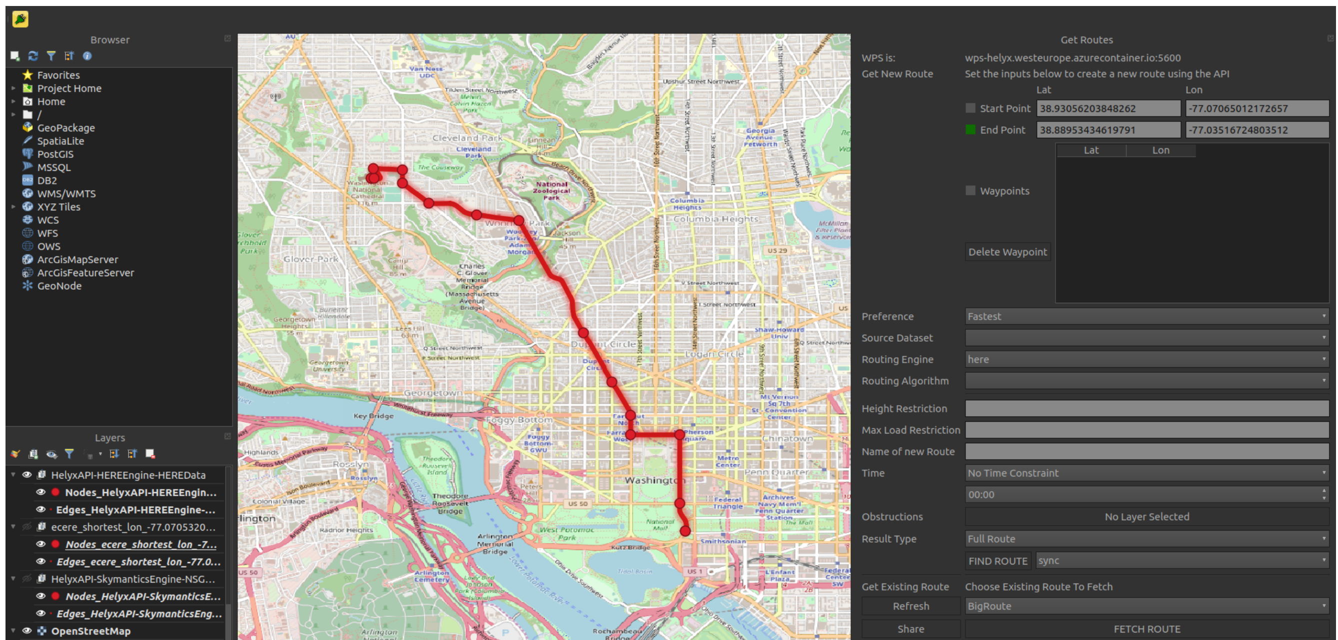

Figure 12. QGIS Plugin Test Route

When using either the HERE routing Engine or the Offline OSRM engine, routes can be made outside of the District of Columbia as they are not constrained by the pilot test data. See Figure 13

Figure 13. QGIS Plugin Sharing Window

The QGIS client plugin provides the functionality to interact with OGC APIs which support the proposed Routing API definition (both Option 1 and 2). The interface for Option 1 is detailed, allowing the user to tweak inputs, import obstructions from the native Layer List and retrieve existing routes from the Routing API. This is in contrast to the Option 2 interface, which is paired back and contains no cosmetic alteration. The Option 2 interface is built purely from an OGC WPS input list exposed by the Option 2 API.

In the case a connection cannot be established with the OGC API - Processes interface. The user also has the option to choose Offline Routing from the initial window. Once that option is selected a simple window opens allowing the user to choose between either OSRM (Open Source Routing Machine) engine or the Ecere routing engine. In addition, the user chooses their way points and a name for the route. Once the 'FIND ROUTE' button is clicked the route is calculated on the local machine and returned to the client that then adds the route to the map.

Future improvements include:

- The display of route instructions in QGIS popups or map tips.
- The inclusion of a tracking processing service to allow for the monitoring of vehicles along routes. This would be complimented by QGIS's feature creation capabilities to construct geofences for alerting and new obstacles during transit.
- The manual creation of routes in QGIS and the subsequent capability to validate and upload routes to either the OGC Routing API or the OGC API Collections endpoint.

In addition to the QGIS client, a basic Cesium client was created. The QGIS client is designed to support most input types, handle all APIs and Engines and provide the required functionality for a route manager or dissemination function. In contrast the Cesium viewer supports only the bare minimum functionality and is intended to allow users to view routes that conform to the Routing Exchange Model, alongside 3D data. In this fashion the Cesium client compliments the QGIS client which cannot be used to visualize the routes in a 3D environment. The Cesium client uses the

Cesium-provided Manhattan 3D Tiles Sample to demonstrate the route visualization through the center of Manhattan using the OSRM engine. 3D Tiles, based on the gITF transmission format, is an OGC Community Standard.

Figure 14. Cesium Cross Manhattan Route

Figure 15. Cesium Small Manhattan Route

8.1.2. GIS-FCU Client

8.1.2.1. Design Considerations

This client is designed as a web GUI to prove the concepts of the proposed Routing API and the underlying proposed Routing Exchange Model. This web client is also implemented as an embedded desktop application to ensure the ability to demonstrate offline routing scenario.

8.1.2.2. Encountered Challenges and Solutions

The main challenges encountered and their solutions discussed here were describing the conformance class and the dynamic GUI generation:

The Routing API provides the endpoint of the "/conformance" path to support discovery of the capabilities of the endpoint. The descriptions of the conformance classes reflect the parameter settings on the client GUI. To make this automatic generation work, the parser of the conformance classes on the client is required.

- Option Routes and option WPS

The Routing API provides different approaches of endpoint implementations. The client must also provide two different workflows and parsers to invoke the endpoints and digest the results. The Routes option is a more intuitive modern web design for resource access while the WPS option requires more basic and in advance knowledge about understanding the WPS to deal with the input, output and process elements. The WPS option supports the asynchronous execution (async)

mechanism natively in the WPS implementation while the Routes option requires a developer implement an asynchronous mechanism like *callback* in advance. Both the Routes and WPS options use Routing Exchange Model, and the client can reuse this model parser.

- Synchronous and Asynchronous

The Routing API supports synchronous and asynchronous execution. The client must design different workflow to handle job waiting (WPS option) or callback (Routes option) of the async approach.

- Design of offline application

To provide the offline scenario, the client was also designed to run offline. The approach of this client was to use a C# WebBrowser component to embed the client as a desktop application.

- Offline base map

When the client runs as a desktop application, the base map switches to the local OSM tiled vector data for better performance.

- Offline routing engine

When losing the internet connection, the client relies on the local routing engine to compute the routes. The local routing engine is an OS dependent library or command line program.

8.1.2.3. Final Component Description

The technical details are described in the following section.

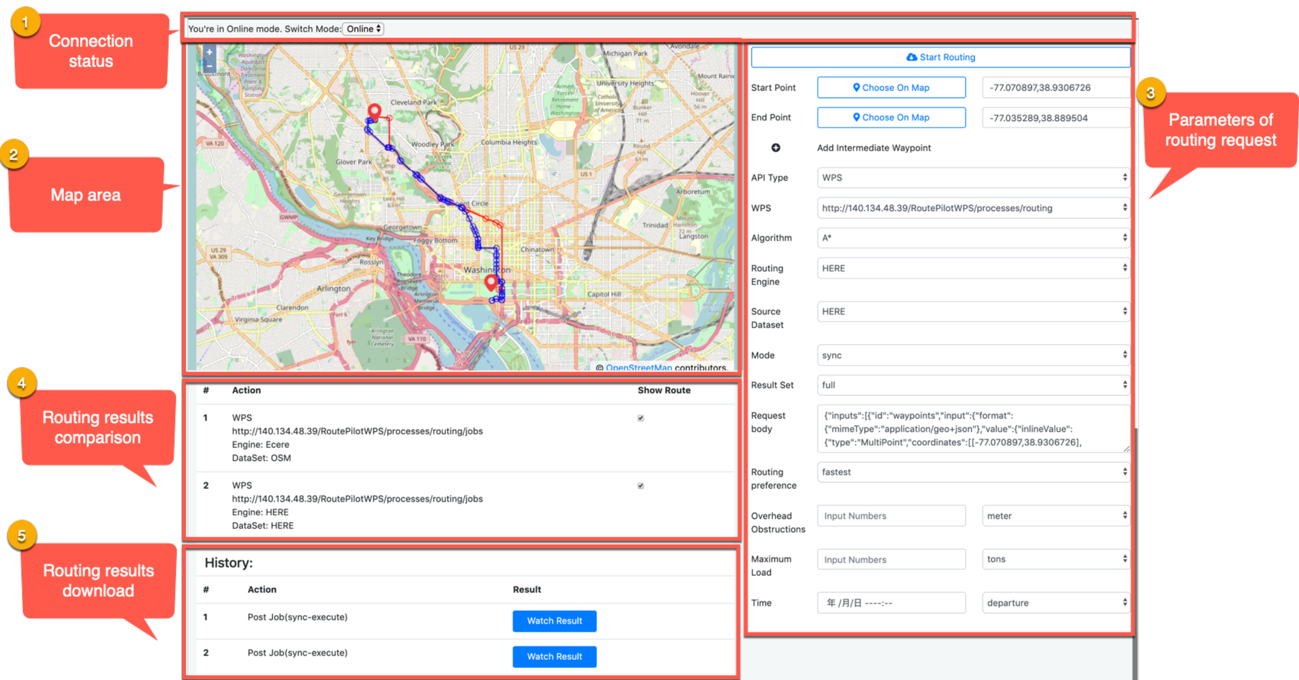

Figure 16. GIS.FCU Client UI

- Connection status
    - The connection status area indicates whether the client is in Online or Offline mode.

- The offline mode is an important feature in this pilot due to its application to DDIL environments. The key points of the implementation are the offline maps and offline routing Engines. In case of a limited client computing environment, the approach of vector tiles is recommended for better performance consideration. The offline routing engine is an OS-dependent library or command, it is recommended the implementation of a routing engine should also consider the OS environment.
- When in Online mode, the client uses the online OSM as a base map; when in Offline mode, it uses local OSM vector tiles.
- When in Offline mode, the client ignores any online WPS endpoints and use the local Routing Engine instead.
- The overview of the offline mode is illustrated as shown in Figure 17.

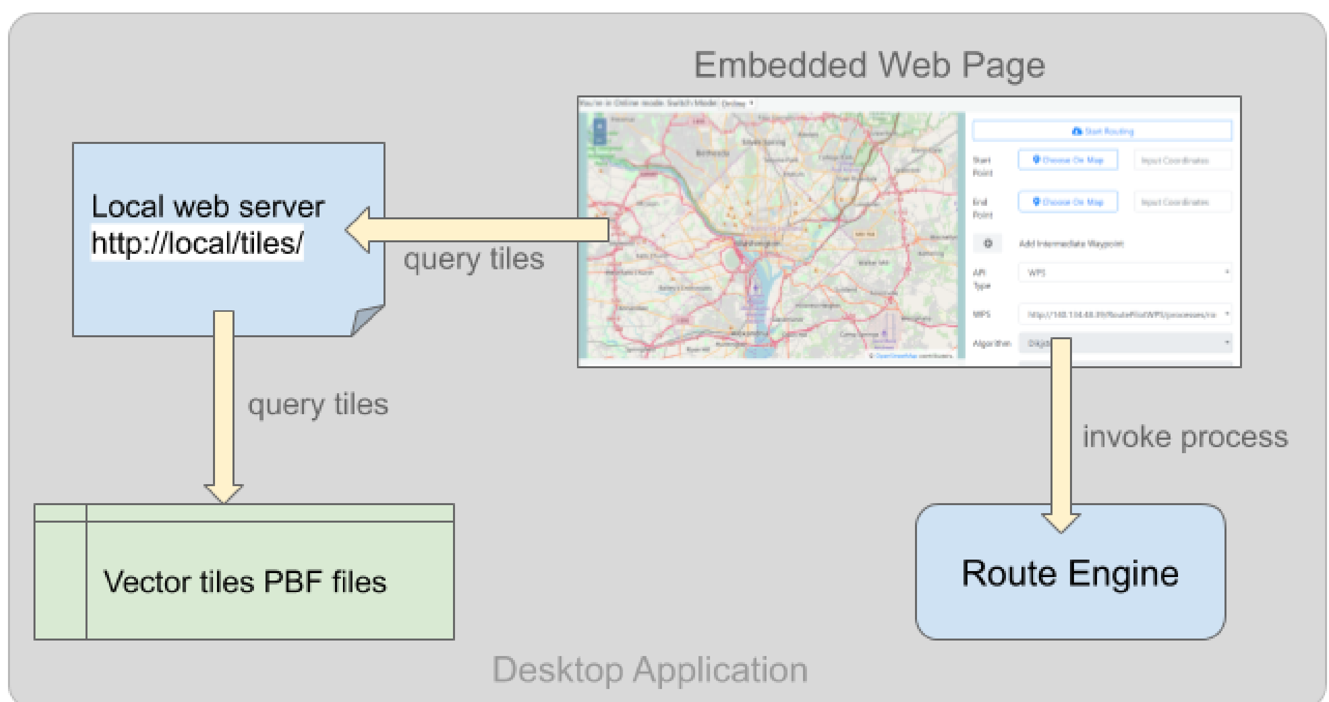

Figure 17. GIS.FCU offline mode overview

- Map area
  - The map implementation is based on Openlayers.
- Parameters of routing request
  - This section renders the parameters dynamically based on the conformance class of the API endpoint. The conformance class from different endpoints might be customized with different contents, this might cause this dynamic rendering fail.
- Routing results comparison
  - The results can be compared on the Map area by checking the checkbox.
- Routing results download
  - The results can be accessed in this area.

8.1.3. Ecere Client

8.1.3.1. Design Considerations

Ecere built a cross-platform mapping and routing client application featuring a 3D view. This was done leveraging the Ecere GNOSIS geospatial software technology [http://ecere.ca/gnosis].

A simple graphical user interface was developed allowing the selection of the routing points, the components, the algorithms and the parameters to use for the route calculation.

The client was used in multiple Technology Integration Experiments (TIEs), including both online and offline scenarios. This was done to support connecting to multiple open routing API services to perform route calculations, using the proposed Route Exchange Model.

The resulting routes were visualized in the client, and situated in a geospatial context with a number of available data layers. This included 3D content such as digital terrain elevation models and 3D buildings, background imagery, as well as styled and dynamically labeled OpenStreetMap vector data.

Figure 18. Ecere 3D routing client, showing route from National Cathedral to Washington Monument (Google Maps, OSM, Open Data D.C.)

8.1.3.2. Encountered Challenges and Solutions

The experiments involved components from different participants, combining the client capabilities with a variety of processing services endpoints implementing the Open Routing API (Ecere, Skymantics, Helyx, 52° North and GIS-FCU), three different routing engines used for computing the routes (Ecere, Skymantics and HERE), and different source data for the roads network used by those calculations (OpenStreetMap, HERE and NSG).

The large number of combinations tested, together with the new and evolving nature of both the Open Routing API and the components implementation, as well as the compact schedule of the

pilot, posed many challenges. This however, resulted in improvements to both the API description and the components implementation.

8.1.3.3. Final Component Description

The simplest scenario is the fully disconnected use case, where the client application embeds the Ecere routing engine and OpenStreetMap roads network data and makes use of them internally. As the client can also visualize compact optimized maps offline, this approach works in all DDIL environments, as both the roads network and maps to display was preloaded.

The client can also connect to online Open Routing API endpoints which will perform the route calculation. The communication mechanism between the routing client and the routing endpoint is defined by the Open Routing API specifications designed during the pilot. The client sends out an HTTP POST request embedding a JSON object with the parameters of the route calculation to perform.

The data visualized in the client came from multiple sources and formats:

- OpenStreetMap [https://www.openstreetmap.org] vector data (the roads features are also used for building the roads network graph for the routing engine)
- 3D buildings generated from the OpenStreetMap buildings based on the Simple 3D Buildings model [https://wiki.openstreetmap.org/wiki/Simple_3D_buildings], encoded as E3D [http://ecere.com/E3D.pdf] (for Kyushu island in Japan)
- Google Maps for global background high resolution satellite imagery
- Blue Marble (Next Generation) [https://earthobservatory.nasa.gov/features/BlueMarble] for global background low resolution satellite imagery (Reto Stöckli, NASA Earth Observatory).
- View Finder Panoramas [http://www.viewfinderpanoramas.org/Coverage%20map%20viewfinderpanoramas_org3.htm] for world-wide 3" elevation derived partially from SRTM, by Jonathan de Ferranti
- 3D Buildings with Detailed Roof in Washington, D.C. from OpenData D.C. [https://opendata.dc.gov/datasets/274f7c2b5f7c4ae19f165d9951057a00], transformed into CityGML by GIS-FCU in Testbed 14 - CityGML and AR initiative (OGC document #18-025 [http://docs.opengeospatial.org/per/18-025.html]), and then converted into E3D [http://ecere.com/E3D.pdf] for this project.

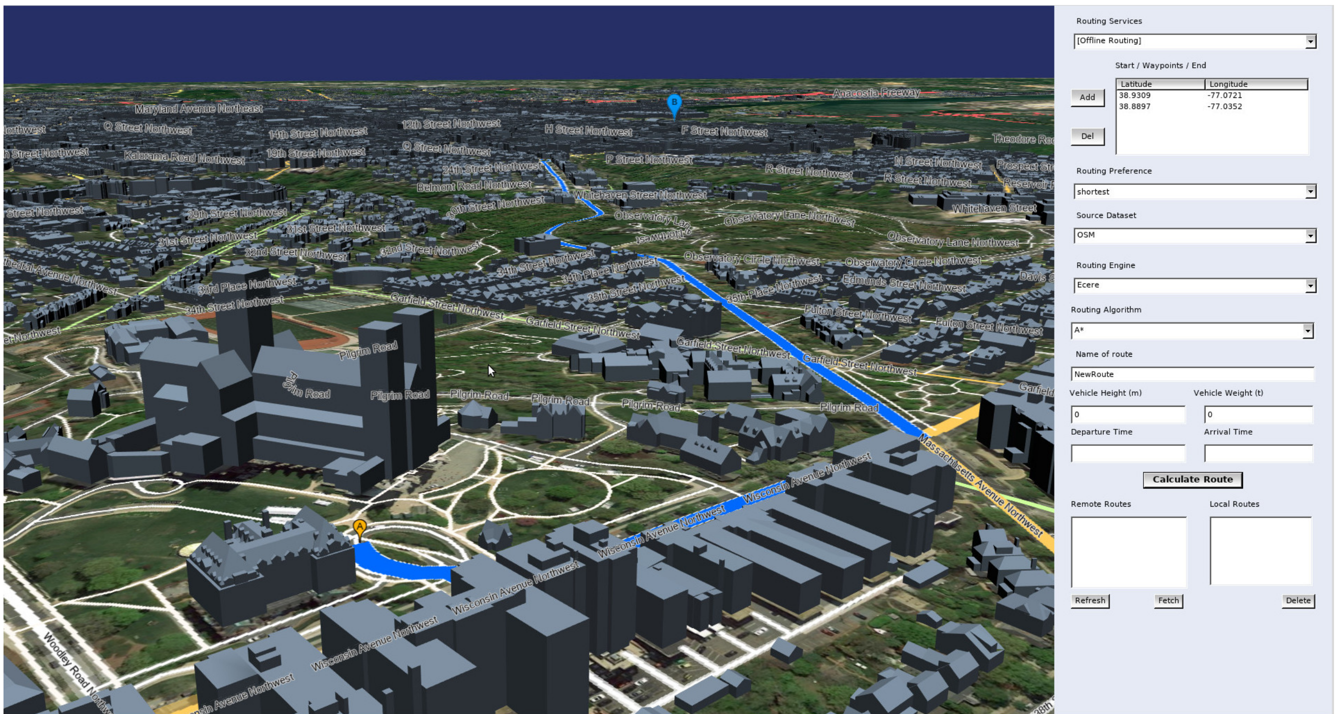

Figure 19. Ecere 3D routing client, showing area around National Cathedral (Google Maps, OSM, Open Data D.C.)

Figure 20. Ecere 3D routing client, showing area around Washington Monument (Google Maps, OSM, Open Data D.C.)

8.2. OGC API - Processes Profile Implementations (WPSs)

8.2.1. 52North WPS

8.2.1.1. Design Considerations

While conformance classes are useful for communicating the capabilities of services, their expressiveness is not enough to convey the information needed in the use cases described in this Pilot. The architecture of a distinct service per engine (see Final Component Description) was ultimately chosen to compensate for the difference in supported inputs and parameter values across engine implementations.

For Example, the HERE engine does not support the selection of the source dataset but is fixed to the usage of the HERE data, while the Skymantics engine is able to operate on the NSG, HERE and OSM datasets. If a service offers both engines via the 'engine' parameter it has to decide if the 'http://www.opengis.net/orapip/routing/1.0/conf/source-dataset' conformance class should be listed or not. If the client decides to conform, the route calculation will fail, either silently or with an error message, if a client selects the HERE engine and the OSM dataset. If it chooses to only list conformance classes that are valid for all engines, a client cannot assume that the parameter is supported by any of the engines without out-of-band information.

By having one API endpoint for each engine, and by such one conformance class declaration for each engine, clients can refrain from using out-of-band information or issuing possibly invalid requests, but can solely rely on the information the service offers.

For the most part, this only concerns the simplified Routing API and not the OGC API - Processes option, as the conformance classes are solely created from the process descriptions. As these only list the supported inputs and allowed parameter values, clients have all the information they need. One exception is the 'http://www.opengis.net/orapip/routing/1.0/conf/intermediate-waypoints' conformance class, which cannot be expressed using the standardized process description, as there is no way to describe how many points are required or allowed inside a GeoJSON 'MultiPoint'. One solution to overcome this would be to split the 'waypoints' parameter into three distinct parameters as it is done for the internal processes.

All routing engines support the 'core', 'callback', 'delete-route' and 'sync-mode' conformance classes as these are not engine specific but implemented within the proxy.

Example 1. Conformance classes of the Ecere routing engine

'GET https://testbed.dev.52north.org/orp/ecere/conformance HTTP/1.1'


```
{
  "conformsTo": [
    "http://www.opengis.net/spec/ogcapi-processes-1/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/callback",
    "http://www.opengis.net/orapip/routing/1.0/conf/delete-route",
    "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode"
  ],
  "http://www.opengis.net/orapip/routing/1.0/conf/core": {
    "values": ["fastest", "shortest"]
  }
}
```


The Conformance classes of the HERE routing engine additionally list the optional 'max-height', 'max-weight', 'obstacles', 'time' and 'intermediate-waypoints' conformance classes as these are supported by the HERE engine.

Example 2. Conformance classes of the HERE routing engine

'GET https://testbed.dev.52north.org/orp/here/conformance HTTP/1.1'


```
{
  "conformsTo": [
    "http://www.opengis.net/spec/ogcapi-processes-1/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/callback",
    "http://www.opengis.net/orapip/routing/1.0/conf/delete-route",
    "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode",
    "http://www.opengis.net/orapip/routing/1.0/conf/max-height",
    "http://www.opengis.net/orapip/routing/1.0/conf/max-weight",
    "http://www.opengis.net/orapip/routing/1.0/conf/obstacles",
    "http://www.opengis.net/orapip/routing/1.0/conf/time",
    "http://www.opengis.net/orapip/routing/1.0/conf/intermediate-waypoints"
  ],
  "http://www.opengis.net/orapip/routing/1.0/conf/core": {
    "values": ["fastest", "shortest"]
  }
}
```


The Conformance classes of the Skymantics routing engine additionally lists the optional 'routing-algorithm' and 'source-dataset' conformance classes as these are supported by the Skymantics engine. The supported values of these parameters are generated from the input description of the

process.

Example 3. Conformance classes of the Skymantics routing engine

'GET https://testbed.dev.52north.org/orp/skymantics/conformance HTTP/1.1'


```
{
  "conformsTo": [
    "http://www.opengis.net/spec/ogcapi-processes-1/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/callback",
    "http://www.opengis.net/orapip/routing/1.0/conf/delete-route",
    "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode",
    "http://www.opengis.net/orapip/routing/1.0/conf/routing-algorithm",
    "http://www.opengis.net/orapip/routing/1.0/conf/source-dataset"
  ],
  "http://www.opengis.net/orapip/routing/1.0/conf/routing-algorithm": {
    "values": [
      "astar",
      "astar-bi",
      "astar-ch",
      "dijkstra",
      "dijkstra-bi",
      "dijkstra-ch"
    ]
  },
  "http://www.opengis.net/orapip/routing/1.0/conf/source-dataset": {
    "values": ["HERE", "NSG", "OSM"]
  },
  "http://www.opengis.net/orapip/routing/1.0/conf/core": {
    "values": ["fastest", "shortest"]
  }
}
```


If all three engines were combined in a single API instance (see Potential conformance classes of a single API instance), the service would have to list the conformance classes of all engines. With this approach the client has no way to determine if a parameter is really supported by the chosen engine. According to the conformance classes, the client is allowed to issue a request using the HERE engine and the OSM source dataset. As this specific combination is not supported the API would have to return an error or fail silently.

Example 4. Potential conformance classes of a single API instance


```
{
  "conformsTo": [
    "http://www.opengis.net/spec/ogcapi-processes-1/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/core",
    "http://www.opengis.net/orapip/routing/1.0/conf/callback",
    "http://www.opengis.net/orapip/routing/1.0/conf/delete-route",
    "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode",
    "http://www.opengis.net/orapip/routing/1.0/conf/engine",
    "http://www.opengis.net/orapip/routing/1.0/conf/routing-algorithm",
    "http://www.opengis.net/orapip/routing/1.0/conf/source-dataset",
    "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode",
    "http://www.opengis.net/orapip/routing/1.0/conf/max-height",
    "http://www.opengis.net/orapip/routing/1.0/conf/max-weight",
    "http://www.opengis.net/orapip/routing/1.0/conf/obstacles",
    "http://www.opengis.net/orapip/routing/1.0/conf/time",
    "http://www.opengis.net/orapip/routing/1.0/conf/intermediate-waypoints"
  ],
  "http://www.opengis.net/orapip/routing/1.0/conf/routing-algorithm": {
    "values": [
      "astar",
      "astar-bi",
      "astar-ch",
      "dijkstra",
      "dijkstra-bi",
      "dijkstra-ch"
    ]
  },
  "http://www.opengis.net/orapip/routing/1.0/conf/source-dataset": {
    "values": ["HERE", "NSG", "OSM"]
  },
  "http://www.opengis.net/orapip/routing/1.0/conf/core": {
    "values": ["fastest", "shortest"]
  },
  "http://www.opengis.net/orapip/routing/1.0/conf/engine": {
    "values": ["here", "skymantics", "ecere"]
  }
}
```


8.2.1.2. Encountered Challenges and Solutions

During the pilot the following shortcomings in the current draft of the OGC API - Processes were identified.

There are two ways of defining an allowed range for a literal input of a process. The range can be defined using the 'allowedRanges' or using the 'allowedValues' in the 'valueDefinition' of a 'literalDataDomain'. While the former approach supports only ranges, the latter also supports the

definition of values (see Redundancy in the OGC API for Processes description language. Both, 'allowedRanges' and 'allowedValues' allows the definition of ranges). The Pilot participants recommend that the 'allowedRanges' approach be removed from the draft specification.

Example 5. Redundancy in the OGC API for Processes description language. Both, 'allowedRanges' and 'allowedValues' allows the definition of ranges

```
{
  "process": {
    "inputs": [
      {
        "id": "input",
        "input": {
          "literalDataDomains": [
            {
              "valueDefinition": [
                {
                  "minimumValue": "0",
                  "maximumValue": "10",
                  "rangeClosure": "closed"
                }
              ]
            },
            {
              "valueDefinition": {
                "allowedRanges": [
                  {
                    "minimumValue": "0",
                    "maximumValue": "10",
                    "rangeClosure": "closed"
                  }
                ]
              }
            }
          ]
        },
        "minOccurs": 0,
        "maxOccurs": 1
      }
    ]
  }
}
```

There is a risk of redundancy when specifying an input that can occur an unlimited number of times. Table 16 of the WPS 2.0.2 standard (14-065r2) indicates that the default value for both the minOccurs and the maxOccurs attributes is 1. In contrast, the draft OGC API – Processes specification does not explicitly state the default values of these attributes. This means that there could be a risk of omission of the 'maxOccurs' property being interpreted as 'maxOccurs' set to

unbounded (see Incorrect interpretation of 'maxOccurs' omission and 'maxOccurs=unbounded' as meaning semantic equivalence). The former approach has the benefit of using just one type for the 'maxOccurs' key instead of allowing a 'string' or an 'integer, and is therefore recommended. It is recommended that the editors of the OGC API – Processes specification clarify that the default value for both the minOccurs and the maxOccurs attributes is 1.

Example 6. Incorrect interpretation of 'maxOccurs' omission and 'maxOccurs=unbounded' as meaning semantic equivalence

```
{
  "process": {
    "inputs": [
      {
        "id": "input1",
        "input": {
          "literalDataDomains": [...]
        },
        "minOccurs": 0
      },
      {
        "id": "input2",
        "input": {
          "literalDataDomains": [...]
        },
        "minOccurs": 0,
        "maxOccurs": "unbounded"
      }
    ]
  }
}
```

Many JSON libraries are able to automatically parse JSON objects into language specific structures or objects. One example of this is the popular Jackson library for Java, which can convert between JSON and annotated Java classes. The current draft version of the OGC API - Processes specification makes extensive usage of the 'oneOf' keyword of JSON Schema. This allows for the definition of multiple alternative schemas for the same property or object. While there is support for this kind of approach, libraries most often rely on discriminator values to determine the actual type of the object.

For example, the 'input' property of an inputs element can contain three different objects (see Different input types of the OGC API for processes).

The concrete type can only be determined by the existence of distinct property names (i.e. 'literalDataDomains', 'supportedFormats' and 'supportedCRS'). This approach is unsupported for most of the popular frameworks and requires manual type conversion or custom parsers. The same applies to the 'valueDefinition' of 'literalDataDomains' where the concrete type is determined by the JSON type (i.e. a JSON array denotes an 'allowedValues', a JSON object with the 'allowedRanges' property denotes an 'allowedRanges', any other JSON object denotes an 'anyValue' and a JSON string

denotes a 'valuesReference').

Further, the semantic meaning of specifying '"valueDefinition": { "any": "false" }' is unclear.

Thus, adding discriminator values to the relevant entities of the specification is highly recommended to add discriminator values to the relevant entities of the specification (see Different input types of the OGC API for processes with type discriminators for an example).

Example 7. Different input types of the OGC API for processes

```
{
  "process": {
    "inputs": [
      {
        "id": "literalInput",
        "input": {
          "literalDataDomains": [...]
        }
      },
      ,
      {
        "id": "complexInput",
        "input": {
          "supportedFormats": [...]
        }
      },
      {
        "id": "bboxInput",
        "input": {
          "supportedCRS": [...]
        }
      }
    ]
  }
}
```

Example 8. Different input types of the OGC API for processes with type discriminators

```
{
  "process": {
    "inputs": [
      {
        "id": "literalInput",
        "input": {
          "type": "literal",
          "literalDataDomains": [
            {
              "valueDefinition": {
```

```
                "type": "AllowedValues",
                "values": [...]
              }
            },
            {
              "valueDefinition": {
                "type": "AnyValue"
              }
            },
            {
              "valueDefinition": {
                "type": "ValuesReference",
                "reference": "..."
              }
            },
            ,
            {
              "valueDefinition": {
                "type": "AllowedRanges",
                "ranges": [...]
              }
            }
          ]
        }
      },
      {
        "id": "complexInput",
        "input": {
          "type": "complex",
          "supportedFormats": [...]
        }
      },
      {
        "id": "bboxInput",
        "input": {
          "type": "bbox",
          "supportedCRS": [...]
        }
      }
    ]
  }
}
```

The current draft of the OGC API - Processes specification only supports the retrieval of all results of a process in a single document via '/processes/{processId}/jobs/{jobId}/result' (see below). This approach does not support sharing the result of a route computation without additional logic on the recipient side. A possible solution to this would be the definition of an additional endpoint (like '/processes/{processId}/jobs/{jobId}/result/{outputId}') to directly retrieve the result.

Example 9. Result document with a single output

```
{
  "outputs": [
    {
      "id": "route",
      "value": {
        "inlineValue": {
          "type": "FeatureCollection",
          "status": "successful",
          "name": "The name of the Route.",
          "features": [...]
        }
      }
    }
  ]
}
```

8.2.1.3. Final Component Description

Each routing engine implementation is encapsulated in a single WPS process. Their process descriptions only specify the inputs and parameter values that are supported and enable clients to execute the processes without any prior knowledge of the underlying engine. While the processes are very similar to the specified routing process, they differ in some key points from the routing engine capabilities:

- As the processes are deployed in a single service instance, the process identifier is not *routing*, but *org.n52.routing.here*, *org.n52.routing.ecere* and *org.n52.routing.skymantics* respectively.
- The mandatory 'waypoints' parameter, a GeoJSON 'MultiPoint' with at least two points, is divided in one optional and two mandatory inputs: 'origin', the mandatory GeoJSON 'Point' containing the start point of the route, 'destination', the mandatory GeoJSON 'Point' containing the end point of the route and 'intermediates', an optional GeoJSON 'MultiPoint' containing all intermediate waypoints. With this simple change clients can easily determine if a process supports intermediate waypoints and the separation of origin and destination makes it clearer how many points are required.

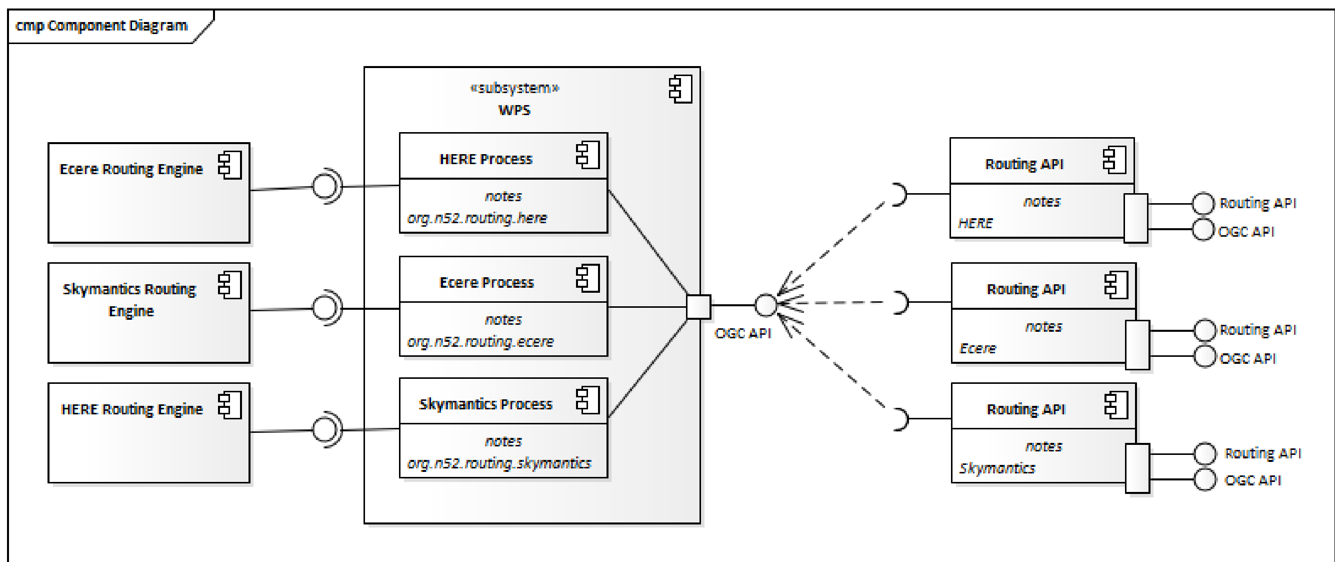

Figure 21. Architecture of the 52°North implementation

To achieve compliance with the routing specification, a Spring Boot [https://spring.io/projects/spring-boot] application was developed that can proxy the process. The application implements both specified options, the OGC API for Processes with a single *routing* process and the simplified Routing API. The actual route computation is delegated to the WPS process, while the proxy is responsible for the persistence of routes and route definitions, the conversion between route definitions and execute documents, handling of synchronous computations and callback handling. Besides this, the proxy splits the supplied 'waypoints' parameter into its internal representation as three distinct parameters.

8.2.2. GIS-FCU WPS

8.2.2.1. Design Considerations

The WPS supporting the Routing API is divided into two options - a WPS option and a Routes option. In order to reduce the implementation overhead and duplication, this implementation was created based on a WPS kernel, and reused this kernel as the other option - Routes.

8.2.2.2. Encountered Challenges and Solutions

- Option Routes and option WPS

The draft Routing API provides two sets of interfaces but similar data structures (Routing Exchange Model). In order to reduce the implementation overhead, the API was designed using the following structure.

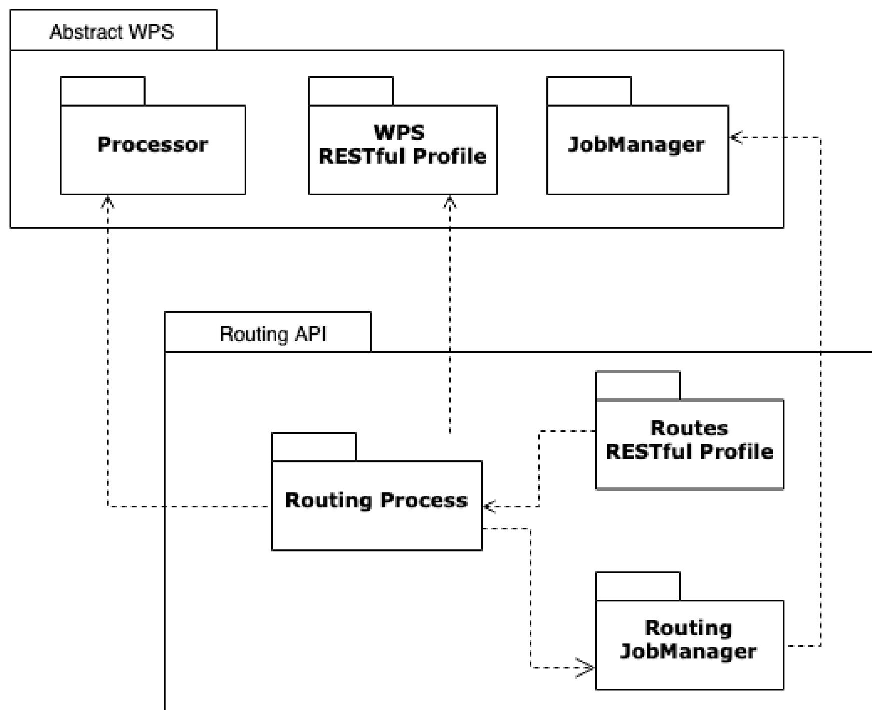

Figure 22. GIS.FCU WPS Components Diagram

The Routes option shares the same functions as the WPS option, but uses a different RESTful Profile to fulfill the definitions of the Routing API.

- Various Routing Engines
  - The scenario introduces 3 Routing Engines, namely Ecere, Skymantics, and HERE. The Ecere and Skymantics engines use similar interfaces and a Routing Exchanges Model, but the HERE API uses a different interface and models. To implement this WPS with different routing engines, the implementation of the process should compose the correct parameter sets for each engine and invoke the right endpoints.
- Sync or Async request
  - For option WPS, the asynchronous request is recommended to follow the WPS specification which uses jobControlOptions as shown in the following listing.

Example 10. async setting in the WPS

```
{
    "inputs": [...],
    "outputs": [...],
    "jobControlOptions": "sync-execute",
    "outputTransmission": "value"
  }
```

- Asynchronous cache duration
  - The server must keep the routing results until the client consumes them or until it times out. This WPS implementation used a simple memory cache to maintain the results.

- Unit of Measurement
  - In the early phase of this pilot, the Unit of Measurement (UOM) brought some confusion on the server side when converting the UOM for max weight or height. Thus, the UOM was well defined in the SWAGGER specification to avoid these confusions.
- i18n contents
  - The contents of the Routing Exchange Model might have characters from other languages. This problem was discussed on github [https://github.com/opengeospatial/RoutingAPIPilot/issues/5] and the content-header approach is recommended for better i18n information exchange.

8.2.2.3. Final Component Description

This WPS implementation successfully connects to three different engines - HERE, Ecere, and Skymantics. Further, three different datasets - HERE, OSM, and NSG were available.

The conformance classes are the place to harmonize the client and the service without fully understanding the entire specification in advance. The format of the conformance class must be simple and clear. This WPS used the "one endpoint for all engines" approach to offer the routing services. For example, see the following listing.

Example 11. Conformance class

```
{
    "conformsTo": [
        "http://www.opengis.net/spec/ogcapi-processes-1/1.0/conf/core",
        "http://www.opengis.net/orapip/routing/1.0/conf/core",
        "http://www.opengis.net/orapip/routing/1.0/conf/intermediate-waypoints",
        "http://www.opengis.net/orapip/routing/1.0/conf/max-height",
        "http://www.opengis.net/orapip/routing/1.0/conf/max-weight",
        "http://www.opengis.net/orapip/routing/1.0/conf/obstacles",
        "http://www.opengis.net/orapip/routing/1.0/conf/routing-engine",
        "http://www.opengis.net/orapip/routing/1.0/conf/routing-algorithm",
        "http://www.opengis.net/orapip/routing/1.0/conf/source-dataset",
        "http://www.opengis.net/orapip/routing/1.0/conf/time",
        "http://www.opengis.net/orapip/routing/1.0/conf/callback",
        "http://www.opengis.net/orapip/routing/1.0/conf/result-set",
        "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode",
        "http://www.opengis.net/orapip/routing/1.0/conf/delete-route"
    ],
    "http://www.opengis.net/orapip/routing/1.0/conf/core": {
        "values": [
            "fastest",
            "shortest"
        ]
    },
    "http://www.opengis.net/orapip/routing/1.0/conf/routing-engine": {
        "values": [
            "Skymantics",
            "Ecere",
```

```
                "HERE"
            ]
        },
        "http://www.opengis.net/orapip/routing/1.0/conf/routing-algorithm": {
            "values": [
                "Dikjstra",
                "Floyd Marshall",
                "A*",
                "dijkstra-bi",
                "dijkstra-ch",
                "astar-bi",
                "astar-ch"
            ]
        },
        "http://www.opengis.net/orapip/routing/1.0/conf/source-dataset": {
            "values": [
                "NSG",
                "OSM",
                "HERE"
            ]
        },
        "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode": {
            "values": [
                "sync",
                "async"
            ]
        },
        "http://www.opengis.net/orapip/routing/1.0/conf/result-set": {
            "values": [
                "full",
                "overview",
                "segments"
            ]
        }
}
```

The client can very easily parse the conformance class and generate the routing parameters on the client side. The downside of this approach is that the engine might not support specific algorithms or datasets, and the client would as a result receive an HTTP 404 error response which might lead the client to be confused by the error handling.

8.2.3. Skymantics WPS

Below is the basic application flow for WPS operations

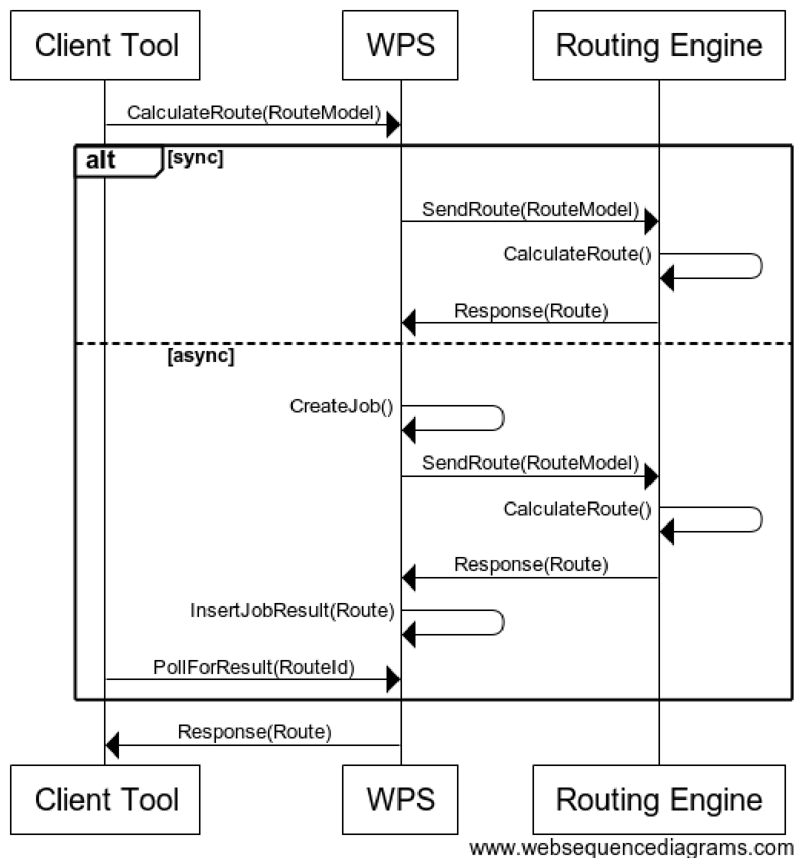

Figure 23. Sequence diagram of sync and async route request / response.

8.2.3.1. Design Considerations

Java was the language of choice for the Skymantics server development, utilizing the Spring Framework (https://spring.io) for RESTful API design. The Framework's large community of support and simple implementation allows for quicker development time with an operational standard. An additional use of Spring is its capability to rapidly test new implementations required in this pilot, including both internal to the WPS and from external members due to its embedded servlet. Maven was also chosen as the build tool of choice, due to both developer familiarity and its robust plugin options. The WPS implementation was deployed on an Azure Container instance. A Docker image of a Tomcat 9 Linux server was used as a baseline, with the application Web Archive (WAR) file and the relevant production libraries included. This containerized image was then deployed on Microsoft Azure's Container Instances. This allows for ease of deployment in a production environment, and also could allow for other providers to implement their own container on separate hardware.

The WPS needed to be able to take a variety of conformant JSON, including GeoJSON, and respond in the appropriate manner. As such, request Model classes needed to be made in order to perform request validation and any additional processing.

In addition, two API Options were proposed during the Pilot kickoff - the more codified WPS Option, and a less verbose Routes Option. This meant that any request to the WPS could be different, but must be handled in the same fashion regardless of input. To facilitate this, Skymantics implemented two different endpoints a client could submit requests to:

```
http://52.189.65.85:8080/orp/Routes
http://52.189.65.85:8080/orp/processes/routing/jobs
```

8.2.3.2. Encountered Challenges and Solutions

Initially, some increased development time was required. This was due to the requirement for both a simplified Route option and a more complex WPS option, as some properties of a more enriched WPS option may not exist on a simplified option (such as Units of Measurement). However, different implementation methods have yielded important insights into OGC API development. A complex Routing Model requirement such as Option WPS led to increased code complexity and therefore development time; However, this complexity reduces the amount of out-of-band information required by the client application to appropriately communicate with the various routing engines implemented in the pilot.

8.2.3.3. Final Component Description

Two forms of endpoints to this WPS were postulated, conforming to a less stringent Routes form (AKA Option 1) and a heavier, more generic form WPS (AKA Option 2). Both forms of requests and their accompanying markup are accepted by the WPS. Internally, the input received by the WPS is serialized via Jackson JSON library into a Route object. This representation inherits from a generic ProcessInput abstract class specifying properties not exclusive to routing, such as MimeType of the request, a name, etc, as per the WPS standard. If the Route conforms to the input specified by the WPS process offering, a Job is then generated by the server, with an accompanying JobId. This job is then passed to a JobHandler singleton, which orchestrates all HTTP requests to a routing engine. Apache's HTTP Commons library [https://httpcomponent.apache.org] was used to facilitate communications between the WPS and the relevant Routing Engines. This includes a multithreaded, non-blocking input-output offering in which requests may be made synchronously and returned to the end client, or alternatively returned asynchronously.

The asynchronous communications approach utilized Apache's HTTP Commons library to achieve asynchronous callback functionality of web request / responses and Google's Guava library to provide a cache of currently running route operations. When a Job is created and the Asynchronous option is indicated, a callback method on the HTTP communication classes insert the response of a particular routing engine into the value of the Cache for a given job key. Consequently, a user may access this result up to 20 minutes after a result has been inserted into the cache. After this time period, the record is expunged from the data structure to reduce memory footprint on the server. These implementations are the same regardless of which markup Option is indicated by the user- there is no delineation between Routes. As the Route Object is a subset of Jobs, and the Route Model is of a Job Result subclass, both response to the end client are of the same type, and therefore agnostic to both the WPS Cache and the End user- The WPS handles Option 1 requests the same way it would handle an Option 2 request, and responses to the client are the same Route Model response regardless.

The Jackson Java library was utilized to validate and transform the received Route Model request to a strongly-typed Route object. Validation of parameters and GeoJSON structures are conducted, and any missing parameters are defaulted to predefined values (such as using A* as a default algorithm, or using Skymantics Routing Engine if none is specified). This Object is then deserialized back to

JSON and sent to the specified Routing Engine, with additional requirements if it is a bespoke API non-conformant with the Route Model.

The WPS also has HATEOAS capability, where route requests in an asynchronous manner return their respective API options, such as result endpoint and status endpoint.

Example 12. Example Request


```json
{
  "name": "Route from A to B",
  "waypoints": {
    "type": "MultiPoint",
    "coordinates": [
      [
        -77.0721,
        38.9309
      ],
      [
        -77.0352,
        38.8897
      ]
    ]
  },
  "preference": "fastest",
  "maxHeight": 4.5,
  "maxWeight": 12,
  "dataset": "OSM",
  "engine": "Ecere",
  "algorithm": "A*",
  "when": {
    "timestamp": "2019-05-23T19:06:32Z",
    "type": "arrival"
  }
}
```

Example 13. Example Response

```
{
    "_links": {
        "Result": {
            "href": "http://52.189.65.85:8080/orp/routes/d84e9765-b03b-4e59-8d11-
6bc89c52a832"
        },
        "Status" : {
            "href": "http://52.189.65.85:8080/orp/routes/d84e9765-b03b-4e59-8d11-
6bc89c52a832/status"
        },
        "self": [
            {
                "href": "http://52.189.65.85:8080/orp/routes?mode=async"
            },
            {
                "href": "http://52.189.65.85:8080/orp/routes?mode=sync"
            }
        ]
    }
}
```

8.2.4. Helyx WPS

8.2.4.1. Design Considerations

The goal was to implement a scalable, robust and performant WPS with a rapid startup time. To enable this, Helyx used a microservice approach and chose Option 1 to build the server against. Option 1 suited a microservice approach better than Option 2 as Option 2 is monolithic in nature due to the required amount of inputs. However Option 2 could be treated as a gateway with intelligent routing, filtering and service discovery. In which case, the WPS acts as a gateway to broker with the processes deployed as separate microservices. However, in doing so we would lose the benefit of OpenAPI definitions which Option 1 provided.

For the Option 1 implementation Helyx used Kotlin, a Java Virtual Machine language by Jetbrains which has proven to be useful in building robust solutions. Given one of the goals for the implementation was performance, Kotlin has a number of benefits. The conciseness and performance of Kotlin allows for a non-blocking approach to development, improving performance and robustness.

The Kotlin microservice was built on top of a native web server and placed in an *Uber JAR*, that is, a JAR file that contains all of the dependencies. This Jar file was placed in a Docker container with environment variables to allow for scalability, which enabled exploitation of a relatively small memory footprint and fast startup time. This was achieved using a configuration management tool called Ansible [https://www.ansible.com].

A number of serialization utilities were used in the implementation including Jackson and Moshi.

Moshi was used to satisfy a requirement for polymorphic data objects to mirror the polymorphic nature of option 2. An additional requirement was a lightweight Publish-Subscribe pattern embedded into the API, this design pattern provides a subscriber URI for a client to POST the data to when the route is ready to be consumed.

8.2.4.2. Encountered Challenges and Solutions

8.2.4.2.1. Conformance classes

During implementation, clients required the capability to retrieve supported functionality from each routing engine. The initial approach was to provide a simple list in the following format to expose the engine capabilities via the conformance links:

```
"http://www.opengis.net/orapip/routing/1.0/req/HERE-routing-engine",
"http://www.opengis.net/orapip/routing/1.0/req/HERE-routing-algorithm",
"http://www.opengis.net/orapip/routing/1.0/req/HERE-source-dataset"
```

This exposes some issues. There were over 25 different elements in the list leading to an unmanageable number of conformance classes to standardize. In addition, this linear list could not account for dependent functionality. For instance, one routing engine may support obstructions but another may not therefore, an alternative approach was taken to provide nested conformance classes to prevent the API from providing conflicting functionality for differing engines by providing an initial set of keys. Each key was mapped to a list to provide potential input values for the functionality of the key. While this successfully solved the above challenge, the approach still led to a large number of keys. This approach also means the keys are not unique in the payload returned in the request to the conformance endpoint. Therefore, a client has to cross-reference the keys in the initial array with the following indexes as a means to retrieve input values.

```
conformsTo:
  0: "http://www.opengis.net/spec/ogcapi-processes-1/1.0/conf/core"
  1: "http://www.opengis.net/orapip/routing/1.0/conf/core"
  2: "http://www.opengis.net/orapip/routing/1.0/conf/intermediate-waypoints"
  3: "http://www.opengis.net/orapip/routing/1.0/conf/max-height"
  4: "http://www.opengis.net/orapip/routing/1.0/conf/max-weight"
  5: "http://www.opengis.net/orapip/routing/1.0/conf/obstacles"
  6: "http://www.opengis.net/orapip/routing/1.0/conf/here-routing-engine"
  7: "http://www.opengis.net/orapip/routing/1.0/conf/ecere-routing-engine"
  8: "http://www.opengis.net/orapip/routing/1.0/conf/ecere-routing-algorithm"
  9: "http://www.opengis.net/orapip/routing/1.0/conf/ecere-routing-dataset"
  10: "http://www.opengis.net/orapip/routing/1.0/conf/skymantics-routing-engine"
  11: "http://www.opengis.net/orapip/routing/1.0/conf/skymantics-routing-algorithm"
  12: "http://www.opengis.net/orapip/routing/1.0/conf/skymantics-routing-dataset"
  13: "http://www.opengis.net/orapip/routing/1.0/conf/time"
  14: "http://www.opengis.net/orapip/routing/1.0/conf/callback"
  15: "http://www.opengis.net/orapip/routing/1.0/conf/result-set"
  16: "http://www.opengis.net/orapip/routing/1.0/conf/sync-mode"
  17: "https://helyx.co.uk/ogc/routing/1.0/conf/sse"
http://www.opengis.net/orapip/routing/1.0/conf/core:
  values:
    0: "fastest"
    1: "shortest"
http://www.opengis.net/orapip/routing/1.0/conf/skymantics-routing-algorithm:
  values:
    0: "astar"
    1: "astar-bi"
    2: "astar-ch"
    3: "dijkstra"
    4: "dijkstra-bi"
    5: "dijkstra-ch"
http://www.opengis.net/orapip/routing/1.0/conf/skymantics-routing-dataset:
  values:
    0: "HERE"
    1: "OSM"
    2: "NSG"
http://www.opengis.net/orapip/routing/1.0/conf/ecere-routing-algorithm:
  values:
    0: "astar"
    1: "astar-bi"
    2: "astar-ch"
    3: "dijkstra"
    4: "dijkstra-bi"
```


Figure 24. Helyx API Conformance Classes

In the future, it may be prudent to implement a tier-based matrix, like below:

```
http://www.wps-helyx.westeurope.azurecontainer.io:5600/conformance
    --> "http://www.opengis.net/orapip/routing/1.0/req/skymantics-routing-engine",
        --> "routing-algorithm",
            - "astar-ch"
            - "astar"
        --> "source-dataset",
            - "OSM"
            - "NSG"
            - "HERE"
        --> "core"
            - "shortest"
            - "fastest"
```

The above example negates the need for cross-referencing. This allows for an efficient method of parsing and is less ambiguous. In the future, conformance classes should be an important aspect of future OGC API design. The extent to which conformance class approaches could be standardized should be investigated further.

8.2.4.2.2. Simplified requests

Simplified requests and JSON serialization created another challenge as they expect clients to be able to send any request by eliminating keys in the payload allowing for a minimal request. This poses a number of issues when the request does not conform to the JSON schema in the API. The

API server or client has to gracefully handle any deviation from the schema caused by these omissions. There are security risks in taking a lenient JSON payload approach rather than a stricter method where any unexpected request would return a malformed error response code. A potential risk when using a lenient JSON approach would be the service is open to a Distributed Denial-of-Service (DDOS) attack if intensive computation is applied to each potential request. Intensive computation occupies threads that would otherwise be safe if an unexpected payload is sent to the service. Another risk is that the server could be open to Man in the Middle attacks and JSON Injection based attacks if the service is less vigilant due to the minimal approach.

8.2.4.3. Final Component Description

In the final representation of the API, the landing page conforms to the draft OGC API - Common specification in most aspects except the routes URI which has POST, GET and DELETE capabilities to modify a resource. At this point the OGC API - Common specification does not yet consider POST and DELETE capabilities.

Another method of execution was implemented named Server Sent Events (SSE). As mentioned previously, Server Sent Events are a hybrid approach much like a websocket where a consumer subscribes to a connection with the API after initializing a request and following the link in the location header. The API then forwards the resulting route through this tunnel-like connection to the client. This implementation is useful as another method of execution to provide streaming functionality to consumers without websocket capabilities.

As this service is open to the web, the service would need a secure authentication option. The current implementation of the Helyx WPS provides a basic internal Access Management via request headers. With the current implementation, a client application can provide a user id when making requests and the service will authenticate based on those credentials. A future improvement on the basic implementation is to implement authentication using a token-based(Jwt) approach for Access Management.

The application was briefly load-tested using the HERE routing engine and had interesting results. With 10,000 concurrent requests over a minute the service handled well with only 300mb of RAM usage. The bottleneck is listing the resources in the '/routes' URI, due to this being a large array. A large array only allowed a browser to display ~12000 elements and when submitting the request with Postman the response was never received, even with GZIP compression in place. This testing demonstrates that while the routing API is robust, the significant quantity of resources proves to be a limitation for consumer applications. Potential solutions could include:

- Chunking the data into a multipart response
- Subscribing to the request via a websocket or otherwise

Any solution would have to be accounted for in the specification.

Future improvements to the draft Routing API could include:

- Integration with a common collections endpoint - The Routing API could reference processing resources at the common collections endpoint rather than at the routes endpoint. This avoids duplication of capability between the Routing API and the collections endpoint and supports the Separation of Concerns Design Principle.

- Hazelcast & binary collections - All process results could be retained in an In-Memory Data Grid such as Hazelcast or Redis which enables the service to hold potentially billions of entries in a binary format. This level of abstraction provides further opportunities to exploit.
- Serverless application - A serverless approach could be adopted to enable simple interoperability and deployment across all devices, by removing the routing microservice as a web service. The POST capability would be exposed by an intelligent gateway and when a route is requested the application starts up, gets a route and deploys it to the collections endpoint. This serverless approach would be complemented by the proposed In-Memory Data Grid approach referenced above.
- Links & their relationships - In the current version of the Swagger Document, the JSON schema for Links denotes the only mandatory element is href and rel should be a mandatory field. - See https://tools.ietf.org/html/rfc8288#section-2. Without this field it is difficult for clients implementing HATEOAS compliant functionality, as there is little information gained from the 'href' attribute. Going forward if the OGC would keep HATEOAS in mind, consideration could be made towards RFC8288 when designing OpenApi schema documents. This removes the need for href string parsing and utilizing common libraries that parse link relationship types.

8.3. Routing Engine Implementations

8.3.1. Skymantics Routing Engine

8.3.1.1. Design Considerations

The scope of the Skymantics routing engine was relatively broad; extracting data from two different datasets (OSM and NSG) and trying several flavors of the Dijkstra (well known shortest path) algorithm. The scope was later extended to additionally cope with HERE data and the A* algorithm family. The A* algorithm is a routing process based upon nodes, its significance is that it routes by finding the vertices that are closest to the destination, rather than the start point. In order to be able to stretch the scope as much as possible within the project timeframe, the top priority for the design of the routing engine was rapid prototyping. Performance was considered secondary and was only addressed at the last stages of the implementation.

Python3 was chosen as the language to implement the routing engine. This allowed use of as many available off-the-shelf libraries as possible, such as Fiona (to extract data from the GeoPackages) https://pypi.org/project/Fiona/, Vincenty (to calculate distances) https://pypi.org/project/vincenty/, or json (to create the geojson output based on the Routing Exchange model defined in the pilot) https://docs.python.org/3/library/json.html , among others. The data extraction process was designed to store the data on roads and junctions in separate MySQL databases using a common, pre-processed data structure. This structure describes the road mesh in terms of a graph and it includes nodes, edges, shortcuts, and additional information to process route restrictions or to generate the draft Route Exchange Model. By using this common data structure, a route could be calculated in the same way, independently from the dataset it originated from, whether OSM, NSG or HERE. This facilitated the comparison of routes from different sources.

In order to have full control of the implementation, the routing algorithms were built from scratch, without the use of external libraries. Three different options were considered from the start for each algorithm family:

1. A vanilla version, start calculating the route from A until B is reached.
2. A bi-directional version, start calculating the route from A to B and from B to A backwards; stop when both paths meet.
3. A version with Contraction Hierarchies optimization (which would use the bi-directional version of the algorithm but prioritizes the use of higher hierarchy roads, such as freeways, than lower ones, such as residential streets).

8.3.1.2. Encountered Challenges and Solutions

8.3.1.2.1. Dataset related challenges

- Different speed limits

Datasets define different speed limits for the same street, leading to different route duration estimations, as well as different routes calculated per dataset when requesting for the shortest route.

The HERE dataset defines the speed limit for each direction in the same street, which is an optimal approach, whereas OSM and NSG datasets define one single speed limit for both directions.

Some streets have missing speed limit information and were set to -999999 in NSG or to 0 in HERE or *missing* the *speedlimit* tag in OSM. This lack of information forced the routing engine to *estimate* the speed limit for those streets, making use of the road hierarchy information ("rin_roi" property in NSG, "z_order" property on OSM, "FUNC_CLASS" property in HERE) and using the most common speed limit found in other streets with similar hierarchy.

Note: this is one reason why different routing engines or different datasets will produce different routes when using the same waypoints.

- Different levels of street hierarchy

The OSM dataset contains many more secondary streets, alleys and paths that the other datasets lack (roughly about 30% more streets), therefore it is able to find shortcuts where others cannot. Alternatively, the NSG dataset had the fewest street levels, which provided a challenge when comparing routes.

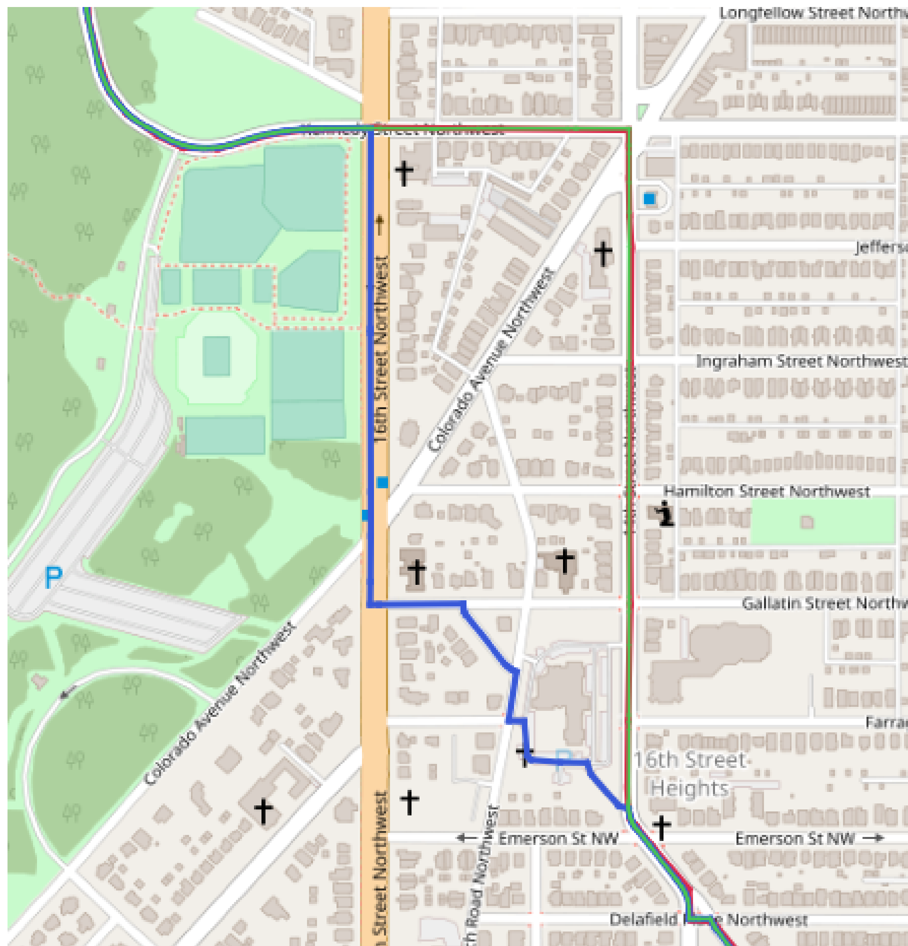

Figure 25. Shortest path calculated with OSM (blue), NSG (green) and HERE (red). OSM finds a shorter path due to its better street capillarity.

8.3.1.2.2. Routing related challenges

- Contraction Hierarchies might generate substantially different routes

Contraction Hierarchies (CH) is an optimization algorithm that prioritizes the use of the higher hierarchy roads, such as freeways, than lower ones, such as residential streets. It usually gives an optimal or near-to-optimal route at a fraction of the computing cost.

The CH optimization relies on the definition of road hierarchies made by the dataset and assumes that fastest routes will always tend to go through roads with higher priorities. This is usually true in long-distance routes, but not necessarily in urban routes, such as in this pilot, which might lead to routes that are substantially different from the ones generated by the same algorithm without the CH optimization.

As a rule-of-thumb, a route calculated using CH optimization would take 10 times less computing time, but the route generated would be usually between 5 and 10% less optimal.

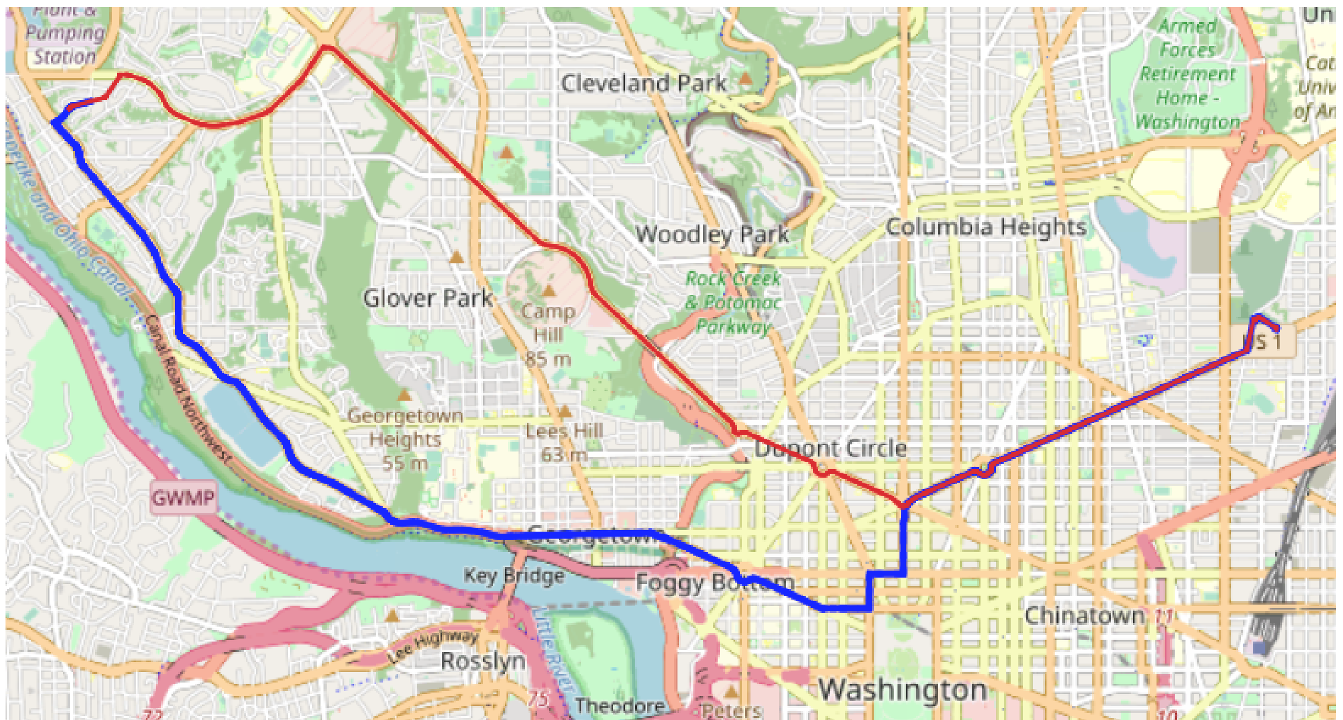

Figure 26. Dijkstra calculated fastest route (red) vs dijkstra-ch calculated fastest route (blue) with OSM dataset. Dijkstra-ch was x50 faster to compute but the route was 5% below the optimal.

As the hierarchies are defined in the dataset, the results of the CH algorithms would depend heavily on the dataset used. Here is an extreme example for the same route calculated from the National Cathedral to Washington Monument, using HERE and OSM datasets, with both basic A* and A* with Contraction Hierarchies algorithms. For both datasets, the A* algorithm finds the most direct route. However, in the case of HERE dataset, the CH algorithm generates a route that is 60% longer, even though it takes just 0.2 seconds to calculate (instead of 2 seconds). Whereas in the case of the OSM dataset, the CH algorithm takes just 0.4 seconds to calculate (instead of 7 seconds) but the route is still the optimal one.

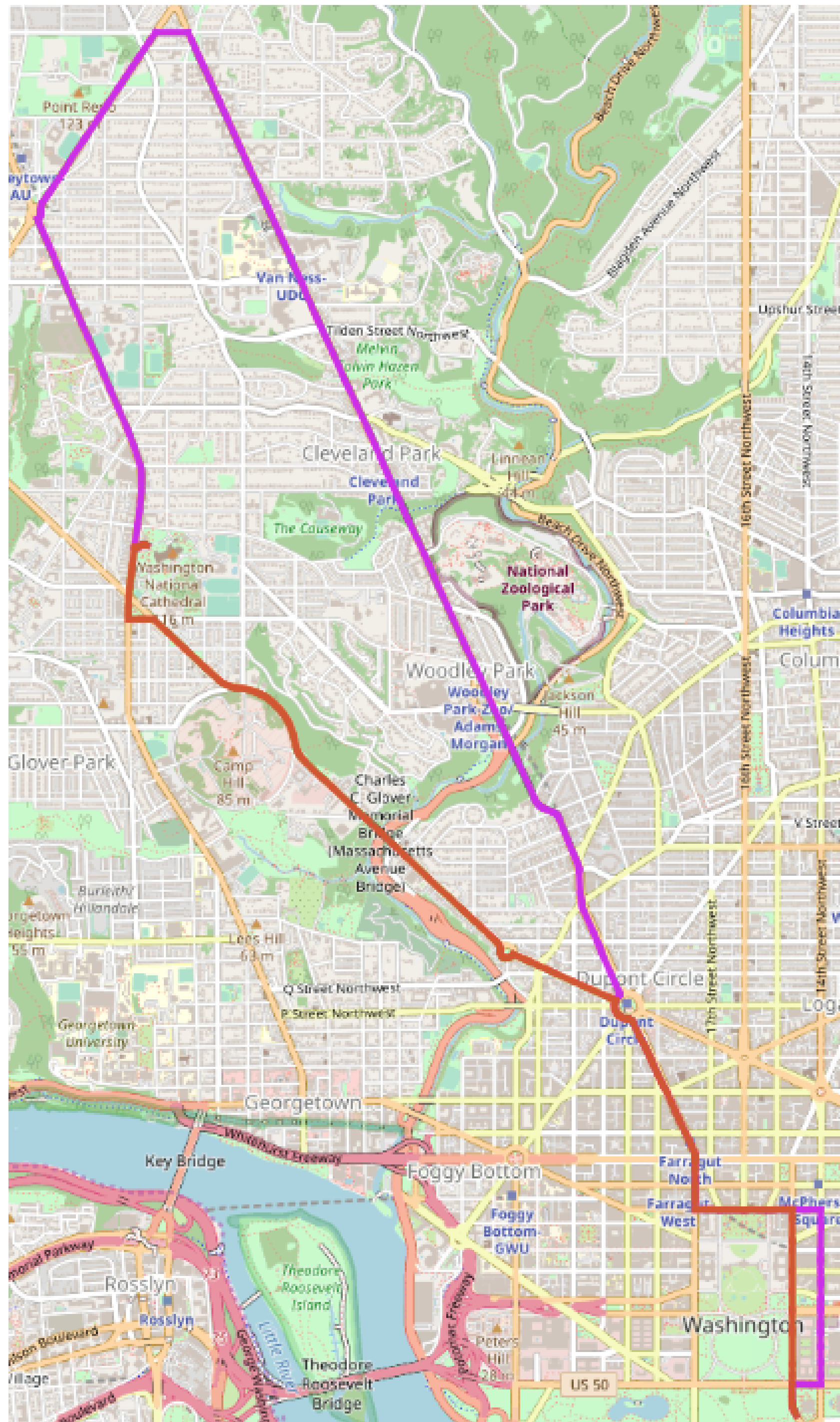

Figure 27. On the HERE dataset, the route calculated with Contraction Hierarchies (pink) is 60% longer, even though it is calculated at x10 faster computation speed.

Figure 28. On the OSM dataset, the route calculated with Contraction Hierarchies is the optimal, but at almost x20 faster computation speed.

- Maneuver penalties

When calculating the fastest route between two points, an application cannot rely solely on speed limits, but it is necessary to take into account maneuvers and the changes in speed they imply. For example, turning right into a different street should have a penalty, and turning left should have an even higher penalty. If we do not apply these penalties, the fastest route between two points might end up being a winding road.

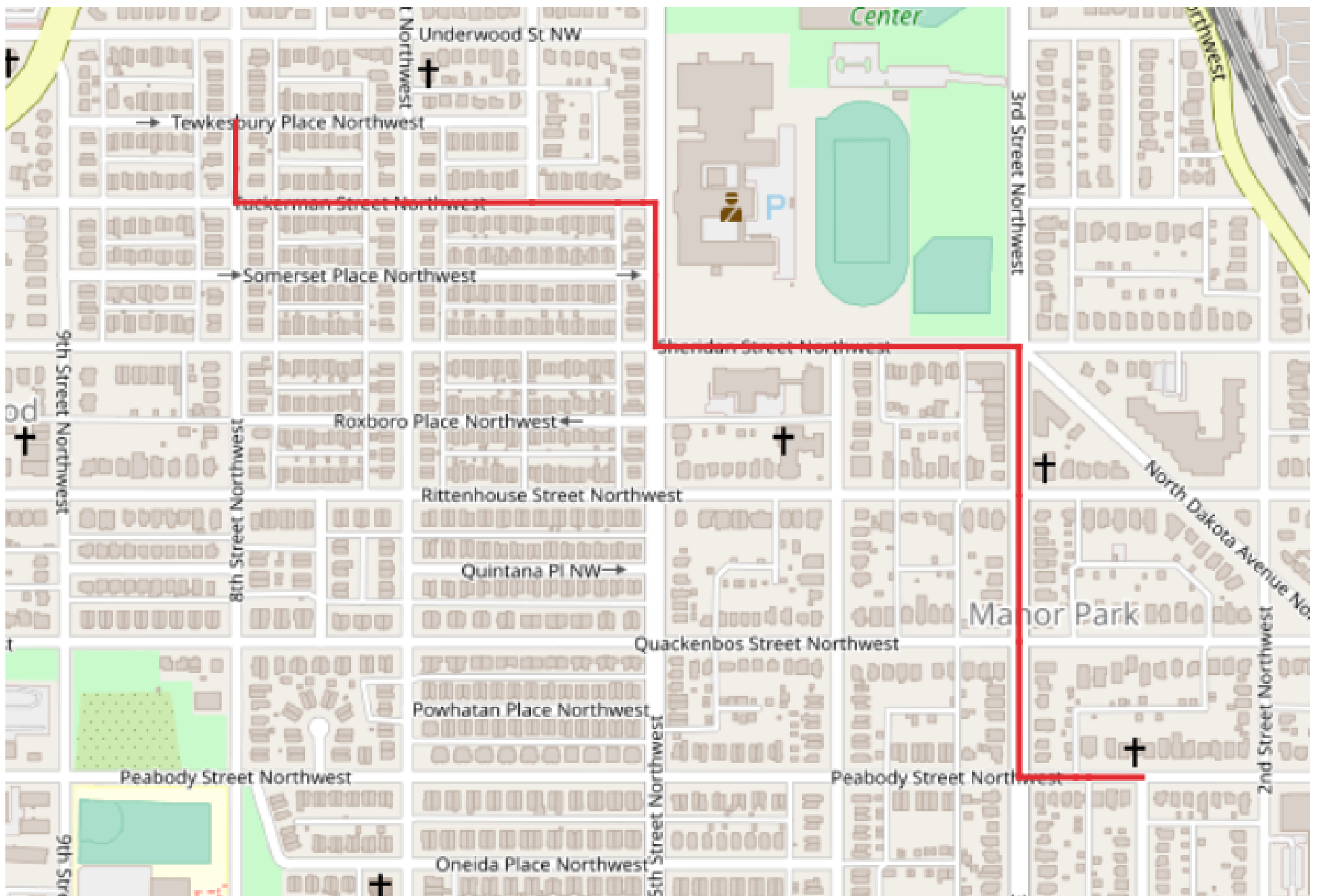

Figure 29. Fastest route calculated in a residential area without maneuver penalties.

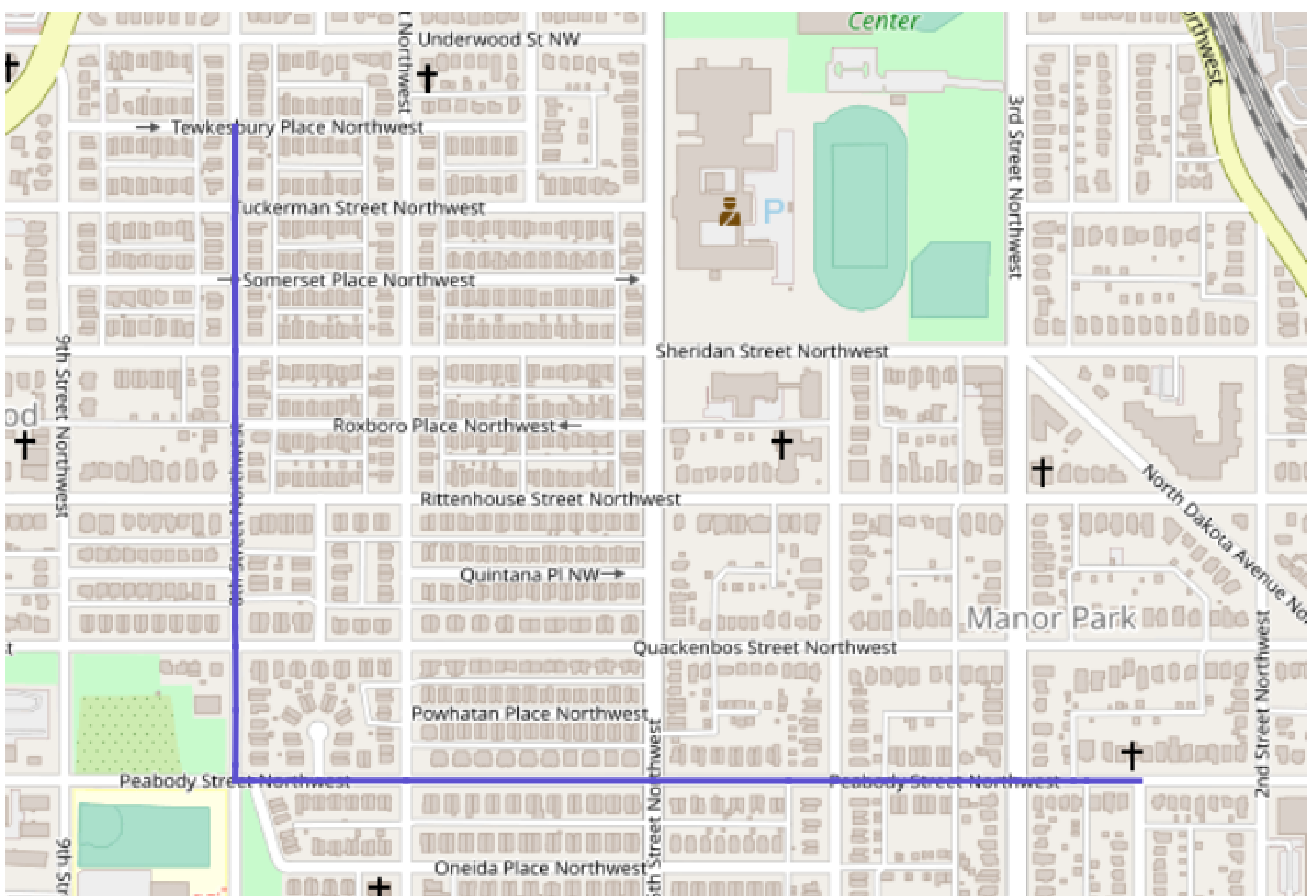

Figure 30. Same route calculated with maneuver penalties.

Note: this is one reason why different routing engines will produce different routes for the same way points.

- Maneuver restrictions only present in HERE dataset

A unique feature included in the HERE dataset is forbidden maneuvers. That is, a series of moves that are not allowed even if physically possible, such as a forbidden turn at a crossroad. There is even information regarding the times when the turn is forbidden, which provides a fine-grained level of detail to calculate accurately a route depending on the time.

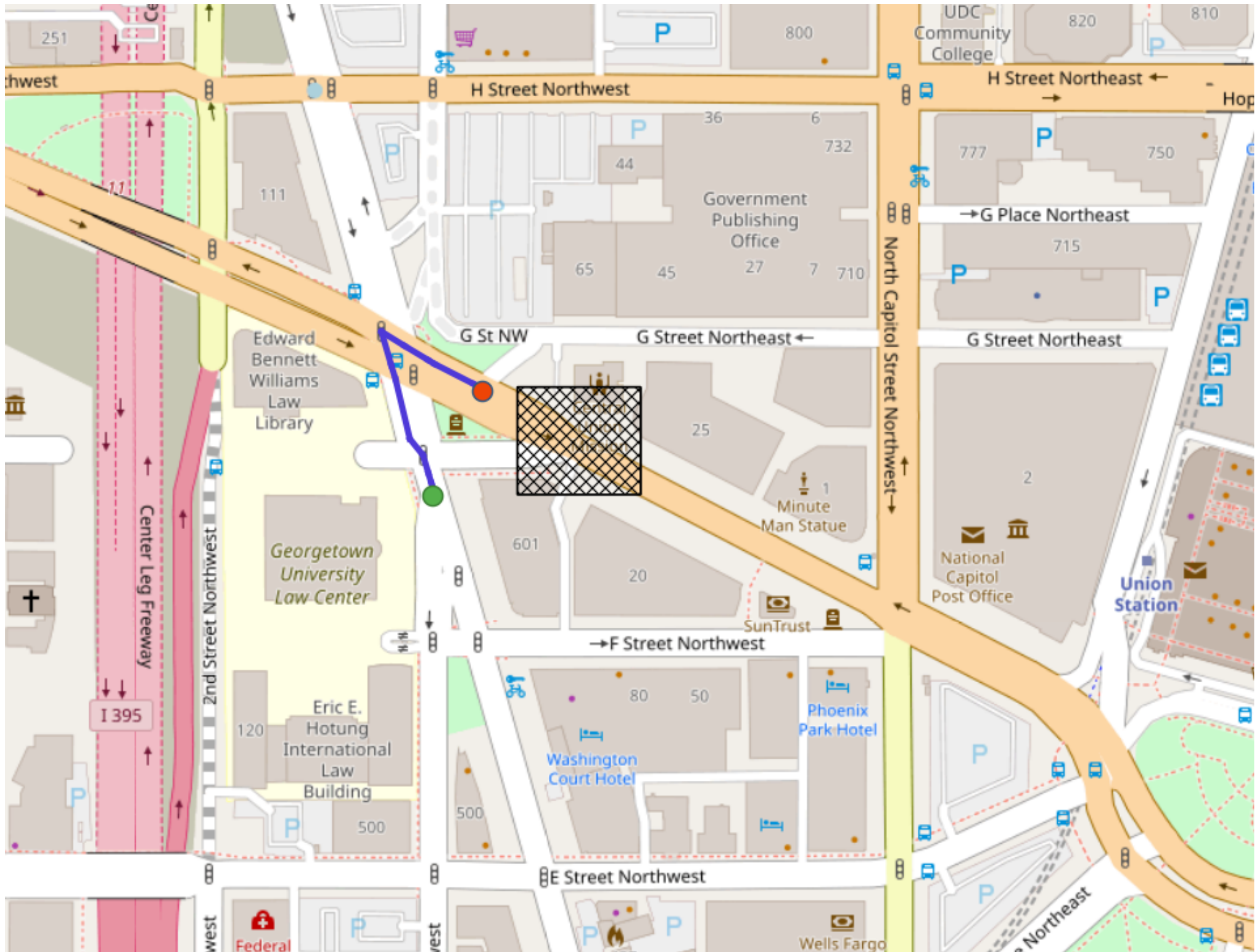

Figure 31. OSM route from red dot to green dot (stripped area blocked to force driving forward). OSM routes through a forbidden turn.

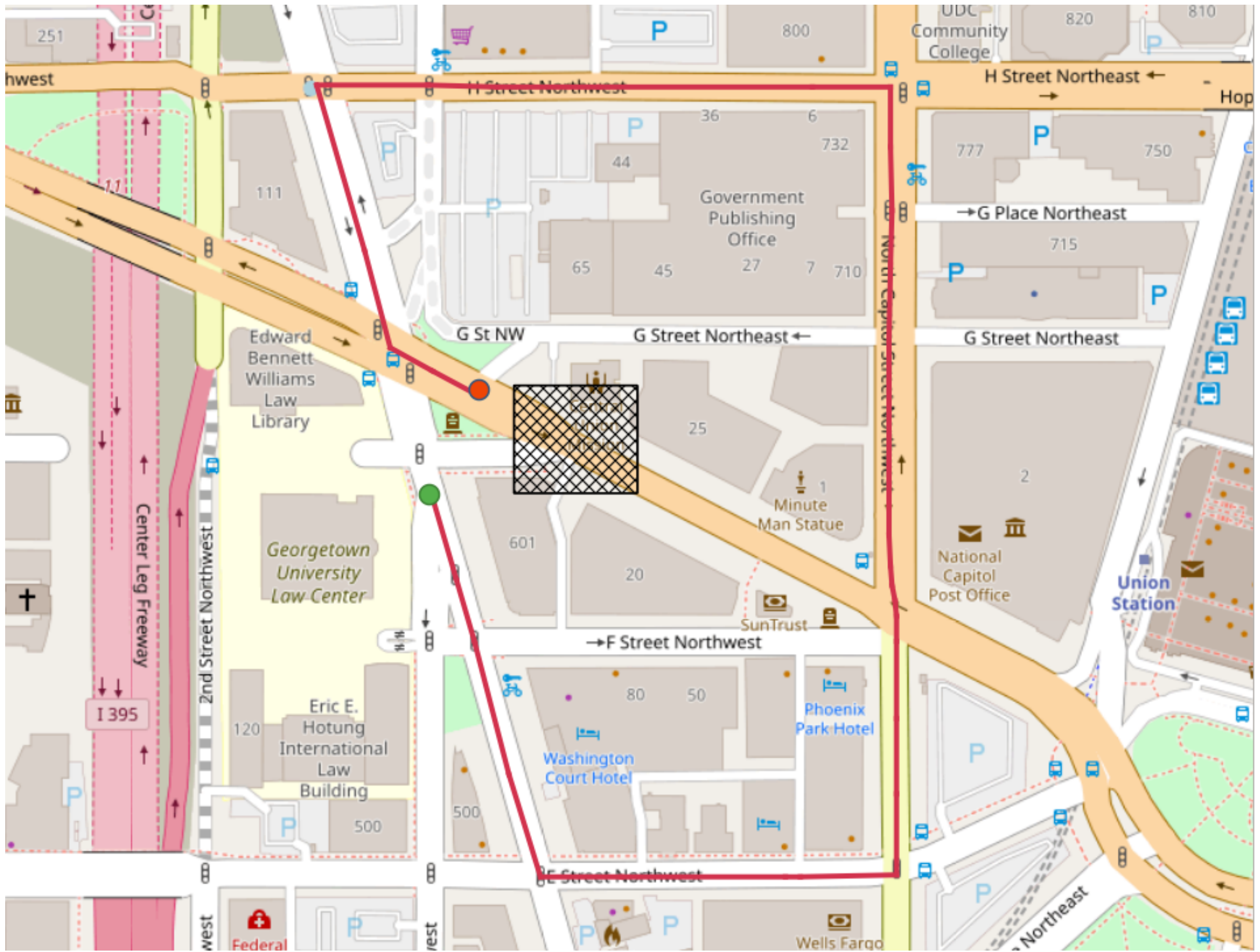

Figure 32. Same route using HERE dataset. The route is calculated to avoid using the forbidden turn.

A practical case is shown below in traveling from Farragut Park to L'Enfant Plaza using HERE and OSM datasets. The image uses the Dijkstra bidirectional algorithm and fastest route.

Figure 33. Routes from Farragut Park to L'Enfant Plaza, using OSM dataset (blue) and HERE dataset (red).

Both routes are relatively similar, but actually OSM does not follow the law: it is not allowed to turn left to Constitution Avenue if driving South on 15th Street. This is what a user would see, while following OSM driving instructions and needing to turn left, according to Google Street View.

Figure 34. Driver view at Constitution Av with 15th Street. OSM-based route indicates turn left but it is not allowed.

Unfortunately, not all forbidden maneuvers are encoded in HERE dataset, which somewhat undermines the value of this feature.

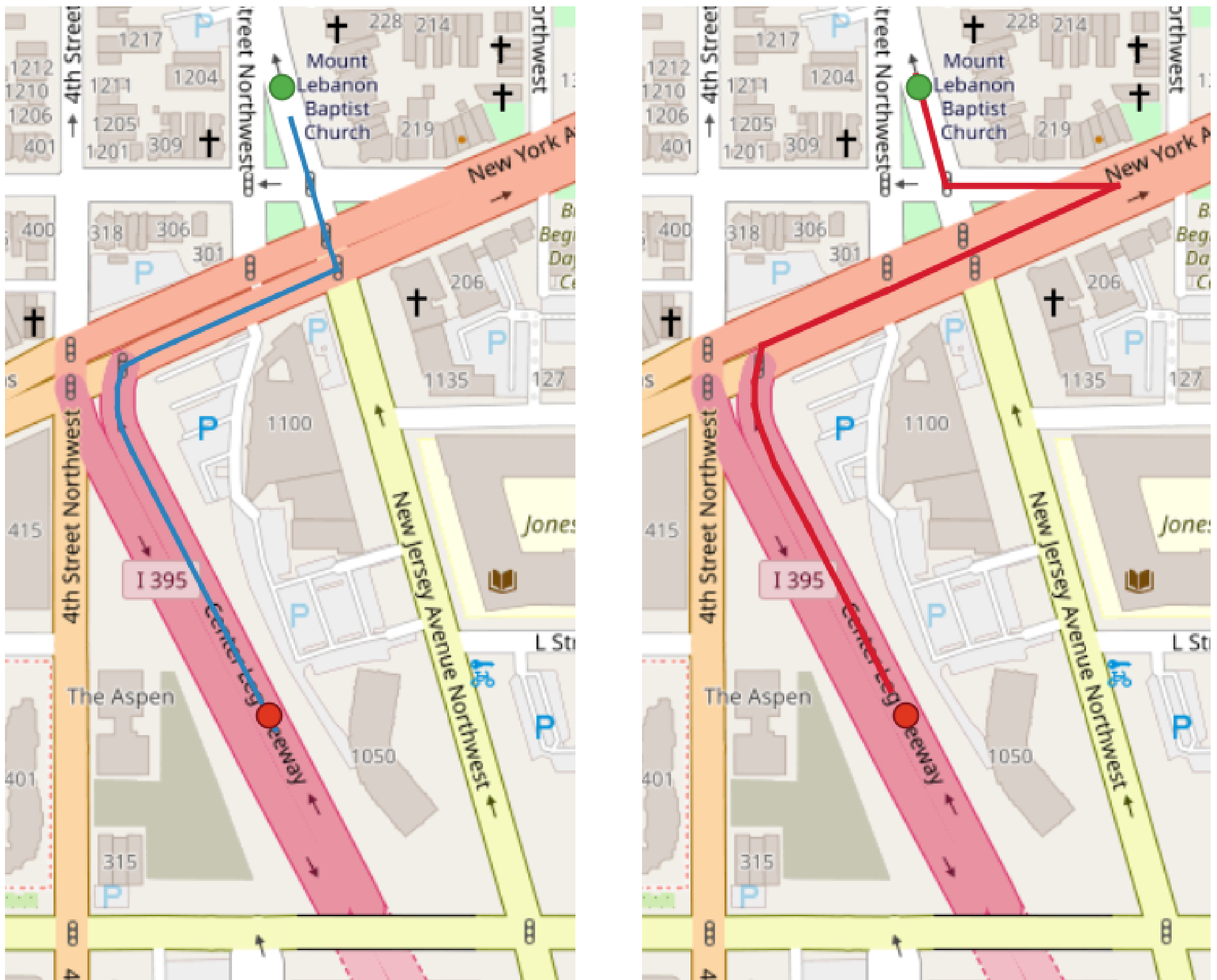

Figure 35. OSM route (left) from red dot to green dot vs HERE route(right). OSM routes through a forbidden turn, which is avoided by HERE only to take next an equally forbidden turn.

8.3.1.2.3. Operation related challenges

- Performance

Once the routing engine was completed and deployed in production, performance became an issue, as some route requests could take several minutes to compute. The main bottleneck was in the communication with the database, which required several thousands of queries for long routes, in turn due to implementation decisions.

The solution applied was first to pre-process all the graphs (that is, for each dataset, with and without shortcuts, as well as forward and backward direction) and to pre-load them into memory together with the rest of information required to build the Route Exchange model. Now routes are calculated in less than a second for short routes or for fast algorithms (like those with Contraction Hierarchies), or in a few seconds for long routes and slow algorithms (like vanilla Dijkstra).

8.3.1.3. Final Component Description

The route engine works in four different stages:

1. Extract data from the datasets and store it in a local database

2. Pre-process the graphs and generate the in-memory routing data
3. When a route request is received, calculate the route
4. Generate the GeoJSON based on the Route Exchange model

8.3.1.3.1. Stage 1: extraction

Data is extracted from the three different datasets, with a process adapted to each one of them. Information on road junctions, road segments, directions, distances, speed limits, road hierarchies, road names and restrictions is collected and stored in a common format.

For the pilot testing area (Washington DC), here is the size of each raw graph:

Dataset	Nodes	Edges
NSG	73475	132768
OSM	110312	213412
HERE	44502	83264

This stage takes between 10 and 15 minutes per dataset.

8.3.1.3.2. Stage 2: pre-process

Nodes are analyzed to find the intersections. Then, shortcuts are built between the intersections in order to speed up route calculations. Here is the size of each preprocessed graph:

Dataset	Edges	Savings
NSG	26962	80%
OSM	120763	43%
HERE	45268	46%

Then, for each dataset, both raw and pre-processed graphs are built (both forward and backward direction), saved and preloaded into memory. Other information required to build the Route Exchange model, such as road names or relation between shortcuts and raw edges, is also preloaded into memory.

This stage usually takes between one and several hours per dataset.

8.3.1.3.3. Stage 3: route calculation

When a route request is received, a route is calculated based on the input data. Raw graphs are used to find the path from the start/end points to the closest intersection; then pre-processed graphs are used to speed up calculation. For bidirectional and CH algorithms, both forward and backward graphs are used.

The route calculated includes information on the sequence of nodes, the edges connecting them, distance and travel time between nodes.

8.3.1.3.4. Stage 4: GeoJSON generation

The route is traveled forward while building the GeoJSON output, following the Route Exchange model specification. A route overview is generated in parallel with each route segment, in order to speed up the process.

8.3.2. Ecere Routing Engine

8.3.2.1. Design Considerations

Ecere built a routing engine for calculating routes using the OpenStreetMap dataset. This engine is named OSMERE for *OpenStreetMap Ecere Routing Engine*.

Ecere decided that it would implement the A* algorithm, and work with a road network constructed from OpenStreetMap data.

The engine was to be used in Technology Integration Experiments with components from other participants including both online and offline scenarios. To do so it produced a GeoJSON object according to the Route Exchange Model, including an overview of the whole route and a description of the road segments with properties indicating the name of the road, the distance (in meters), the duration (in seconds) and the speed limit (including a unit specifier).

Consideration of the acceleration/deceleration time, stop signs and traffic lights as contributing to the route duration was deferred for a future version.

Parameters specifying a preference for fastest or shortest route, additional waypoints, as well as weight or height restrictions were also supported, while the ability to specify obstacles to avoid was planned for future improvements.

The engine was also designed to take into account restrictions such as one-way streets and roads not accessible to motor vehicles.

In order to achieve optimal performance and complement Ecere's GNOSIS geospatial software technology, the Ecere Routing Engine was written using the eC programming language [http://ec-lang.org] and compiled natively. The engine was also designed with scalability considerations in mind, such as the ability to distribute the computation of roads to parallel threads running on different CPU cores.

8.3.2.2. Encountered Challenges and Solutions

The roads network was assembled from original OpenStreetMap data sourced from the `.osm.pbf` format. First, the Protocol Buffer encoding was decoded into multiple vector data layers, stored in a GNOSIS data store consisting of a series of tile pyramids (each containing multiple GNOSIS Map Tiles encoding of the features), as well as associated attributes stored in SQLite databases. This data store was also used by the Ecere client for visualizing the OpenStreetMap overlays.

One of these resulting layers contains (Multi)LineStrings vector features representing all roads. A network graph is then constructed from these roads, with nodes at the end of each segments, in effect more closely resembling the original OpenStreetMap data model consisting of *Nodes* and *Ways* than the Simple Features representation. This network construction required a very minimal

amount of preprocessing and can be done quickly on-the-fly, but a much more optimized version of the engine may later be a little more involved.

GeoPackages of the roads network sourced from the OpenStreetMap dataset were produced by Ecere. However, the pilot participants decided that the definition of a standard GeoPackage layout for use by routing engines was out of scope given the compact schedule. As a standardized format was not yet defined, and since GeoPackage was neither the source data format used by our engine, nor the representation that the routing engine directly works with, the functionality to build the road network graph from GeoPackage was not prioritized. Considerations for such a format however are included in the recommendations section, and once a consensus is reached, support for implementing that import process will be added to the engine.

As OpenStreetMap data is crowd-sourced by a very diverse global community. As such there sometimes is a lack of consistency in how features are tagged. In order to properly implement road access restriction, an analysis of the tags available for the source data had to be conducted, in conjunction with consulting the OpenStreetMap wiki entries describing how the tagging should be done and how to interpret it.

In addition to the tags considered while selecting the nodes and ways to include as part of the roads network data (e.g. the presence of the 'highway' tag), other OpenStreetMap tags were also taken into account for returning the name of roads, the speed limit, height and weight restrictions, as well as access restrictions such as one-ways roads and usage type. Although all routing experiments so far were only done for motor vehicles, road access was categorized based on whether pedestrians, bicycles and/or motor vehicles could use any given way. The following summarizes the OpenStreetMap tags taken into consideration and their interpretation by the Ecere Routing Engine. It is to be noted that this is still at a very early experimental stage, and may be partially incomplete or wrongly interpreted.

highway

The presence of the highway tag is used to identify an OpenStreetMap *Way* as being part of the roads network graph. Default usage access and speed limits are also associated to the different values for this key, but could be overridden by other tags.

Value	Pedestrian	Cycling	Motor vehicles	Speed limit
motorway	No	No	Yes	65 mph
motorway_link	No	No	Yes	65 mph
primary	Yes	Yes	Yes	50 mph
primary_link	Yes	Yes	Yes	50 mph
secondary	Yes	Yes	Yes	45 mph
secondary_link	Yes	Yes	Yes	45 mph
construction	Yes	Yes	Yes	45 mph
tertiary	Yes	Yes	Yes	40 mph
tertiary_link	Yes	Yes	Yes	40 mph
service	Yes	Yes	Yes	30 mph
services	Yes	Yes	Yes	30 mph

Value	Pedestrian	Cycling	Motor vehicles	Speed limit
residential	Yes	Yes	Yes	25 mph
living_street	Yes	Yes	Yes	20 mph
cycleway	No	Yes	No	N/A
footway	Yes	No	No	N/A
pedestrian	Yes	No	No	N/A
steps	Yes	No	No	N/A
track	Yes	Yes	No	N/A
path	Yes	Yes	No	N/A
(other)	Yes	Yes	Yes	30 mph

oneway

If specified and the value is anything other than 'no', the road is understood to be traversable in a single direction. Separate one-way restrictions for different road usage types (pedestrian, cycling, motor vehicles) were not yet implemented. Note that value of '-1' and 'reversible' were encountered in the dataset, but not dealt with specifically.

maxspeed

If provided, the value replaces the default speed unit. This element is currently being interpreted as miles/hour unless it contains the letters 'km'. Ideally, the default would be based upon the country in which routing occurs.

maxheight

Interpreted as the maximum height for a vehicle to be allowed access to a road. If this element contains the letter 'm', height is interpreted as meters; otherwise the height is specified in feet, and numbers following a single quote are interpreted as inches.

maxweight

Interpreted as the maximum weight for a vehicle to be allowed access to a road. If the element contains the letters 'lbs', it is interpreted as pounds; if the element contains the letters 'kg' it is interpreted as kilograms; otherwise the element is interpreted as US tons.

cycle, bicycle, cycleway

If the values for any of these is 'yes' or empty, cycling access is enabled. If the value is 'no' or 'private', it is disabled. If the value is 'designated', cycling access is enabled, while pedestrian and motor vehicle access is disabled.

steps

The presence of this tag disables cycling and motor vehicle access, and enables pedestrian access.

foot, footway

If the values for any of these is 'yes' or empty, pedestrian access is enabled. If the value is 'no' or 'private', this capability is disabled. If it is 'designated', pedestrian access is enabled, while cycling and motor vehicle access is disabled.

vehicle, motor_vehicle, motorcar

If the values for any of these is 'yes' or empty, motor vehicle access is enabled. If the value is 'no' or 'private', this capability is disabled. If the value is 'designated', motor vehicle access is enabled, while cycling and pedestrian access is disabled.

bus, taxi

If the values for any of these is empty, 'yes', or 'designated', motor vehicle, cycling and pedestrian access is disabled.

railway

If the values for any of these is 'yes', or empty, motor vehicle, cycling and pedestrian access is disabled.

access

If the value is 'no', 'private', or 'restricted', motor vehicle, cycling and pedestrian access is disabled.

Consideration of more specific restrictions (e.g. "no left turn"), as described by the "restriction" type of OpenStreetMap relations, have not yet been implemented. Where properly mapped in the data set, this offers full detailed description of different types of restrictions, including time-dependent (hourly and daily) restrictions. Some restrictions are described as 'from' and 'to' ways, while others are 'via' a node or a way.

OpenStreetMap describes the Relation:restriction [https://wiki.openstreetmap.org/wiki/Relation:restriction].

8.3.2.3. Final Component Description

Ecere provided a "bare-bones" online routing API which processes routing requests using the Ecere Routing Engine. Other participants also deployed Open Routing APIs implementations which passed routing requests to the Ecere Routing Engine through this endpoint.

In addition to the online endpoint, the Ecere Routing Engine was also made available to other participants as a Command Line Interface executable for Linux and Windows platforms. This enabled its use in offline scenarios by other client applications, but could also have avoided an extra server roundtrip in other routing endpoints, resulting in a faster response time. A library version of the engine was also proposed, which would be optimal in a server-side integration for loading the road network only once, then issuing multiple routing requests to the engine.

An important current limitation is that when identifying start, end and intermediate waypoints nodes, the engine snaps to nodes in the graph as opposed to being able to use a point on the road in-between two nodes. This can result in snapping to a street closer to the requested point, and not an intersection further away, while the ideal approach would be supporting calculations to that exact position on the road as requested.

In addition to the main test area of Washington D.C., experiments were also done with larger amounts of data from OpenStreetMap, including the entire state of North Carolina, as well as large regions in Japan. These larger datasets allowed to put the routing engine’s performance to a more stringent test and really showcase potential. Despite the early stage of development, the performance has proved to be quite satisfactory.

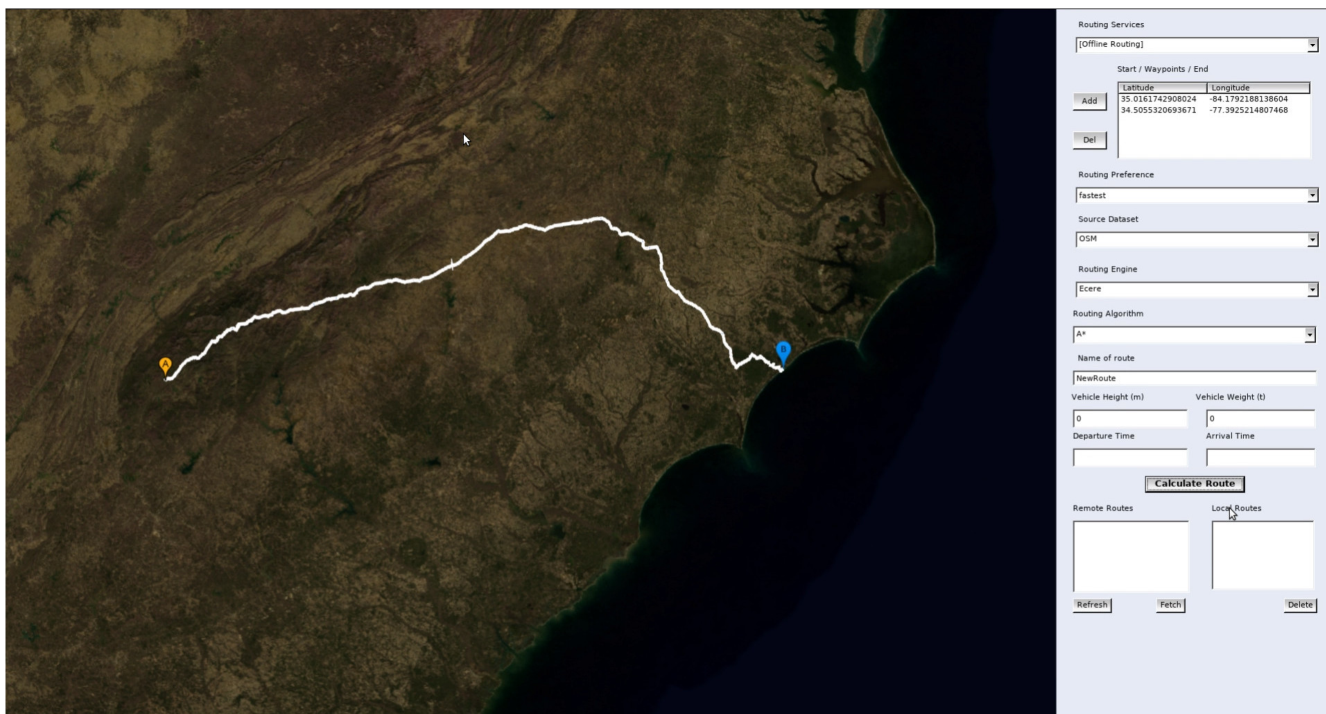

Figure 36. Ecere 3D routing client, showing route calculation across North Carolina (Blue Marble Next Generation)

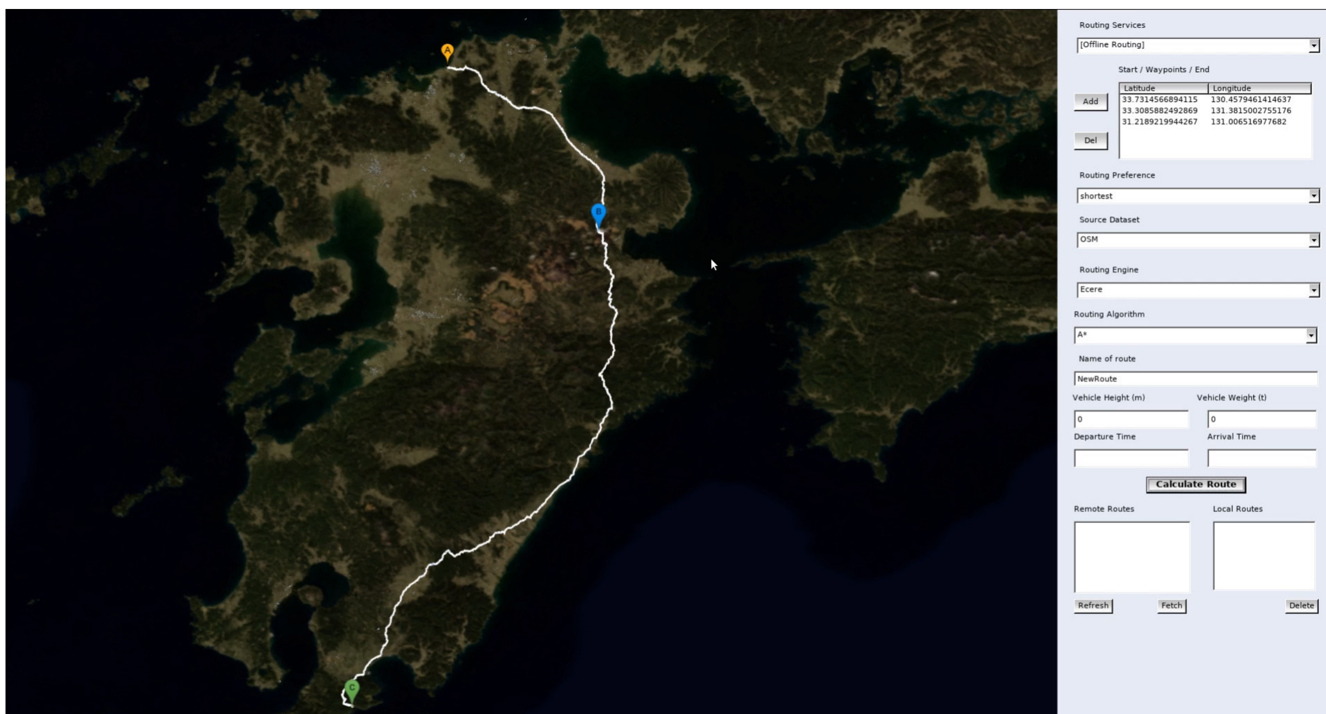

Figure 37. Ecere 3D routing client, showing route calculation across Kyushu island in Japan, including waypoint (Blue Marble Next Generation)

Figure 38. Ecere 3D routing client, showing area around start point of route in Kyushu island, Japan (Google Maps, OSM)

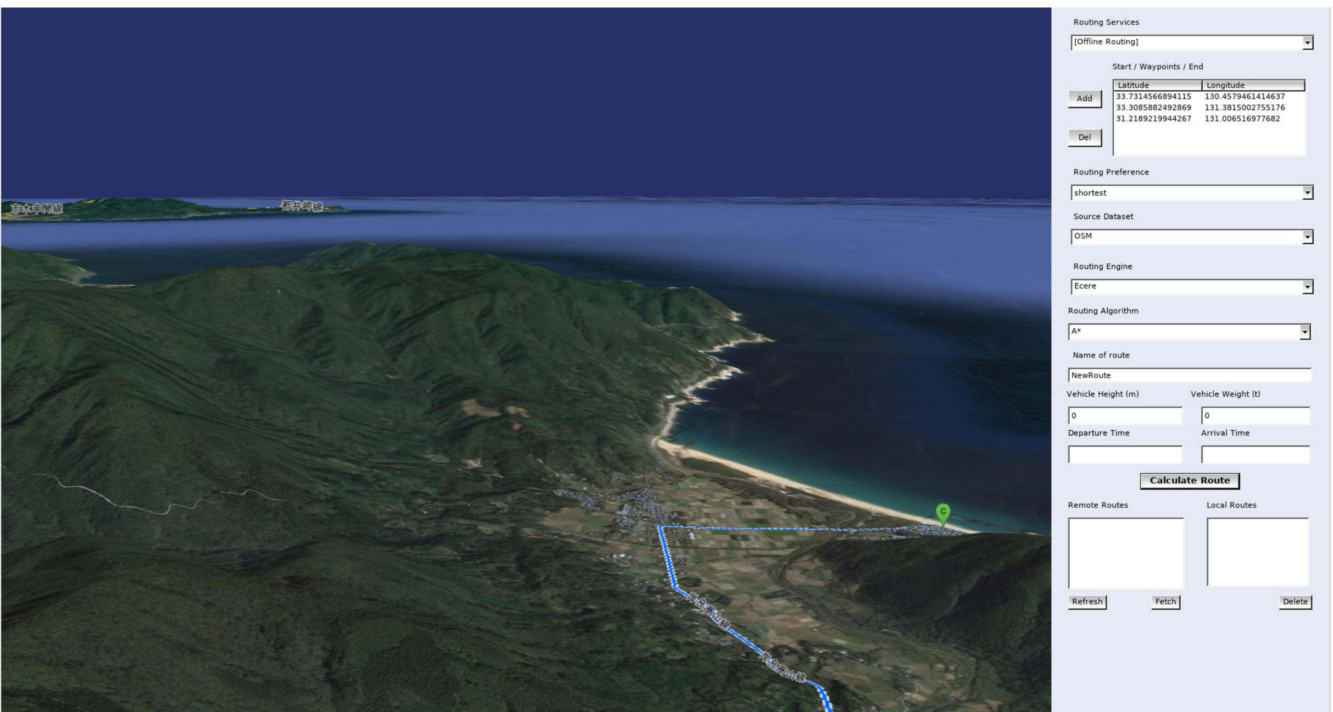

Figure 39. Ecere 3D routing client, showing area around end point of route in Kyushu island, Japan (Google Maps, OSM)

The limited availability of computing resources in mobile and embedded environments, as well as under heavy volume of requests in processing servers, is mitigated by the fact that the Ecere Routing Engine was designed with high performance in mind.

A further optimized version of the engine should be able to achieve even better performance and leverage multiple CPU cores, with an end goal of calculating continent-wide routes from a worldwide OpenStreetMap dataset in just a few milliseconds.

An improvement planned to be achieved in future activities is taking elevation from a Digital Elevation Model into account for calculating energy efficient routes.

Chapter 9. Technology Integration Experiments (TIEs)

The tables in this section record the TIEs conducted during the Pilot. Due to the sheer number of potential combinations between client, API, Routing Engine and dataset, along with a broad range of potential inputs these TIEs record the minimum successful interaction between the various components.

This minimum functionality was determined by defining a set of test parameters for the participants to use within the designated area of the District of Columbia, Washington DC. These test parameters for creating a route are as follows:

- Start point is the National Cathedral.
- End point is the Washington Monument.
- Preference should be Shortest.
- Method used for route retrieval is sync.
- Algorithm used can be any, as the HERE API does not support algorithm as a choice.
- Result type is Full.

For further information on what each of these input parameters represent see OGC 19-040.

The below TIE tests are separated by API, followed by specific comments for these TIE tests. <1 = success, 0 = failure, X = unfinished, -1 = not applicable>

Table 1. Skymantics API TIE Tests

API/Engine/Data Profile	Skymantics Client	Helyx Client	GIS.FCU Client	Ecere Client
Skymantics/Skymantics/OSM	1	X	1	1
Skymantics/Skymantics/NSG	1	X	1	1
Skymantics/Skymantics/HERE	1	X	1	1
Skymantics/Ecere/OSM	1	X	1	1
Skymantics/HERE/HERE	X	X	1	1

Comments

- The Skymantics API supported all Engines and data profile combinations. Certain tests could not be completed due to the API not supporting the conformance classes. These were optional for the Pilot but make up part of the OGC Common API. Therefore clients relying on conformance classes to construct their interface, specifically the Helyx client, could not be tested against the Skymantics API.

Table 2. Helyx API TIE Tests

API/Engine/Data Profile	Skymantics Client	Helyx Client	GIS.FCU Client	Ecere Client
Helyx/Skymantics/OSM	1	1	1	1
Helyx/Skymantics/NSG	1	1	1	1
Helyx/Skymantics/HERE	X	1	1	1
Helyx/Ecere/OSM	1	1	1	1
Helyx/HERE/HERE	X	1	1	1

Comments

- The Helyx API supported all Engines and data combinations.

Table 3. GIS.FCU API TIE Tests

API/Engine/Data Profile	Skymantics Client	Helyx Client	GIS.FCU Client	Ecere Client
GIS.FCU/Skymantics/OSM	X	1	1	1
GIS.FCU/Skymantics/NSG	X	1	1	1
GIS.FCU/Skymantics/HERE	X	1	1	1
GIS.FCU/Ecere/OSM	X	1	1	1
GIS.FCU/HERE/HERE	X	1	1	1

Comments

- The GIS.FCU API supported all Engines and data combinations.

Table 4. 52North API TIE Tests

API/Engine/Data Profile	Skymantics Client	Helyx Client	GIS.FCU Client	Ecere Client
52N/Skymantics/OSM	X	1	X	1
52N/Skymantics/NSG	X	1	X	1
52N/Skymantics/HERE	X	1	X	1
52N/Ecere/OSM	X	1	X	1
52N/HERE/HERE	X	1	X	1

Comments

- The 52North API supported all Engines and data combinations.

Table 5. Ecere API TIE Tests

API/Engine/Data Profile	Skymantics Client	Helyx Client	GIS.FCU Client	Ecere Client
Ecere/Ecere/OSM	X	X	1	1

Comments

- The Ecere routing engine endpoint has been included in the TIE test records, even though it was not intended to be a fully compliant routing API implementation. Some of the clients could not be tested directly against it, as it only provides a single routing path at which to POST a routing request and receive the computed route back, in the Routing Exchange Model format.

There are many more potential testing combinations than those recorded above. This is due to the wide range of input parameters to choose from when creating a route, all of which have impact on the nature of resulting output. A few comparisons are described below with accompanying screenshots to demonstrate this variety.

A clear contrast can be drawn between the different engine outputs. In this case all other inputs were left the same, and the same route was created using each engine. However, it must be noted that the Skymantics and HERE engines are using the HERE dataset in order to create this route. In contrast the Ecere engine only supports OSM as an input. In addition the HERE engine uses a bespoke algorithm rather than A*, which was used with the Ecere and Skymantics engine.

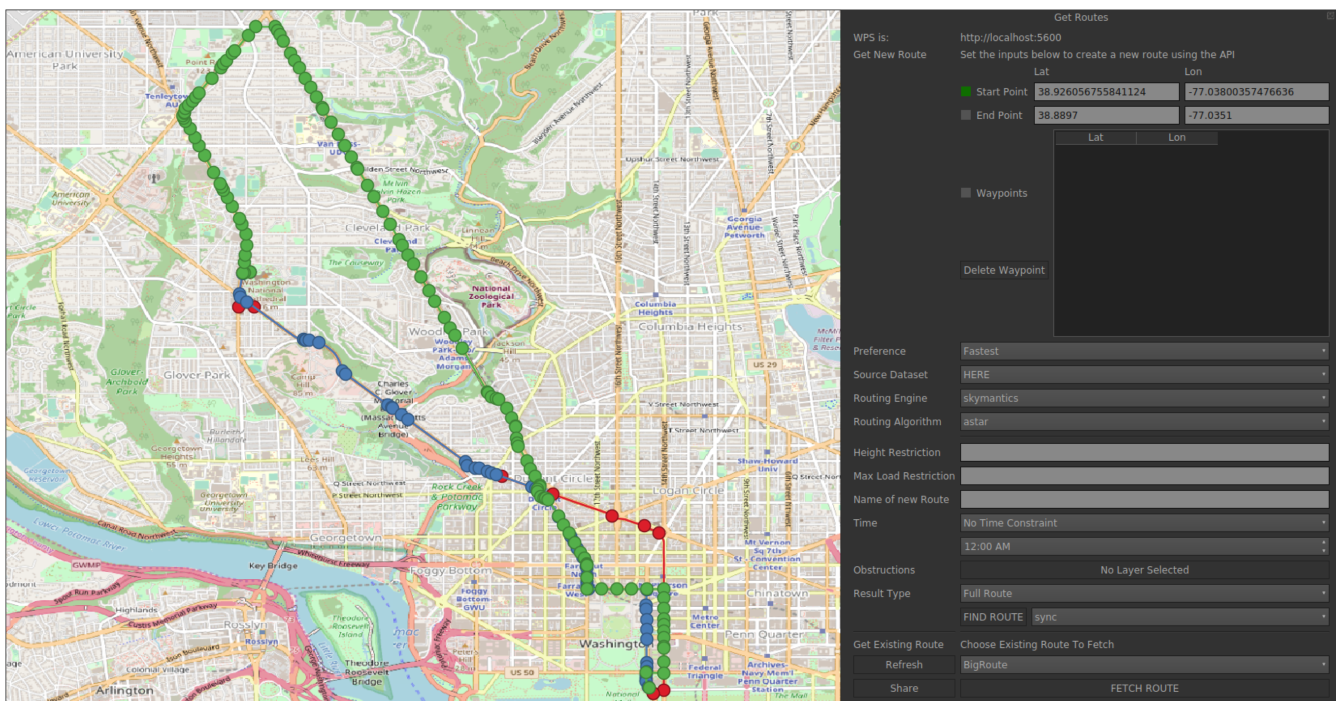

Figure 40. Routing Engines comparison

The three routes differ due to differences in the engines' construction. The engine may or may not:

- Use different weighting of road priority;

- Account for speed limit;
- Account for traffic;
- Account for road restrictions;

These specific engine considerations are further investigated in the implementation sections (Routing Engine Implementations).

Another key comparison can be drawn when changing the routing 'Preference' input. This can be done for each engine. In the below images the fastest routes are denoted by green nodes and edges and the shortest routes are denoted by red nodes and edges.

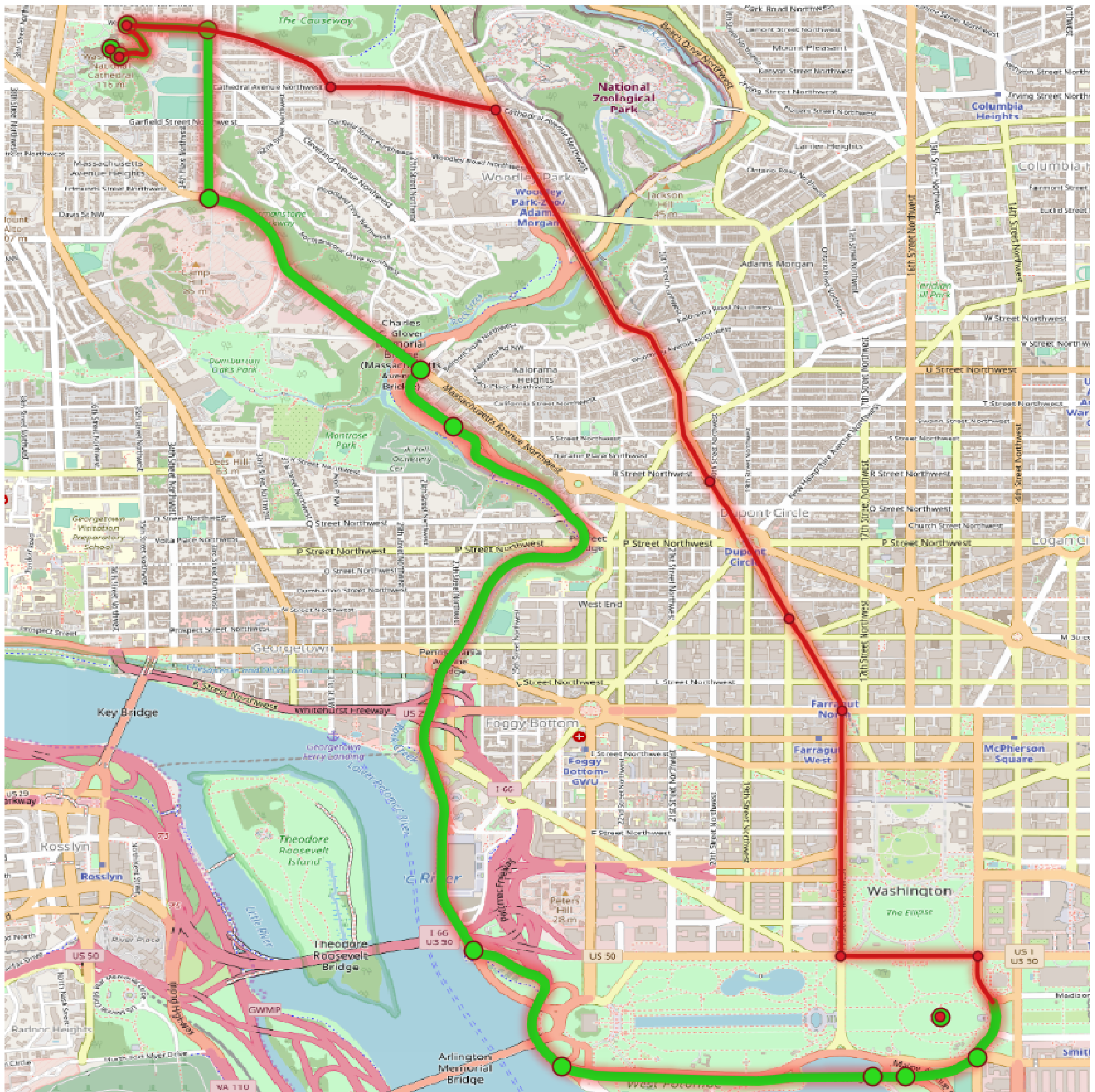

Figure 41. Here Routing Engine Preference comparison

Figure 42. Skymantics Routing Engine Preference comparison

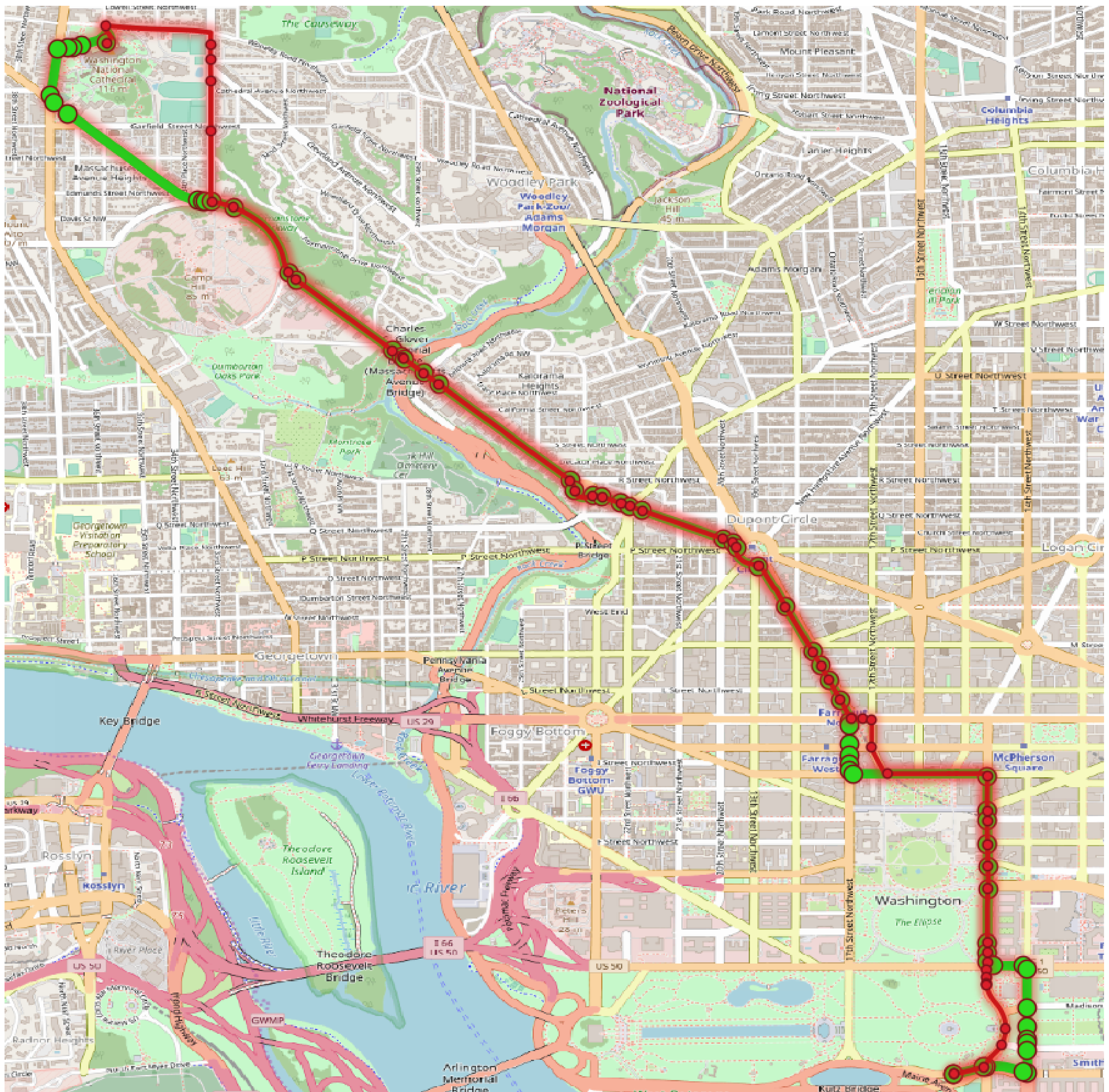

Figure 43. Ecere Routing Engine Preference comparison

The method used by the engines to account for the users preference varies along with the data used to produce the route.

In addition to this broad variety in online functionality, a number of participant components support offline functionality. These either used Ecere's routing engine, which had been converted into a Command Line Interface program, or the Open Source Routing Machine (OSRM), an open source and freely available routing engine.

Both the Ecere Routing Engine and OSRM use OSM data to calculate their routes, therefore all routes created offline were derived from OSM data.

The following clients support offline functionality:

- The Ecere desktop client successfully creates offline routes using the Ecere Routing Engine.
- The GIS-FCU web client successfully creates offline routes using the Ecere Routing Engine.

- The Helyx QGIS client successfully creates offline routes using the Ecere Routing Engine and the OSRM.

A few clients support the conversion of the GeoJSON route exchange model to GeoPackage for the sharing of routes:

- The Helyx QGIS client successfully converts fetched routes to GeoPackages.

Chapter 10. Pilot Recommendations

This section summarizes recommendations resulting from the execution of this OGC Pilot. Further explanation and supporting evidence of these recommendations are found throughout this ER.

10.1. OGC Web Interface Recommendations

These are recommendations that are applicable to OGC Web interfaces in general and are not constrained to Routing capabilities.

- Consider the Conformance class approach.
- The exposure of resources: [Reference Collections endpoint, chunking data and oversized arrays]
- Consider that the currently draft routing API (/routes) vs. traditional WPS (/processes/jobs) approaches could be harmonized by adjusting the draft OGC API - Processes draft specification, while always leaving the option to the service implementation not to expose the processing aspect (e.g. the /processes or equivalent resource), which could still be used in the background. This could apply equally well to other processing capabilities, such as server-side rendering (OGC API - Maps). OGC API - Processes adjustments which would enable this include:
    - Support flexible resource paths being described in /processes (e.g. which could link to /routes)
    - Support synchronous and/or asynchronous requests
    - Support persistent, semi-persistent and/or temporary processing output (results) resources

10.2. OGC Routing Recommendations

10.2.1. Skymantics Recommendations

- **Dataset combination**: An interesting approach would be to use different datasets not as separate silos for route comparisons, but as complementary sources of routing information. This Pilot has proven that there are differences among datasets that lead to substantially different routes for the same origin and destination. The Pilot also demonstrated that there is not one dataset that is *better* in all circumstances as they each have their advantages and benefits. For example, OSM has the highest street capillarity whereas HERE has the deepest detail in traffic rules. It should be feasible to build a routing engine that is able to generate routes based on the combined inputs from different datasets. This is a data conflation issues that has been discussed in several OGC endeavors.
- **Restricted maneuvers** are a useful feature included in the HERE dataset. However, the HERE dataset is not perfect, as some maneuvers that should be restricted are not specified (thus allowed), which means they are ignored by the routing engine. It should be technically possible to evaluate all the possible maneuvers in a dataset and mark those potentially forbidden for manual revision, or even restrict those that are obvious (for example, a turn left in a freeway). This suggestion is not limited to the HERE dataset.
- **Speed limits** are fundamental to estimate the duration of a route. In reality, the estimation of a

speed in a particular street or road depends solely on the status of the traffic at each moment. In order to improve accuracy for estimations, or to find the fastest route more accurately, or to suggest alternatives in case of congestion, counting on real-time traffic information, at least for some main streets. This could be a web service offered by Smart Cities based on information derived from traffic cameras, for example.

  - **Street hierarchies** are essential to apply optimization methods, such as Contraction Hierarchies. But in an urban area, with a dense mesh of streets, statically defined hierarchies such as those defined in a dataset are not enough information to properly categorize them. Additional statistics of road traffic in each street, such as number of vehicles per day, would help find those streets that in theory belong to a low hierarchy but in reality are used by locals as shortcuts and should be treated as high hierarchy links.

10.2.2. Helyx Recommendations

These recommendations are applicable to future OGC routing work and include supported inputs, Route Exchange Model considerations and specific routing API considerations.

- Additional routing request inputs to consider
  - Transport method
  - Account for traffic
- Performance testing of the APIs and the Routing Engines
  - Performance testing of the routing service to gauge the efficiency of the API structure and identify bottlenecks.
  - Performance testing of the routing engines to provide improved capability, test instantaneous responses from the APIs and explore asynchronous and multi-threaded execution.

Future improvements to the GUI application based on QGIS could include:

- The display of route instructions in QGIS popups or map tips.
- The inclusion of a tracking processing service to allow for the monitoring of vehicles along routes. This would be complemented by QGIS's feature creation capabilities to construct *geofences* for alerting and showing new obstacles during transit.
- The manual creation of routes in QGIS and the subsequent capability to validate and upload routes to either the OGC Routing API or the OGC API Collections endpoint

10.2.3. Ecere Recommendations

- Defining a usage type parameter (e.g. pedestrians, cycling, motor vehicles)
- Defining other restrictions such as avoiding ferries
- Support for an energy-efficient preference, taking terrain elevation into account (using a digital terrain elevation model)
- Further evaluation of how OpenStreetMap tags and relations are considered to establish restrictions (e.g. Relation:restriction [https://wiki.openstreetmap.org/wiki/Relation:restriction])

- Defining a standardized GeoPackage layout (or utilizing an existing model) to describe a roads network and associated required properties necessary for routing:
  - This could mainly consist of a Point features table for nodes (such as at intersections), and a LineString features table for roads, along with standardized attributes describing road names, speed limit and usage restrictions. Normalizing the roads attributes would greatly facilitate the work of routing engines, as OpenStreetMap data being crowd-sourced often presents irregularities in the tags specified. This standard GeoPackage roads network layout could enable multiple routing engines to work with a pre-assembled roads network, which itself could be sourced from any data sources.

Chapter 11. Conclusions

The OGC Routing API Pilot has begun devising an approach to interoperable access to routing services by leveraging the next generation of OGC web service interface definitions. The preliminary Routing API specification and Routing Exchange Model defined during this Pilot provide the foundation for a future routing standard to be used alongside a common OGC API. This foundation could be used in the future to integrate disparate routing engines, services and clients allowing for interoperable ways of working in a broad range of user communities.

By focusing on OpenAPI approaches, using JSON as an exchange format and accounting for a broad range of potential user inputs, the pilot implementations have demonstrated a standards-based approach can greatly improve the challenge of interoperability.

The Routing API standard, designed using Swagger Hub, provided the pilot with a single point of reference for all component implementations. The flexible nature of the API provides clear definitions for minimum requirements so that services can be compliant without the need to implement every API element. The same approach was taken with the Routing Exchange model, which provided a single point of truth for all components to rely on. This made the TIE tests trivial from a data exchange perspective, as every component supported the Route Exchange Model.

A number of areas for further investigation have been identified for the consideration of future work. These include, but are not limited to, further focus on conformance class implementation, further integration with the OGC Common collections endpoints and consideration for other user inputs.

The primary question not answered by this pilot regards the future of WPS and OGC API - Processes. That is, should the API support a WPS oriented approach with OGC defined definitions for inputs, and a URI structure that is similar to the WPS 2.0 or should the API support an OpenAPI approach with input definitions defined mostly by OpenAPI and URI patterns rather than definitions. These two approaches may not be mutually exclusive but any attempt to merge the two would need to consider how both handle API resources, interaction with other OGC standards and what constitutes a *process*.

Appendix A: Routing Exchange Model

A.1. Overview

A proposed Routing Exchange Model was required to standardize the creation and transfer of route information between a variety of components in the Pilot. A specific model was required due to the requirements for routing in the three operating scenarios:

- Connected
- Intermittent
- Disconnected

The primary constraint on defining this initial format was a Sponsor requirement that GeoJSON be used. As such, there is not a conceptual component. This has allowed the Pilot participants to focus on the implementation of the format rather than spending time on discussions of a more conceptual model. However, consideration was given to making the format flexible in case it can be used to develop a more format independent variant or be the basis for a conceptual model in the future.

The format supports three variants that map directly onto the possible result choice provided by the Routing API.

These variants include:

- Full: All route information is encoded in a GeoJSON feature collection. This is the default model and must be supported by all Routing API server-side and client-side components.
- Overview: A single GeoJSON feature detailing the route geometry along the network and the main properties of the route.
- Segments: The first segment of a route and a link to the second segment, if there is more than one segment in the route. Subsequent segments will contain a link to the previous and next segment. The final segment contains only a link to the previous segment. Each segment is a GeoJSON feature.

All implementations need to support the first variant (Full). The other two variants provide additional functionality to support a variety of use cases, specifically the Denied, Degraded, Intermittent and Low Bandwidth (DDIL) network use cases. Both options allow the user to retrieve the minimal information required for their use case. This could be the current step of the route, or an overview to assess the suggested route. In both cases only small GeoJSON objects are passed over the DDIL network. These variants also provide the option to conduct a comprehensive assessment of routes thereby allowing the user to request a number of route overviews in quick succession, choose the route which is most preferable and then reuse the route definition of the chosen route to request the full route.

These variants relate to the Routing API (OGC 19-040) route definition element. The route definition allows the user to choose the variant they wish to request, and it is the primary method by which a client requests a route via POST request to the OGC Routing Process API.

The section below specifies the requirements classes that implementations need to conform to.

A.2. Requirements class "Route Exchange Model (core)"

This requirement class states requirements that apply to all representations of a route.

The normative statements use the JSON Schema variant defined by OpenAPI 3.0 to specify schema components.

Requirements Class	
http://www.opengis.net/orp/routing-api/1.0/req/rem	
Target type	JSON object
Dependency	GeoJSON [https://tools.ietf.org/rfc/rfc7946.txt]
Dependency	Date and Time on the Internet: Timestamps [https://tools.ietf.org/rfc/rfc3339.txt]

Requirement 1	**/req/rem/geojson**
A	Every representation of a route SHALL be a valid GeoJSON object.

Requirement 2	**/req/rem/date-time**
A	Every representation of a timestamp SHALL be a valid `date-time` value according to RFC 3339, 5.6 [https://tools.ietf.org/html/rfc3339#section-5.6].

A.3. Requirements class "Route Exchange Model (full)"

This requirements class specifies the complete representation of a route, i.e., a representation that includes all information about the route.

Requirements Class	
http://www.opengis.net/orp/routing-api/1.0/req/rem-full	
Target type	JSON object
Dependency	Requirements class "Route Exchange Model (core)"
Dependency	Requirements class "Route Exchange Model (overview)"
Dependency	Requirements class "Route Exchange Model (segment)"

Requirement 3	**/req/rem-full/fc**

A	The complete representation of a route SHALL be a valid GeoJSON feature collection.

Recommendation 1	**/rec/rem-full/fc-name**
A	The feature collection SHOULD have a member with the name "name" that is a title of the route with the following OpenAPI 3.0 schema: `type: string`

Typically the name will be provided by the requester of the route.

Requirement 4	**/req/rem-full/fc-status**

A	The feature collection SHALL have a member with the name "status" describing the processing status of the route with the following OpenAPI 3.0 schema:
	`type: string`
	`enum:`
	`  - accepted`
	`  - running`
	`  - successful`
	`  - failed`

The values are defined as follows:

accepted
: The routing job is queued for execution.

running
: The route is being computed.

successful
: The route is available.

failed
: The route could not be computed.

Requirement 5	**/req/rem-full/fc-features**

A	The feature collection SHALL contain the following features, depending on the status: • a route overview (see Requirements class "Route Exchange Model (overview)", only for status `successful`) • the start point of the route (see Requirement "start point") • the end point of the route (see Requirement "end point") • one or more segments (see Requirements class "Route Exchange Model (segment)", only for status `successful`)
B	The sequence of the segments SHALL be in their order along the route.

Recommendation 2	**/rec/rem-full/fc-links**

A	The feature collection SHOULD have a member with the name "links" with the following OpenAPI 3.0 schema:
	
	`type: array`
	`items:`
	`  type: object`
	`  required:`
	`    - href`
	`  properties:`
	`    href:`
	`      type: string`
	`    rel:`
	`      type: string`
	`    type:`
	`      type: string`
	`    hreflang:`
	`      type: string`
	`    title:`
	`      type: string`

B	There SHOULD be a link with • `rel` with value `self` • `type` with value `application/geo+json` • a URI to fetch the route in `href`

C	There SHOULD be a link with • `rel` with value `describedBy` • `type` with value `application/json` • a URI to fetch information about the definition of the route (start and end point, constraints) in `href`

Requirement 6	**/req/rem-full/start**
A	The start point of the route SHALL be a GeoJSON feature with a Point geometry.
B	The feature SHALL have a property `type` with the value `start`.
C	The point geometry of the feature SHALL depend on the status of the route: • "successful": identical to the first point of the route overview. • otherwise: identical to the start point in the definition of the route.
D	If the feature has a property `timestamp`, it SHALL be of type `string`, format `date-time`, and indicate the (estimated) departure time.

Requirement 7	**/req/rem-full/end**
A	The end point of the route SHALL be a GeoJSON feature with a Point geometry.
B	The feature SHALL have a property `type` with the value `end`.
C	The point geometry of the feature SHALL depend on the status of the route: • "successful": identical to the last point of the route overview and identical to the point in the last segment. • otherwise: identical to the end point in the definition of the route.
D	If the feature has a property `timestamp`, it SHALL be of type `string`, format `date-time`, and indicate the (estimated) arrival time.

A.4. Requirements class "Route Exchange Model (overview)"

Requirements Class	
http://www.opengis.net/orp/routing-api/1.0/req/rem-overview	
Target type	JSON object
Dependency	Requirements class "Route Exchange Model (core)"

Requirement 8	/req/rem-overview/feature
A	The route overview SHALL be a GeoJSON feature with a LineString geometry.
B	The feature SHALL have a property `type` with the value `overview`.
C	The line string geometry of the feature SHALL be the path from the start point to the end point of the route.
D	The feature SHALL have a property `length_m` (type: `number`) with the length of the segment (in meters).
E	The feature SHALL have a property `duration_s` (type: `number`) with the estimated amount of time required to travel the segment (in seconds).
F	If the feature has a property `maxHeight_m`, the value SHALL be of type `number` with a known height restriction on the route (in meters).
G	If the feature has a property `maxLoad_t`, the value SHALL be of type `number` with a known load restriction on the route (in tons).
H	If the feature has a property `obstacles`, the value SHALL be of type `string` and describe how obstacles were taken into account in the route calculation.
I	If the feature has a property `processingTime`, it SHALL be a `date-time` as specified by RFC 3339, 5.6 [https://tools.ietf.org/html/rfc3339#section-5.6] and state the time when the route was calculated.

J	If the feature has a property `comment`, the value SHALL be of type `string` and explain any minor issues that were encountered during the processing of the routing request, i.e. any issues that did not result in an error.

Recommendation 3	**/rec/rem-overview/properties**
A	The route overview SHOULD have the property `processingTime`.
B	If the API has access to the information, the route overview SHOULD have the properties `maxHeight_m`, `maxLoad_t`, and `obstacles`.

A.5. Requirements class "Route Exchange Model (segment)"

Requirements Class	
http://www.opengis.net/orp/routing-api/1.0/req/rem-segment	
Target type	JSON object
Dependency	Requirements class "Route Exchange Model (core)"

Requirement 9	**/req/rem-segment/feature**
A	Each segment of the route SHALL be a GeoJSON feature with a Point geometry.
B	The segment feature SHALL have a property `type` with the value `segment`.
C	The point geometry of the feature SHALL be the last position of the segment and be on the line string geometry of the route overview.
D	The feature SHALL have a property `length_m` (type: `number`) with the length of the segment (in meters).
E	The feature SHALL have a property `duration_s` (type: `number`) with the estimated amount of time required to travel the segment (in seconds).

F	The sum of all `length_m` values of segments SHALL be identical to the `length_m` value in the route overview.
G	The sum of all `duration_s` values of segments SHALL be identical to the `duration_s` value in the route overview.
H	If the feature has a property `maxHeight_m`, the value SHALL be of type `number` with a known height restriction on the segment (in meters).
I	If the feature has a property `maxLoad_t`, the value SHALL be of type `number` with a known load restriction on the segment (in tons).
J	If the feature has a property `speedLimit`, the value SHALL be of type `integer` with a known speed limit on the segment.
K	If the feature has a property `speedLimit`, the unit of the speed limit SHALL be specified in a property `speedLimitUnit`; the allowed values are `kmph` (kilometers per hour) and `mph` (miles per hour).
L	If the feature has a property `roadName`, the value SHALL be of type `string` with the road/street name of the segment.
M	If the feature has a property `roadName` and the feature is part of a response to a HTTP(S) request, the language SHALL be specified in the `Content-Language` header.
N	If the feature has a property `instructions`, the value SHALL be of type `string` with an instruction for the maneuver at the end of the segment. Allowed values are `continue`, `left` and `right`.

A.6. Requirements class "Route Exchange Model (segment with links)"

Requirements Class	
http://www.opengis.net/orp/routing-api/1.0/req/rem-segment-with-links	
Target type	JSON object
Dependency	Requirements class "Route Exchange Model (segment)"

Requirement 10	**/req/rem-segment-with-links/next-prev**

<table>
<tr><td>A</td><td>Each segment SHALL have a member with the name `links with the following OpenAPI 3.0 schema:
<pre><code>type: array
items:
  type: object
  required:
    - href
  properties:
    href:
      type: string
    rel:
      type: string
    type:
      type: string
    hreflang:
      type: string
    title:
      type: string</code></pre></td></tr>
<tr><td>B</td><td>Unless the segment is the last segment of the route, the segment SHALL have a link
• `rel` with value `next`
• `type` with value `application/geo+json`
• a URI to fetch the next segment along the route in `href`</td></tr>
<tr><td>C</td><td>Unless the segment is the first segment of the route, the segment SHALL have a link
• `rel` with value `prev`
• `type` with value `application/geo+json`
• a URI to fetch the previous segment along the route in `href`</td></tr>
</table>

Appendix B: Revision History

Table 6. Revision History

Date	Editor	Release	Primary clauses modified	Descriptions
August 27, 2019	T. Brown	.1	all	pre-review draft
September 04, 2019	T. Brown	.2	all	intial review draft