HETEROGENEOUS DATA INTEGRATION FOR OPERATIONS AND TRAVEL INFORMATION SHARING

by

Chang Liu

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

November 2014

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor, Dr. Qing Yang, whose expertise, support and patience added considerably to my graduate experience. I appreciate his tremendous guidance on both my academic study and personal life. Without his help I would not be able to continue my second year of graduate school and to get involved in this project. It is an honor to be his student and learn from him.

I express my warm thanks to our team members, especially Eben and John who help create the XML schemas, query the road segments and many other things. I would also like to thank Dr. Rafal Angryk who gave me the chance to study in the United States. I am grateful for his enormous support and advices.

TABLE OF CONTENTS

TABLE OF CONTENTS - CONTINUED

LIST OF FIGURES

Figure Page

ABSTRACT

The North/West Passage (N/WP) corridor follows I-90 and I-94 from Washington to Wisconsin. The Operations and Travel Information Integration Sharing (OTIIS) system provides traveler information on the eight state corridor-wide scales in a single website. This work presents the approach to ingest the heterogeneous data from the Departments of Transportation (DOT) of the eight states along the N/WP corridor, and from third party sources such as NOAA Forecast Database. This thesis details the process of fetching, parsing the data feeds, updating the database; introduces all the web services used in the project and describes how to resolve related geometry data issues. The valuable potential benefits of such data injectors would be not only feeding the OTIIS website with well formatted, real time travel information, but also facilitating the development of the API for potential use by other approved systems or developers.

INTRODUCTION

The North/West Passage (N/WP) corridor follows interstate 90 and interstate 94 from Washington to Wisconsin that spans 2000 miles. Drivers that are planning to travel a long distance on the corridor usually need to gather various information before the journey. Such information could include road work information, road closure information, crash information, congestion information, fuel station information, weather forecast information, weather alert information, rest area information, mountain pass information, temporary restriction information, etc. It is hard to get every single piece of the above information from separate data sources, especially the route spans a long distance on different states. To facilitate the travel planning on N/WP corridor, the Operations and Travel Information Integration Sharing (OTIIS) project provides information on the eight state corridor-wide scale in a single website. The corridor-wide traveler information system will not only benefit the use and safety of commercial vehicle operator who maybe the major user of the future system but also benefit recreational traveler.

A primary objective and challenge of the project is to ingest the heterogeneous data from the Department of Transportation (DOT) of each involved state and from third party data sources. Once the required data is fetched and parsed, it will be pushed into a centralized database. Finally it will be ready to feed the OTIIS website and facilitate the development of the API for potential use by other approved systems or developers.

SYSTEM OVERVIEW

To digest all data feeds from different sources, a series of data injectors are deployed at the backend. Data feeds could roughly be divided into three categories (traffic related data from DOT of member states, weather related data from National Oceanic and Atmospheric Administration (NOAA) data center and other relatively more static data from independent sources). The injectors work as a funnel to ingest all heterogeneous data in different format, and export a stream of data in a common standard to the database. Data is inserted and updated in database periodically by establishing connection between the injectors and the database. Then build on top of the Google Map API, users on either PC end or mobile end could send requests to access the traveler information through system API calls.

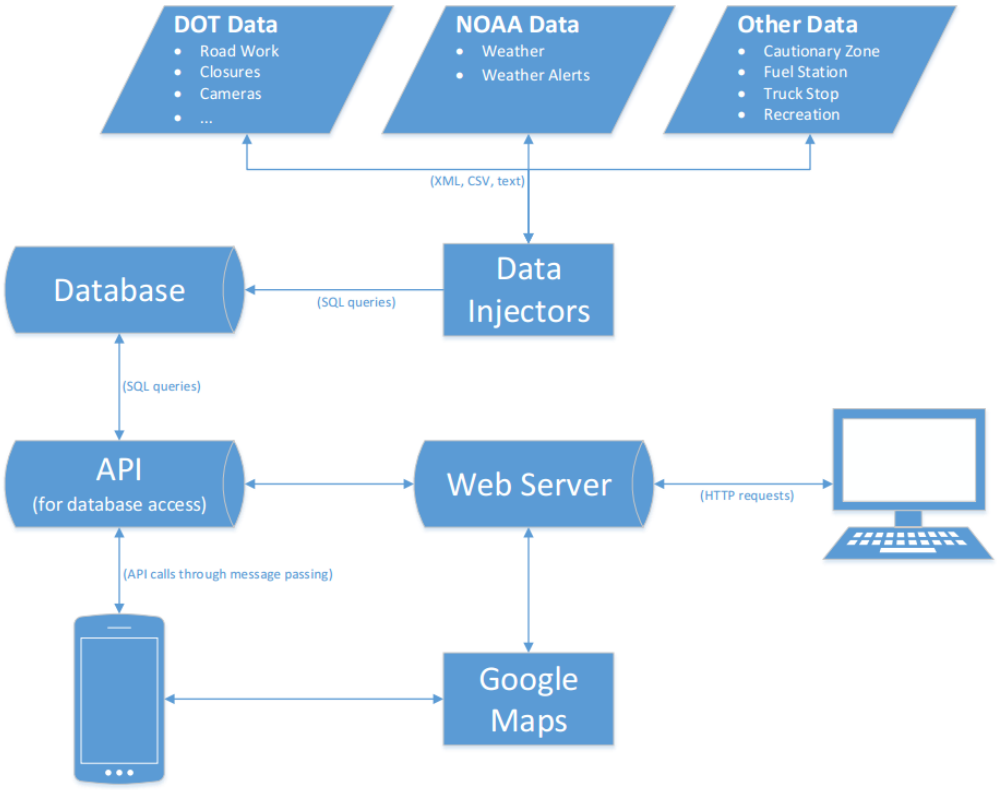


Figure 1 System Overview

Figure 1shows the architecture of the whole system. As we introduced, data comes from three main sources in the format of XML, CSV, plain text and others. The injectors are responsible for fetching the feeds periodically, parsing the data and pushing the trimmed data into spatial database by SQL queries. The API works between the front end and the back end. Data query requests are passed to the API and data streams are sent back to the front end by querying the database. In this paper, we detail the processes of building data injectors.

Required Layers Overview

This section describes an overview about the requirements of each system layers. To build the data injector properly, it is necessary to well understand the details of each required layers. The specific needed data and the corresponding data format of each layer are introduced below.

Road Work layer contains the status of the road work, specific location, begin and end date for the road work, expected impact and notes describing additional relevant details. The popup of road work layer on the OTIIS website has the following items:

- `Status: [active or planned]`
- `Dates: [MM/DD/YY to MM/DD/YY]`
- `Location: [mile posts] or [XX miles west of TOWN NAME]`
- `Notes: [resurfacing, striping, etc.]`
- `Impact: [low, medium, high]`

Similarly, *Incident/Crash* contains date and time stamp along with the location, description and travel impact of the incident. *Road Condition* contains the event's location, date time stamp, description and detailed condition. *Weather Forecast* contains the current and the next four days' location of the weather forecast, chance of precipitation, wind speed, visibility and temperature. Different from weather forecast, *Weather Alert* layer forecasts more serious weather alert information based on broader area, such as severe thunderstorm warning, flash flood warning. This layer contains items like date time stamp, location, conditions, forecast and details of alerts. *Road Closure* comes with date time stamp, location, closure description, the duration of the road closure. *Temporary Restriction* shows the restrictions status and its beginning and end date,

description of its location and restriction. *Mountain Pass* is a complex layer that combines the weather forecast, Road Weather Information System (RWIS) and camera together with the location of the pass. The required data items and format show as follows.

- Date: [MM/DD/YY]
- Time: [HH:MM]
- Location: [pass name]
- Max Grade: [XX%]
- Elevation: [XXXX feet]
- Camera: [button to show camera view if applicable]
- RWIS: [button to show RWIS data if applicable(temp, wind, precipitation, etc.)]
- Weather Forecast: [button to show weather and forecast weather for that location]

Historic Crash Trend contains the location, date and time stamp, and a short description of the type of incidents. *RWIS* shows the RWIS location, date time stamp, the site's elevation, air temperature, surface temperature, wind speed and visibility. *Camera Image* displays the camera's location, image date time stamp and camera image. *Traffic Congestion* shows the congestion's location with a date time stamp, and congestion description. *Truck Stop* layer contains the location, name, amenities and truck parking info. *Fuel Station* shows the location, name and amenities. The required data and format of Truck Stop layer show as follows.

- Location: [mile posts] or [town/area name]
- Name: [Flying J, etc.]
- Amenities: [showers, restaurant, etc.]
- Parking: [## truck spaces] or [S, M, L]

Rest Area is a similar static layer that shows the same information as the previous two, location, name and amenities. *Recreation Interests* shows the type, name, location, additional distance, additional drive time, description and a website link. *Truck Parking* shows the location, name, amenities and number of truck parking spaces. *Travel Time Comparisons w/ Alternate Routes* layer indicates the distance and expected travel time for each route.

Data Source and Format

In general, traffic related data comes from the data center of each state's DOT. Other layers come from third party data sources. Traffic related layers include *Incident, Road Work, Road Closure, Road Condition, Truck Restriction,* and *Traffic Congestion*. All of these layers are warped in the format of XML. And each state has its own schema designed to structure its data storage, though most of them more or less follow the guideline of Traffic Management Data Dictionary (TMDD). TMDD is a standard data exchange format defined by the institute of Transportation Engineers (ITE). It provides a broad array of data fields for describing roadway event and condition information (Engineers). Since TMDD is designed to support a comprehensive and completed description of a variety of events, not all of the TMDD fields are necessary for the website. The following section describes the format of the traffic related data feed from each member state.

Minnesota and Idaho

These two states have the same traffic events reporting system and have the same methodology to report event data. A TMDD formatted XML file works as data feed through HTTP, requiring a pair of username and password. A snapshot of data feed from Minnesota is shown below.

```
▼<FEUMessages>
  ▼<feu:full-event-update xmlns:feu="http://www.northamericanhub.org">
    ▼<message-header>
      ▼<sender>
          <organization-id>MNSEG</organization-id>
          <center-id>MNSEG</center-id>
        </sender>
        <message-type-version>1</message-type-version>
        <message-number>424312</message-number>
      ▼<message-time-stamp>
          <date>20141015</date>
          <time>055809</time>
          <utc-offset>-0500</utc-offset>
        </message-time-stamp>
      ▼<message-expiry-time>
          <date>20180318</date>
          <time>055809</time>
          <utc-offset>-0500</utc-offset>
        </message-expiry-time>
      </message-header>
    ▼<event-reference>
        <event-id>MNSEG-424312</event-id>
        <update>20</update>
      </event-reference>
    ▼<event-indicators>
      ▼<event-indicator>
          <status>updated</status>
        </event-indicator>
      ▼<event-indicator>
          <priority>10</priority>
        </event-indicator>
      </event-indicators>
    ▼<headline>
      ▼<headline>
          <pavement-condition>normal driving conditions</pavement-condition>
        </headline>
      </headline>
    ▼<details>
      ▼<detail>
          <element-id>1</element-id>
        ▼<descriptions>
          ▼<description>
```


Figure 2 Sample feed from Minnesota

A chunk of MN's XML feed is shown in figure 2. A field called "headline" defines different kinds of events, thus all of the traffic related layers mentioned above

(*Incident, Road Work, Road Closure, Road Condition, Truck Restriction, Traffic Congestion*) are contained in this single data feed.

Washington

Washington state designed a well-defined traveler information API to provide third parties with a single gateway to the layers of *Incident, Road Work, Road Closure, Road Condition, Truck Restriction, Traffic Congestion, Camera*. We access Washington's data through the web service of REST over HTTP with an assigned access code. The XML chunk below is a data feed sample from Washington.


```
▼<Alert>
   <AlertID>168099</AlertID>
   <County i:nil="true"/>
  ▼<EndRoadwayLocation>
     <Description>International Boundary</Description>
     <Direction>N</Direction>
     <Latitude>49.002105713</Latitude>
     <Longitude>-122.485054016</Longitude>
     <MilePost>15.15</MilePost>
     <RoadName>539</RoadName>
   </EndRoadwayLocation>
   <EndTime>2014-09-05T17:00:00</EndTime>
   <EventCategory>Debris</EventCategory>
   <EventStatus>Open</EventStatus>
   <ExtendedDescription/>
  ▼<HeadlineDescription>
     Wednesday, Sept. 3 to Friday, Sept. 5 - One lane of SR 539 will be closed near the border
     daily from 7 a.m. to 5 p.m. for pavement work. Flaggers will direct traffic through the
     work zone.
   </HeadlineDescription>
   <LastUpdatedTime>2014-08-28T13:31:32.843</LastUpdatedTime>
   <Priority>Medium</Priority>
   <Region>Northwest</Region>
  ▼<StartRoadwayLocation>
     <Description>H Street</Description>
     <Direction>N</Direction>
     <Latitude>48.993347168</Latitude>
     <Longitude>-122.485137939</Longitude>
     <MilePost>14.55</MilePost>
     <RoadName>539</RoadName>
   </StartRoadwayLocation>
   <StartTime>2014-09-03T06:30:00</StartTime>
 </Alert>
```


Figure 3 Sample feed from Washington

North Dakota

Instead of using a single integrated data feed for all traffic related layers, North Dakota assigns a XML file for each layer of *Incident, Road Work, Road Closure, Road Condition, Truck Restriction.* The format of each feed follows the version 3.0 of the TMDD.

Wisconsin

Wisconsin provides a TMDD format based XML feeds on HTTP server. Three separate feeds are provided, in which layers of *Incident, Road Work, Road Closure, Road Condition* are contained.

Montana

Montana provides three separate custom event XML data feeds on its FTP server. In those three data feeds, layers of *Incident, Road Work, Road Condition* are covered. One unique character about MT's data is instead of providing latitude/longitude pairs as location information, milepost is used as location information. We will discuss our method of mapping milepost to route segment in geometry type in the following chapters. The XML chunk below is a sample of data feed from MT.

```
<SEGMENT>
  <ROW_NUM>2</ROW_NUM>
  <SIGNED_ROUTE>I-90</SIGNED_ROUTE>
  <BEG_MP>221</BEG_MP>
  <END_MP>243</END_MP>
  <SEGMENT_DESC>6 TO 16 MILES EAST OF BUTTE OVER HOMESTAKE PASS</SEGMENT_DESC>
  <CONDITION>DRY/MOSTLY DRY</CONDITION>
</SEGMENT>
<SEGMENT>
  <ROW_NUM>3</ROW_NUM>
  <SIGNED_ROUTE>I-90</SIGNED_ROUTE>
  <BEG_MP>210.9</BEG_MP>
  <END_MP>221</END_MP>
  <SEGMENT_DESC>15 MILES WEST OF BUTTE TO BUTTE</SEGMENT_DESC>
  <CONDITION>DRY/MOSTLY DRY</CONDITION>
</SEGMENT>
```

Figure 4 Sample feed from Montana

Wyoming

Wyoming provides two separate custom event XML data feeds on its server. Layer of *Road Work, Road Closure, Road Condition* are contained. The figure below shows the snapshot of a sample feed from Wyoming.

```
<section n="0" roadCode="SHERI90NMONTI" route="ML90I" commonName="I 90 / US 87" fromMp="0.0"
toMp="10.1" longitude="-107.2289539" latitude="44.95167745" description="I 90 / US 87 between
the Montana State Line and Ranchester - <b>EASTBOUND</b>" rptTime="2014-11-02 23:39:43"
eightCodes="81" nineCodes="91" eightOne="1" eightTwo="0" eightThree="0" eightFour="0"
eightFive="0" eightSix="0" nineOne="1" nineTwo="0" nineThree="0" nineFour="0" nineFive="0"
nineSix="0" nineSeven="0" bi="0" cl1="0" cl2="0" fr="0" nlt="0" ntt="0" nut="0" c2lhpv="0"/>
```

Figure 5 Sample feed from Wyoming

The snapshot above shows an event of *Road Condition.* It should be noted that Wyoming does not have a plain text to describe the road condition. Instead, it uses a code system to identify different conditions. It is represented by the different integer values in the fields of "eightCodes" and "nineCodes" in the feed. The figure below explains the meanings of each value in the code system.

Code	Meaning
81	Surface: Dry
82	Surface: Wet

83	Surface: Slick
84	Surface: Slick in spots
85	Surface: Drifting snow
86	Surface: Closed
91	Atmospheric: Clear
92	Atmospheric: Snowfall
93	Atmospheric: Rain
94	Atmospheric: Strong Winds
95	Atmospheric: Fog
96	Atmospheric: Blowing snow
97	Atmospheric: Limited visibility
B1	Black Ice
CL1	Chain law tier 1
CL2	Chain law tier 2
FR	Falling rock
NLT	No light trailers
NTT	No trailer traffic
NUT	No unnecessary travel
C2LHPV	Closed to light, high-profile vehicles

Figure 6 Wyoming code system

South Dakota

South Dakota has its custom event XML feeds on its server. Layer of *Incident, Road Work, Road Closure, Road Condition, Truck Restriction* are contained. The format of the feed roughly follows the TMDD.

Weather forecast

Weather forecast information is obtained from NOAA Forecast Database [1]. The data request/response is made possible by the National Digital Forecast Database (NDFD) XML Simple Object Access Protocol (SOAP) server. The NDFD XML is a service providing the public, government agencies, and commercial enterprises with data from the National Weather Service's (NWS) digital forecast database. This service provides NWS customers and partners the ability to request NDFD data over the internet and receive the information back in an XML format. Once a SOAP request from client end is sent to the NOAA data center, a customized NDFD XML will be created on the server. And then the data feed in XML format will be wrapped as a SOAP response being sent back to the client end, which is our injector. A sample weather forecast data feed is showing below.

The NDFD XML contains forecasts for combinations of a lot of meteorological parameters, such as temperature, probability of precipitation, weather, weather condition, etc.

Static Layers

Most static layers come in the format of CSV. Usually all the needed information from member states of a specific layer is summarized into a single spreadsheet, though

some extra unneeded information is also included which needs to be cleared. The figure below shows a chunk of *Cautionary Zones* feed.

Type	State	Interstate	MP start	MP end	lat/long end A (estimate)	lat/long end B (estimate)
Cautionary Zone	WA	90	90	95	-120.82068440000002,47.1525444	-120.7618904,47.0997526
Cautionary Zone	WA	90	120	125	-120.3115368,46.9574684	-120.20974160000002,46.9436411
Cautionary Zone	WA	90	255	260	-117.8238201,47.4090963	-117.7531815,47.4618707
Cautionary Zone	ID	90	60	65	-115.9495354,47.488789	-115.85383419999998,47.4691818
Cautionary Zone	MT	90	25	30	-115.2477837,47.3221851	-115.15439990000002,47.2929694
Cautionary Zone	MT	90	90	100	-114.2139959,47.0087918	-114.0534496,46.918760400000004

Figure 7 A chunk of Cautionary Zone feed

The layers described above do not cover all required layers. Due to the similarity of other layers, we will not introduce those more in details.

CHALLENGES

In this section we will introduce some of the challenges while developing the system, in the aspects of data format, data accuracy, semantic analysis and etc.

Format Unification

As introduced in the previous sections, member states host their own traffic event reporting system and utilize different standards and format to store their data. The challenge is that we have to make choice of needed data for each state and for each layer. And also we need to fix the hole if the content is missing in some certain cases.

We compare a few example schemas here. For example, for the location information in the layer of *Road Condition* we compare the data format of states Wisconsin, Montana and Minnesota.

The description below is the location information schema in *Road Condition* layer from Wisconsin.

- Link-ownership: always set to WisDOT
- Link-designator: set to the concurrent highway name list for the given segment, e.g., I-90 / I-94.
- Primary-location: describes the starting point for the WRS segment
    - Geo-location: latitude and longitude for the start of a segment
    - Point-name: cross-street description for the start of the segment, in text
    - Upward-area-reference: city and country information for the start of the segment

        - Area-name: city, in text
        - Upward-area-reference/area-id: FIPS country code
        - Upward-area-reference/area-name: country name
    - Secondary-location: describes the end-point for the WRS segment, same structure of primary-locaiton
    - Link-direction: the direction of the WRS segment, currently set to "both directions"

Compared with the well-structured format of location information of Wisconsin, Montana offers a much simpler and less content.

- SIGNED_ROUTE: set to the route name, e.g., I-90, I-94
- BEG_MP: the begin mile post of a road condition section, e.g., 323.3
- END_MP: the end mile post of a road condition section, e.g., 330
- SEGMENT_DESC: the description of a road condition section in text, e.g., Bozeman pass to exit 330 – Livingston

Minnesota provides a similar schema for the location information with the one from Wisconsin, but in a different version. The figure below is the visualization of location schema of Minnesota opened in Microsoft Visual Studio 2010.

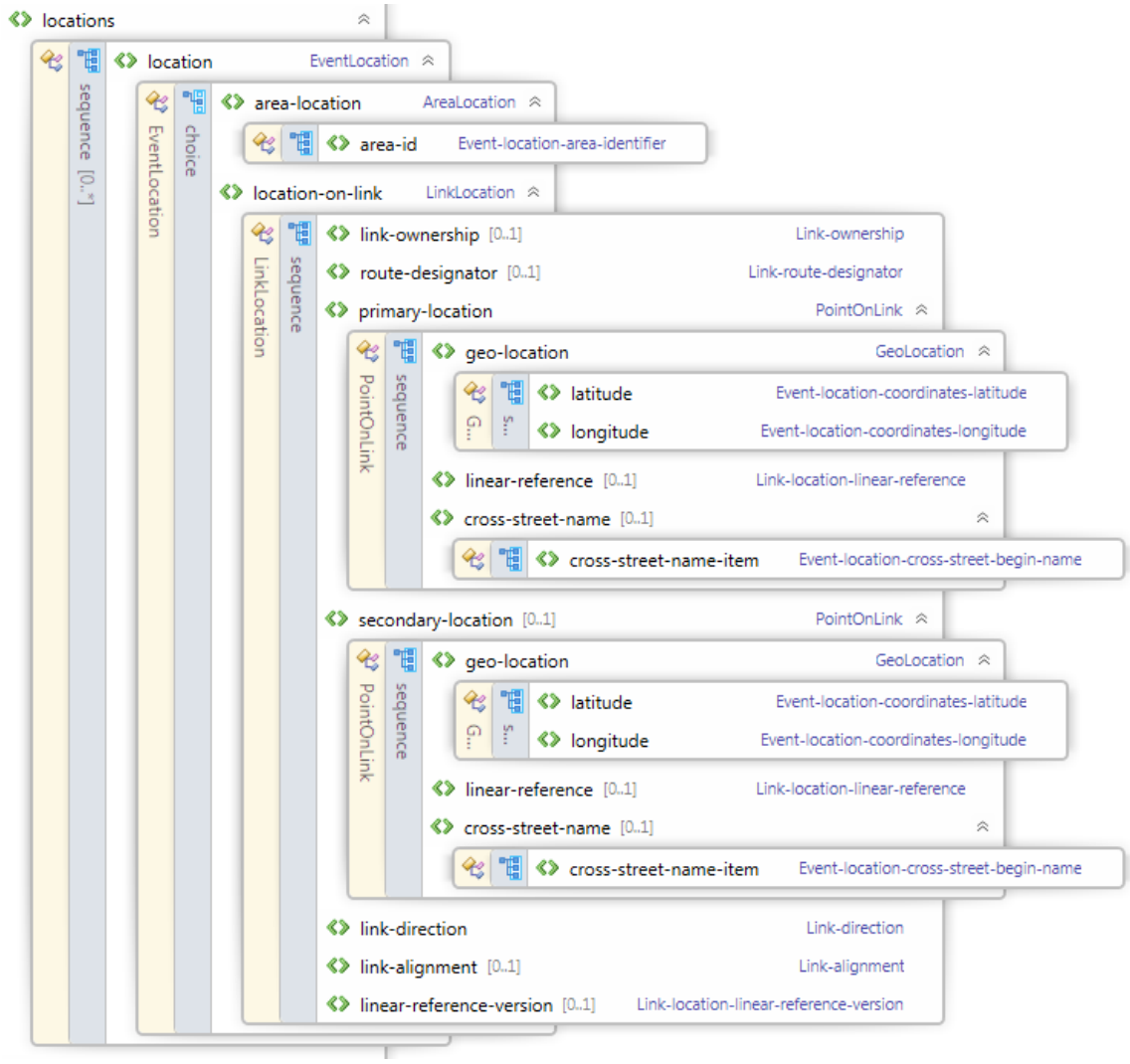


Figure 8 Location schema visualization of Minnesota

It could be seen from the examples above, data feeds come in different format and complexity. It would not be practical to design a standard for each single layer and for each single member state on the injector side. Instead, some format unification work needs to be done to serve as a data parsing standard.

Layer Separation

The idea of separated layers such as *Incident, Road Work, Road Closure and Road Condition* is not natural. Instead it is brought up by the analysis of customer needs. Research and survey have been done to figure out how to categorize all those traffic related data. But obviously not every state separates its data feed into such categories. For example, Washington only provides three separated traffic-related feeds, *Highway Alerts, Traffic Flow* and *Commercial Vehicle Restrictions*. And state Idaho offers a single data source, and defines a variety of event types which are actually more than the number of needed layers. There might be layers that are not necessary for our system, and at the same time we might need to integrate a couple layers into a single layer. It is our task to make all data feeds well separated and fall into the predefined categories. The figure below shows the event types that may appear in Idaho's single data feed.

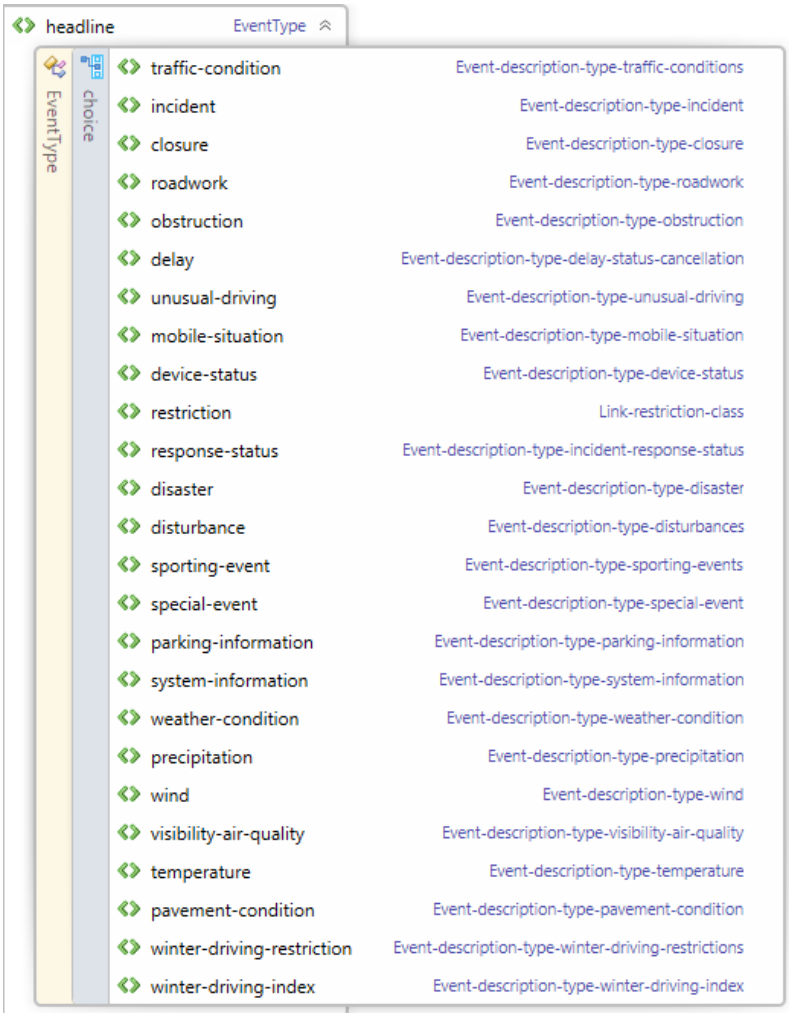


Figure 9 Event types of Idaho feed

Semantic Analysis

Semantic analysis is a necessary logical phase of the parsing process. The parsing process in our injectors is no exception. Semantic analysis is needed, in order to extract key information, to classify different event types and so on.

For example, in some cases the state feed does not only report active events, but also reports inactive events. According to the system requirement, it is our responsibility

to mark active and inactive events based on certain key words or phrases, and handle them in different ways. Also, pattern matching is to be used extensively to extract useful information and to categorize different events. Thus the semantic analysis is needed to match key words or even do more complicated tasks, such as relating syntactic structures in the data feed context.

Data Cleansing

Data cleansing shows as an early stage in the back end coding of the system. It works to remove inconsistent/inaccurate data or make those inconsistent/inaccurate ones consistent/accurate.

It is common that the data feeds contain inconsistent records. Many of the events are reported from the field and may be processed by different individuals using different formats. All of those could cause user entry errors and data inconsistence. A typical example would be the various spelling of event type “Mountain Pass”. While pushing Mountain Pass records into database, we saw many different spellings, such as “Mountain Passes”, “Mountains Passes”, “Mountian Pass”, “Mountain Pas”. In order to always produce valid and accurate data stream, it is important to make sure the data cleansing process take case of as many cases as possible in the early stage of data parsing.

Geometry Data Issues

This is a big and difficult part throughout the whole project. Geometry data shows up in every stage of the system, such as data feed, format translating, data injecting,

database storage and front end displaying. We would discuss our approach of working on geometry type data in detail in the following chapters. But at first the challenges and difficulties would be introduced briefly in this section.

The first challenge would be how to parse the geometry data in the data feed. The various location data could either be a point represented by latitude and longitude pairs, or milepost information on a certain route, or road segments represented by line strings or even merely a location description in text. It would not be an easy task to interoperate geometry data in all those types. Geometry data parsing strategy is to be developed for every possible geometry type. Once the geometry data is identified and matched, an optimal solution will be applied if the feed provides more than one type of geometry data.

The second challenge would be how to store the geometry data in the database. Specially, a suitable database manage system would be chose before anything else. Different host database manage systems have different geometry storage strategies and data types. One primary consideration about the target database manage system is that the geometry data queries and storage should be efficient. Most database manage systems have more than one geometry type, thus research is to be conducted before finalizing the solution type.

The third challenge is how to translate between different geometry types. This geometry translation should be done in the phrase of data injection. For example, the milepost information is the only geometry information provided in the feed, and at the same time the point is the only acceptable location type for that certain layer. In this case, the injector has to do a translation between the milepost and the point represented by

latitude and longitude pairs. Depending on the geometry type of the data source, different translation methods need to be incorporated while parsing the feed. And also format translation needs to be conducted between data feed and the geometry format used in the database.

WEB SERVICES

Extensible Markup Language

Extensible Markup Language (XML) plays as important role in the system. Most of the data feeds show up as a XML file. XML was designed to describe data, it carries data independent of software and hardware. This markup language is widely used for the representation of arbitrary data structures in web services. In this section, we describe the key terminologies of XML, schema and the Perl module for parsing XML documents.

XML terminologies

The following is a list of XML terms and definitions. It is not an exhaustive list of all the constructs but a list of often used ones [2].

Attribute: A name and its value which are included inside an XML tag. For example, in the tag <book isbn=”0-32-4245-X”>, isbn is the name and its value is 0-32-4245-X, values are enclosed in single or double quotes.

Cascading Style Sheet (CSS): A style sheet that defines the appearance of an XML or HTML document directly on the client.

Child: An XML element located inside another XML element.

Parent: An XML element which contains another XML element. The XML element contained by the parent is the child element.

Root Element: The element that contains all of the other elements in an XML document.

Document Type Definition (DTD): A collection of markup declarations contained in a single or multiple XML files that describe an XML document's permissible elements and structure. A DTD ensures that a uniform structure will be used across all documents.

Encoding attribute: An attribute inside the XML declaration that indicates the level of encoding in the document. For example, <?xml version="1.0" encoding="UTF-8"> indicates that a compressed form of Unicode will be used that assigns one byte for ascii characters, two bytes for other common characters, and three bytes for all other characters.

Element: A logical document component which either begins with a start-tag and ends with a matching end-tag or consists only of an empty-element tag. The characters between the start- and end-tags, if any, are the element's content, and may contain markup, including other elements, which are called child elements. An example of an element is <Greeting>Hello, world. </Greeting> (see hello world). Another is <line-break />.

Valid XML: XML that meets the constraints defined by its Document Type Declaration.

XML schema

An XML schema describes the structure of an XML document just like a DTD, but it is more powerful than DTD. The advantages of an XML schema over the DTD are listed below [8].

- XML Schemas are written in XML

- XML Schemas are extensible to additions
- XML Schemas support data types
- XML Schemas support namespaces

Being able to read and understand the XML schema is such an important skill for this project. Since we will not only read extensive schemas of the XML feeds, but will also design the schema to incorporate different feeds for each layer. In this section, we will take a chunk of XML schema as an example to explain the structure and meanings of each segment.

```
<xs:element name=”note”>
 <xs:complexType>
   <xs:sequence>
      <xs:element name=”to” type=”xs:string”/>
      <xs:element name=”from” type=”xs:string”/>
      <xs:element name=”heading” type=”xs:string”/>
      <xs:element name=”body” type=”xs:string”/>
   <xs:sequence>
 <xs:complexType>
<xs:element>
```

The above schema is interpreted like this:

- <xs:element name="note"> defines the element called "note"
- <xs:complexType> the "note" element is a complex type
- <xs:sequence> the complex type is a sequence of elements
- <xs:element name="to" type="xs:string"> the element "to" is of type string (text)
- <xs:element name="from" type="xs:string"> the element "from" is of type string

- `<xs:element name="heading" type="xs:string">` the element "heading" is of type string
- `<xs:element name="body" type="xs:string">` the element "body" is of type string

While working with XSD standard schema, any text-based editor can be used to edit an XML schema but a graphical editor would offer big advantages. The tree structure of an XML schema would be displayed which makes an XSD document easier to read and edit. In our project we utilized a couple XML schema editors which include Altova XML Spy, Liquid XML Studio and Microsoft Visio Studio.

Perl and XML::Parser

Perl is the language we choose to write the data injectors. Perl is a high level programming language that really excels at text processing, Common Gateway Interface (CGI) programming and interfacing with databases. The powerful regular expression engine of Perl makes it a perfect choice for our backend data parsing. The build-in features of Perl makes pattern matching in text an easy work, which is widely used in the injectors. (//we can explain more about why choose Perl.)

Most of the data feeds are represented in XML file. In XML data and markup are mixed together, so the parser has to sift through a character stream and tell the two different things apart. Some characters or symbols separate markups from data, for example primarily angle brackets “<” and “>” are for elements, comments, and processing instructions. Ampersand “&” and semicolon “;” are for entity references. The parser should know when to expect a certain instruction, for example an element that contains data must bracket the data in both a start and end tag. With such designs the

XML parser can quickly chop a character stream into discrete portions as encoded by the XML markup.

We chose the XML::Parser as the Perl module to do the data parsing job. It is built on top of XML::Parser::Expat, which is a lower level interface to James Clark's expat library. Each call to one of the parsing methods creates a new instance of XML::Parser::Expat which is then used to parse the document. Expat options may be provided when the XML::Parser object is created. These options are then passed on to the Expat object on each parse call. They can also be given as extra arguments to the parse methods, in which case they override options given at XML::Parser creation time.

The behavior of the parser is controlled either by "Style" and/or "Handlers" options, or by "setHandlers" method. These all provide mechanisms for XML::Parser to set the handlers needed by XML::Parser::Expat. If neither Style nor Handlers are specified, then parsing just checks the document for being well-formed. When underlying handlers get called, they receive as their first parameter the Expat object, not the Parser object [4].

SOAP Web Services

Some of the data feeds are carried out through Simple Object Access Protocol (SOAP). We will introduce the SOAP web service in this section. SOAP is a communication protocol specification for exchanging structured information in the implementation of web services in computer networks [7].It is a common way for a program running in one kind of operating system to communicate with a program in the

same or another kind of an operating system. A good feature of SOAP is that it is platform independent and language independent.

Another big advantage of SOAP is that it allows you to get through firewalls. Many applications today communicate using Remote Procedure Calls (RPC) between objects like Distributed Component Object Model (DCOM) and Common Object Request Broker Architecture (CORBA), but HTTP was not designed for this. RPC represents a compatibility and security problem. Firewalls and proxy servers will normally block this kind of traffic. Instead, HTTP is supported by all Internet browsers and servers, HTTP requests are usually allowed through firewall. SOAP is based on XML and communicates via HTTP, thus it becomes an advantage of SOAP being more likely to get around firewalls, in addition to its features of OS and language independent.

As we said, a SOAP message follows the format of an XML document. It contains the following elements:

- An Envelope element that identifies the XML document as a SOAP message
- A Header element that contains header information
- A Body element that contains call and response information
- A Fault element containing errors and status information

Figure 10 shows an example of SOAP response message. We are using this example to explain some of the important syntax rules.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Figure 10 A sample SOAP response

As the syntax rules, a SOAP message must use the SOAP Envelope namespace and SOAP Encoding namespace. And it must be encoded using XML. At the same time, a SOAP message must not contain a DTD reference and must not contain XML Processing Instructions.

REST Web Services

Representational **S**tate **T**ransfer (REST) is another web service we use in the process of data parsing. REST is an architecture style for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines. RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations. REST is a lightweight web service compared with SOAP, WSDL, et al. Also, a REST service is both platform independent and language independent. It runs on top of HTTP and can easily be used in the presence of firewalls.

REST describes a set of architectural principles by which data can be transmitted over a standardized interface (such as HTTP). REST does not contain an additional messaging layer and focuses on design rules for creating stateless services. A client can access the resource using the unique URI and a representation of the resource is returned. With each new resource representation, the client is said to transfer state. While accessing RESTful resources with HTTP protocol, the URL of the resource serves as the resource identifier and GET, PUT, DELETE, POST and HEAD are the standard HTTP operations to be performed on that resource.

It needs to be noticed that REST is an architectural style of networked system. It's not a standard itself, but does use standards such as HTTP, URL, XML, etc.

Web Services Description Language

Web Services Description Language (WSDL) is the standard format for describing a web service in XML format. It is an XML based protocol for information exchange in decentralized and distributed environments. WSDL is often used in combination with SOAP and XML schema to provide web service over the internet. A client program connecting to a web service can read the WSDL to determine what functions are available on the server. Any special data types used are embedded in the WSDL file in the form of XML Schema. The client can then use SOAP to actually call one of the functions listed in the WSDL.

CHALLENGE SOLUTIONS

Standard Schema

Since member states report and categorize their feeds differently, it is hard to make choice of needed data for each layer. While it is good to know that according to the system requirement certain information is required for each layer, there is more work needed to be done in addition to a simple list of required columns. The responsibility of data injector is to work seamlessly between data parsing and the database. For almost every layer, we need to store more information than those required items. Some of those added items would facilitate backend debugging, some would facilitate API calls and front end data representation. To work seamlessly with the database, the format of each item in the layer needs to be consistent with the data type of the corresponding table column in the database.

For each layer we build an XML schema that defines the required items, data format and the hierarchical structure of the items. It needs to be noticed that not all items are necessary, instead some are optional items. It is totally understandable since not all the items have the same priority and need to be displayed to the users. Figure 11 is a screenshot of a sample schema of the layer *Camera*.

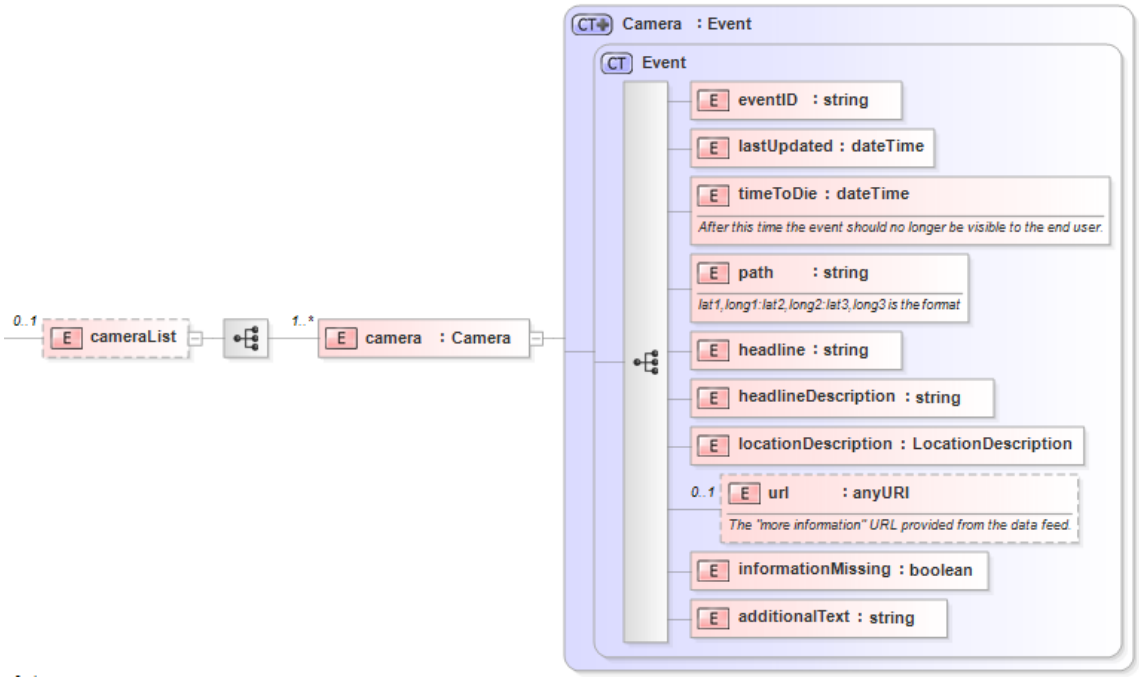


Figure 11 Sample schema of *Camera* layer

It could be seen that some items such as *additionalText*, *informationMissing* are not the required information from the system requirement, but they will be useful to incorporate those supplemental data in the state feeds. We build a schema for each single required layer. Those schemas work as the standard module and guideline for the data injectors. For each layer, by comparing the standard schema and the data feed we can screen out those non-needed and redundant information and only parse those data items that shown on the standard schema. And also, those missing data items from the data feed would be easily found.

Data Mapping

Without doubt, it is a good idea to have clickable layer on the OTIIS website. These layers cover the most users' needs of planning a travel. However, the categorization of these layers is not natural, and most of our member states don't have the same definition of these layers. And since we don't (we also should not) have any direct control on the event reporting system of each state's DOT, it is unrealistic to ask our member states to change their data feeds and make the different events tailored to the proposed layers.

We read the definitions of the proposed layers, conducted research on the data feed schema, studied the data feeds in a relatively long time span and discussed with DOT professionals. At last we mapped the data items from the data feed to the required layers. The spreadsheet below shows the data mapping of traffic-related layers.

	Road Work	Temp Truck Restriction	Crash/Incident	Road Closures	Road Conditions	Traffic Congestion
	XML Tags in Feed	XML Tags in Feed	XML Tags in Feed	XML Tags in Feed	XML Tags in Feed	XML Tags in Feed
MN	feu-g: roadwork	feu-g: restriction	feu-g: incident, anything else not otherwise listed	feu-g: closure	feu-g: traffic-condition, weather-condition, visibility-air-quality, pavement-condition, winter-driving-restriction	feu-g: delay
ID	feu-g: roadwork	feu-g: restriction	feu-g: incident, anything else not otherwise listed	feu-g: closure	feu-g: traffic-condition, weather-condition, visibility-air-quality, pavement-condition, winter-driving-restriction	feu-g: delay
WI	Wis511LCS: roadwork		Wis511: accidentsAndIncidents	Wis511LCS: closure	WRS-conditions: winterDrivingRestrictions	traffic speed lower than 30 mph
ND	workzones: roadwork	loadresrtrict-current: restriction-class	alerts: all phrases	severe.xml: closure	roads: winter-driving-index	
MT	roadwork		incident		conditions	
WA	HighwayAlerts: Construction, Maintenance, P-1 Sand, Plowing, Deicing, Sand, Utilities	CVRestrictionData: Chain Enforcement, Flammable Restriction, Travel Restriction	HighwayAlerts: anything not listed for other layers	HighwayAlerts: Bridge Closed, Closure, Lane Closure, Debris blocking, Emergency closure, HCB Closed Main, HCB Closed Marine, HCB Closed Police, HCB Closed	HighwayAlerts: Avalanche Control, Bridge, Bridge Lift, Debris, HCB Open, Hood Canal Bridge, In Service, ITS & IT, Keller Ferry, Out of Service	TrafficFlow: Boat Traffic, Heavy Traffic
WY	construction: roadwork			roadconditions: CODE 86	roadconditions: CODE 81-85	
SD	samplefeed: construction	samplefeed: restrictions, vehicle_restrictions	samplefeed: event	samplefeed: closure	samplefeed: conditions	

Figure 12 Data mapping table

In figure 12, we made a one-on-one mapping in the data feed and in the required layers. In most cases, each member state has more than one feed. We specify the name of each feed and list the XML tags in the feed. The light blue cells indicate that certain layer is available, otherwise the cell color is set as red. From the table above we can see that not all layers from member states are available.

Semantic Analysis

Generally speaking the event reporting system that adopts TMDD standard offers a more accurate and is easier to parse data feeds. This is because the TMDD has a well-defined structure and categorizes different events into pre-defined types. This effort

effectively reduces the number of open text fields. For example, a well formatted data feed in TMDD standard may describe a restriction event as below.

```
<descriptions>
  <description>
     <phrase>
        <restriction>height limit</restriction>
     </phrase>
  </description>
  <description>
     <quantity>
        <link-restrictions>
            <restriction-height>447</restriction-height>
        </link-restrictions>
     </quantity>
  </description>
</descriptions>
```

In the above code example, useful data is wrapped around by markup tags and elements, which could be easily filtered out. And the data stream follows a good format and is well structured. In contrast, the same event described in open text field may look like this: Road restriction with height limit of 447. In this case, semantic analysis is going to be needed to extract the event type *restriction*, headline as *height limit* with the *quantity* of 447. Semantic analysis is hard to implement and its accuracy would be an issue. Fortunately, this is only a few open fields we need to deal with in the data feeds. One example is the nested description in static layers. We noticed that the supplemental URL shows up in the description filed of the feed, but we do have another separate field that stores the URL link. So in this case the semantic analysis is needed to identify and extract the URL link.

The strategy would be firstly identify the part of the URL that precedes the "://" characters, which defines the protocol used for the URL. Second, find the part of the URL located between "://" and the first "/". This part describes the server on which the website is located, and is composed of several separate sections, separated by periods, in the form "subdomain.domain.extension". Third, identify the part of the URL located after the first "/". This part usually locates the target file you are accessing.

The program will be complicated and likely be slowed down if there are a lot of semantic parsing cases. Fortunately, in most cases what the injector has to do is pattern matching instead of complex semantic analysis. Due to the difficulty of parsing an open field text, member states usually input text in certain patterns. For example, an inactive road condition event is usually described in certain phrases, like "DRY/MOSTLY DRY" or "no information". By matching these limited phrases, we know a *Road Condition* event is currently active or inactive. For another example, state Wyoming uses different code number to describe different event types. CODE 86 indicates a road closure event, CODE 82 indicates that the road surface is wet. So in this case what we need to do is simply detect and match different code values and interpret the corresponding meaning behind the code.

Time Format and Data Cleansing

Time stamp is an important component in the data feeds. It is required to record the event's start, end and update time for most live events. Our member states along the North/West Passage spans four time zones in the United States. It becomes necessary to

record the time offset for each event. When the injector is parsing a data feed, it will read the time offset if it is provided. Or record the time offset based on the location, if it is not provided. Also, the time stamps may be represented in different format. Some may be as straightforward as “2014-10-09 21:01:46” or “20130806 142737”, while some others may appear as more complicated Unix time. The formal time stamp could be easily parsed by counting the number of digits to identify the right part of the data and time. Unix time is defined as the number of seconds that have elapsed since 00:00:00 Coordinated Universal Time (UTC), Thursday, 1 January 1970. And it won’t be a problem to be translated into the time format we choose for the database.

In addition to those frequently updated layers such as *Road Work, Truck Restriction*, there are static layers which don’t need to be updated. Static layers include *Truck Stop, National Park, Rest Area, Cautionary Zone*, etc. Most of these data feeds are in CSV format and need to be cleaned before pushing into database. Depending on the volume of the feed, the data cleansing job could either be conducted manually or automatically. If a data feed file contains a lot of entries and hard to be checked by hand, we can write a script to solve the problem. Thanks to the build-in feature of regular expression, language Perl is powerful of doing “search and replace” and pattern matching.

RESOLVING GEOMETRY DATA ISSUES

The OTIIS website is built on top of the Google Map. The geometry type data exists throughout all the stages during the system developing procedure. Specifically for the data injectors, we need to parse all the geometry type data in all the feeds, design a database which is geometry type friendly and translate between different geometry types. In this chapter, we will discuss all the strategies, spatial database design, data structures, geometry type data representations that involved in the following sections: geometry data in the feed, geometry data in database and geometry data type translations.

Geometry Data in the Feed

Different feeds utilize different geometry data types to represent their location information. Most of our member states choose latitude longitude pairs to describe the location of the event. One point location is indicated by a single pair of latitude longitude information, which is shown below.

```
<event-location>
  <primary-location>
      <geo-location>
        <latitude>46914898</latitude>
        <longitude>-97991255</longitude>
      </geo-location>
  </primary-location>
</event-location>
```

It could be seen from the XML chunk above, one point location could be identified by a single pair of coordinates. The secondary location tag is absent. We can use the build-in method of the chosen Perl module to extract the latitude and longitude

pairs. The code sample below extracts the coordinates and make them ready to use for further functions.

```
sub hdl_char{
      my($p, $str)=@_;
      if($currentTag eq "latitude" && $prime_loc_flag){
            $prime_latitude=$str/1000000;
      }elsif($currentTag eq "longitude" && $prime_loc_flag){
            $prime_longitude=$str/1000000;
           $prime_point="'$prime_latitude,$prime_longitude'";
      }
      ##more lines of code##
}
```

If the event is identified by two points or a road segment, then a single pair of coordinates is not enough. In this case, two points are provided. For the injectors, we set a secondary point flag to mark and extract the second point information. And in the section of geometry data type translation, we will discuss how to translate two points into a consecutive road segment, which is represented by a line string of coordinate pairs. In addition to the popular latitude longitude pairs, milepost is another form of geometry data. Milepost is usually used to indicate an event occurs on the road instead of off road. Milepost information always comes together with the route designator in order to indicate a location. And sometimes a location description in text would show up in the data feed which makes it more human readable.

Geometry Data in Database

PostgreSQL and PostGIS

PostgreSQL is chosen as the Database Management System (DBMS) of the project. Together with this spatial DBMS, PostGIS is also installed. PostGIS is an open source software program that adds support for geographic objects to the PostgreSQL object-relational database. With the support of PostGIS, geographic objects allow location queries to be run in Structured Query Language (SQL). A location query in the geodatabase could be as simple as the followings code shows.

```
SELECT superhero.name
FROM city, superhero
WHERE ST_Contains(city.geom, superhero.geom)
AND city.name = 'Gotham';
```

In addition to the basic location awareness, PostGIS offers many features that other spatial databases such as Oracle Locator/Spatial and SQL Server do not have. PostGIS adds extra data types, such as geometry, geography, raster to the PostgreSQL database. Functions, operators and index enhancements are also added to these spatial types. All those unique features augment the power of the core PostgreSQL DBMS, making it a fast, feature-plenty and robust spatial DBMS. And this is the exact reason why PostgreSQL is chosen to manage our spatial databases.

ST_Geometry

ST_Geometry is the geometry storage data type in our database. It extends the capabilities of the PostgreSQL database by providing storage for objects (points, lines, and polygons) that represent geographic features. The ST_Geometry data type

implements the SQL 3 specification of user-defined data types (UDTs), allowing you to create columns capable of storing spatial data such as the location of a landmark, a street, or a parcel of land. It provides International Organization for Standards (ISO) and Open Geospatial Consortium, Inc. (OGC) compliant structured query language (SQL) access to the geodatabase and database. It was designed to make efficient use of database resources; to be compatible with database features such as replication and partitioning; and to provide rapid access to spatial data [3].

ST_Geometry itself is an abstract, noninstantiated superclass. However, its subclasses can be instantiated. An instantiated data type is one that can be defined as a table column and have values of its type inserted into it. Although you can define a column as type ST_Geometry, you do not insert ST_Geometry values into the column since it cannot be instantiated. Instead, you insert the subclass values. The following figure shows the structure of data type ST_Geometry and its subclasses.

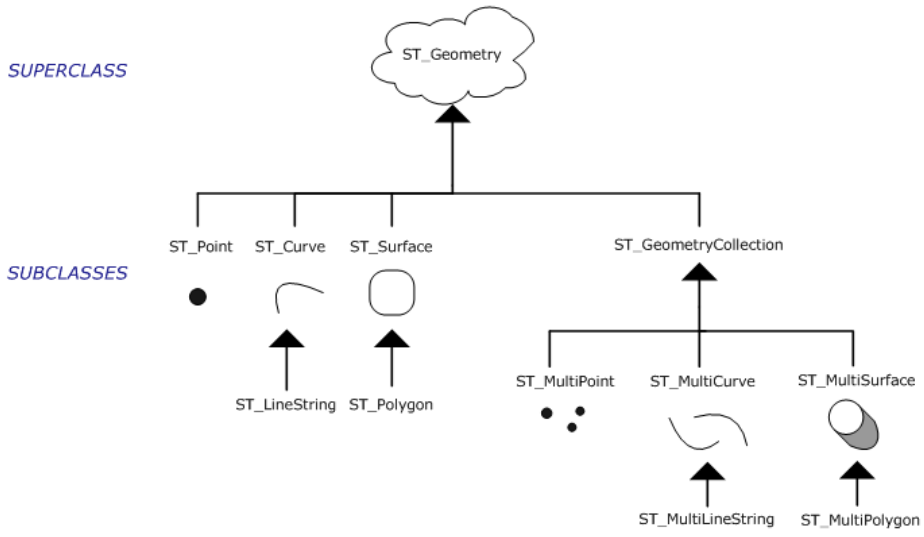


Figure 13 The structure of ST_Geometry and its subclasses

Geometry Data Type Translations

Translate point to geometry type data

Most location information in data feeds are points represented by latitude and longitude pairs. Depends on the event type, location information will be stored in different manners in the Database. In this section, we will discuss the layers that store the point as the location information. Such layers include *Crash/Incident, Traffic Congestion, Weight Station, Truck Stop*, etc. According to the system requirement, only one point is needed to describe the location information of these layers. A secondary point would be discarded if provided in the feed.

Before describing the geometry type translation functions, we introduce two important conceptions that are used in the functions. A **Spatial Reference System Identifier (SRID)** is a unique value used to unambiguously identify projected, unprojected, and local spatial coordinate system definitions. These coordinate systems form the heart of all GIS applications [6]. In our geodatases, SRID is used to uniquely identify the coordinate systems used in the data feeds. SRID values associated with spatial data can be used to constrain spatial operations. For instance, spatial operations cannot be performed between spatial objects with differing SRIDs in some systems, or trigger coordinate system transformations between spatial objects in others. SRIDs are typically associated with a well-known text (WKT) string definition of the coordinate system.

A Well-known text (WKT) is a text markup language for representing vector geometry objects on a map, spatial reference systems of spatial objects and transformations between spatial reference systems. A WKT string for a spatial reference system describes the datum, geoid, coordinate system, and map projection of the spatial objects [9].

PostGIS contains functions that can convert geometries to and from a WKT representation. *ST_GeomFromText* is the function used most frequently to return a specified ST_Geometry value from WKT representation. Its synopsis looks like below.

```
Geometry ST_GeomFromText(text WKT, integer srid);
```

A typical PostGIS function with parameters we use to translate a point to the data in geometry type in the database would be like this:

```
Geometry ST_GeomFromText('Point(-71.060316 48.432044)', 4326);
```

Of which, -71.060316 is the longitude and 48.432044 is the latitude of the event location and number 4326 is the associated SRID value.

Translate segment to geometry type data

One point is not enough when describing the location and displaying on map, for the layers like *Road Closure, Road Condition, and Scenic Route*. Instead a road segment which is a series of points between the start point and end point is needed to be paint on map, and needs to be stored in database in the data type of ST_LineString, which is one of the subclasses of the abstract data type ST_Geometry. In this part we introduce the approach of generating the road segment based on the start and end point, and the method of storing the road segment in geometry type in the geodatabase.

Translate two points to a road segment by using additional road information databases. The road information database stores road segments of each member state. By comparing the distance between the source point and the geometry point on route, the query *twoPointsToLinestring()* returns the route number and the segments between the two points.

Once we got the road segment returned by the function *twoPointsToLinestring()*, we use the following function to get the ST_Geometry value.

```
Geometry ST_GeomFromText($LineString, 4326);
```

The above function is no difference with the one we used for translating a point into a PostGIS geometry. Also the SRID value is unchanged. `$LineString` is the variable that represents a linestring geometry of the WKT version.

Translate milepost to geometry type data

Not every layer has the location information represented by the coordinate system. This is understandable since it may not be necessary or hard to record the accurate coordinates of the events, sometimes a rough location descriptor is acceptable to indicate the location. In this situation the Linear Reference System (LRS) is used to store the location information. A linear reference system stores data using a relative position along existing line features, such as a high way, rivers, railroad and so on. Location is given in terms of a known linear feature and a position along it. For example, Interstate 94, milepost 34.3 uniquely identifies a position.

While there isn't a problem to use linear reference system to identify a position for the event reporting, it is hard to integrate it into our system. Without coordinates we

cannot display a location on map. To solve this problem, the first step would be finding a solution of mapping mileposts along the road to coordinates. Once we got the coordinates information of the event, we can either translate it into a one point geometry data or a line string geometry data, based on the event type and the provided location data. Here we take state Montana as an example to explain the translation process.

We access the data center of the Montana's Department of Transportation with limited permission to download the database of reference markers. The database usually contains reference markers along a lot of on-system routes (such as Interstate, non-Interstate National Highway System and Primary highways), GPS data and many other descriptive items. The figure below shows the reference markers covered by the Montana DOT.

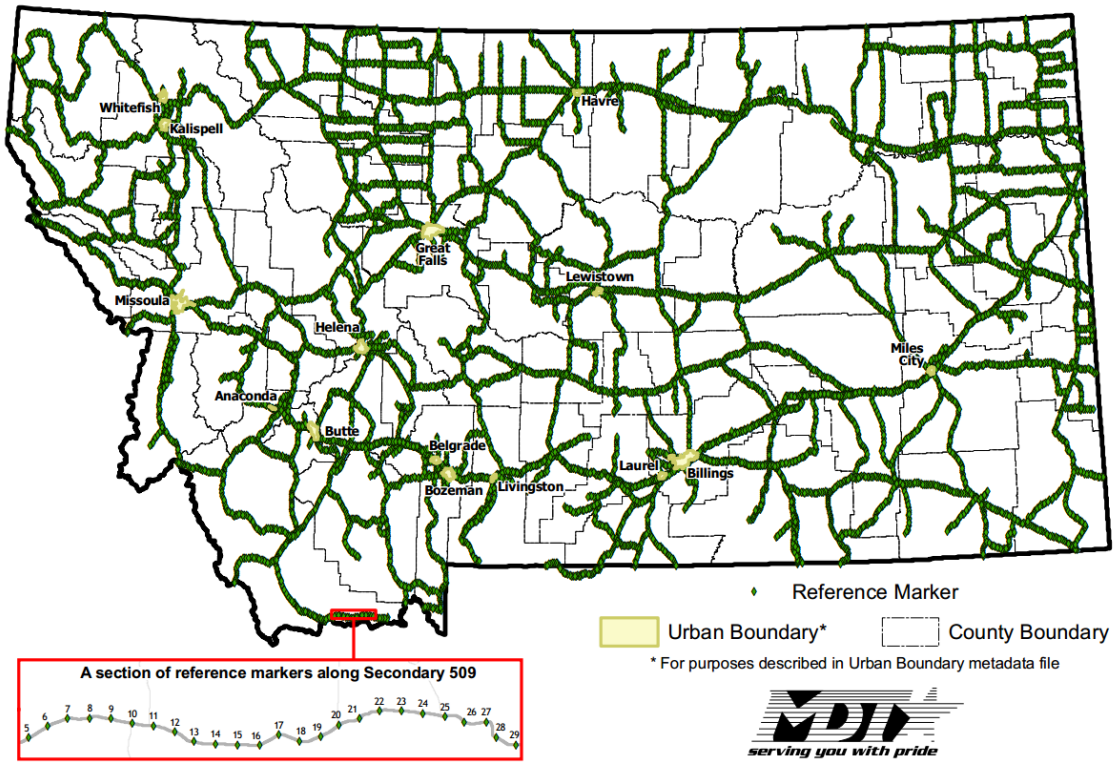


Figure 14 Mileposts covered by Montana DOT

The downloaded files usually need to be opened in ArcGIS software which is a popular geographic information system for working with maps and geographic information. We filter out the non-needed information, and only generate coordinates at every reference markers along the chosen routes.

Once we generated the mapping between the mileposts and coordinates, we build a database table for each interested route that contains the milepost and the corresponding coordinates. Thus each time when we encounter an event without latitude and longitude as location but with the milepost information, we can query the database based on the provided route designator and milepost to fetch the coordinates. And then use the same routine to generate point or linestring of PostGIS geometry type.

DATA PARSING PROCEDURE

In this chapter, we describe the process of data fetching, parsing and database connection in detail. This procedure works for most traffic-related dynamic layers. The flow chart below shows the operation flow in such a procedure.

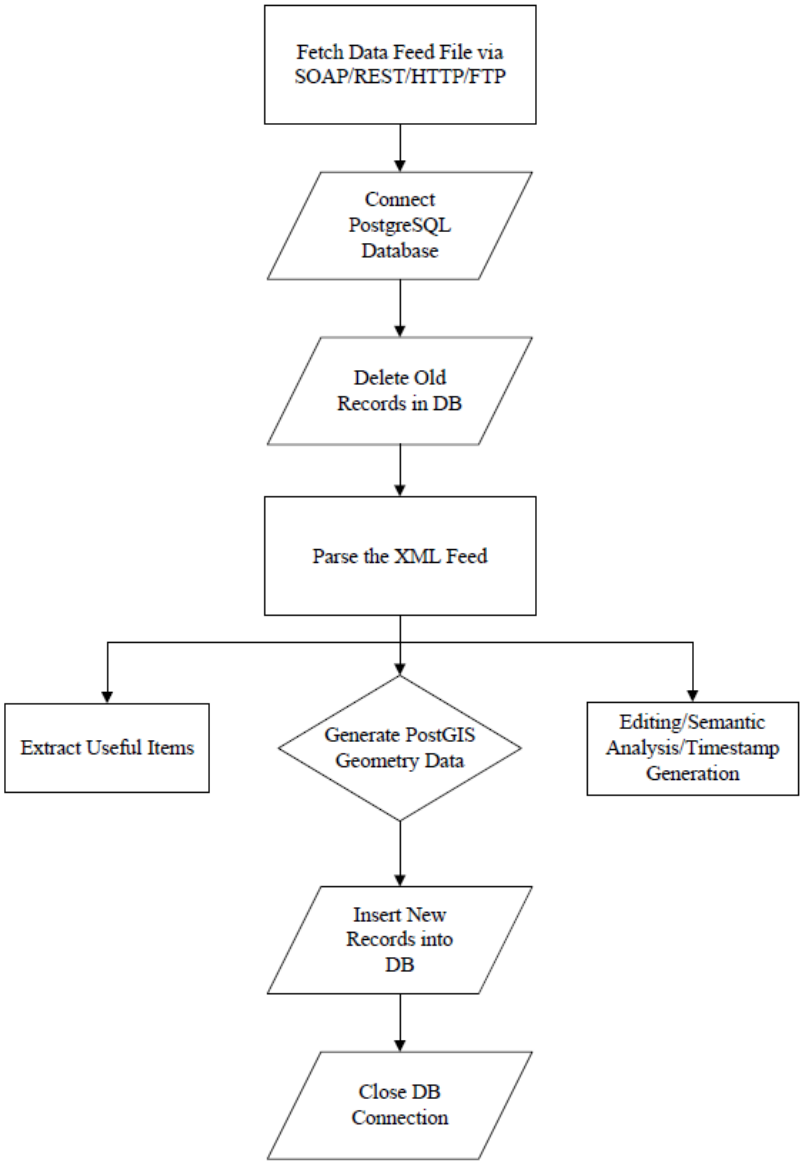


Figure 15 Flow chart of data parsing procedure

Fetch Data Feed

LWP::Simple module provides one of the simplest ways to make a HTTP GET request in language Perl. The *get($url)* function will fetch the document identified by the given URL and return it. The *$url* argument could be either a string or a reference to a URL object. A build-in function of Perl called *die* will raise an exception if the HTTP GET request failed. Similarly, another Perl module called *REST:Client* is used to fetch the data feed through RESTful web service. Before performing an HTTP GET request, a new client of RESTful web service needs to be constructed. Then the same GET request is to be sent to interact with the HTTP RESTful resources. Below is the sample code of initiating a new *REST::Client* and sending an HTTP GET request with basic authentication over HTTPS.

```
my $client = REST::Client->new();
$client->GET('http://wsdot.wa.gov/Traffic/api/
     highwayAlerts/HighwayAlertsREST.svc/
     GetAlertsAsXML?AccessCode={$accesscode}');
$content=$client->responseContent;
```

SOAP::Lite is the module we use for fetching data feed via Simple Object Access Protocol (SOAP). Similarly, a client needs to be constructed before sending any HTTP requests, though the constructor includes some more parameters. It is recommended to save the target data feed file locally for two reasons. Firstly, the saved local file would serve as a backup to the source. Second, it can make code debugging easier since we can easily look into the feed if it is needed.

Connect Database

An exception would occur if the data feed is not fetched successfully. Once the feed is fetched, the next task would be establishing database connection. We establish a connection to the PostgreSQL database using the Perl DBI module. The DBI module is an abstraction mechanism by which a Perl script can interact with a database with minimal regard for which DBMS is being used to host the database. The connection gives us a connection handle which is needed when calling other DBI functions. However, the DBI module itself does not have the ability to communicate with any specific DBMS. Instead it is necessary to install the appropriate backend module. In our case, it is *DBD::Pg*.

DBD::Pg is the Perl driver used to provide access to PostgreSQL databases, which works with the DBI module. The following syntax is the code to connect to a database with authentication.

```
$dbh=DBI->connect("dbi:Pg:dbname=$dbname;host=$host;
            port=$port;" , "$username", "$password");
```

Delete Old Records in Database

Why do we need to delete the old records in a given layer from a member state? This is because of the events' snapshot updating style. The data feed file contains not only the new reported events and the newly updated events, but also contains those unchanged, already existed events. In another word, every latest data feed include all reported concurrent and planned events. Instead of "incrementally" updating events, it is

easier and more efficient to update using snapshot style. So at the beginning of parsing a data feed, we use the following commands to delete old records existed in database.

```
sub hdl_start{
      my($p, $elt, $atts)=@_;
      $currentTag=$elt;
      If($currentTag eq "MSG_WisDOT_ITS"){
            $rows=$dbh->do("DELETE from crashincident where
org_id='WI'") or die $DBI::errstr;
      }
      ##more lines of code##
}
```

The code above is from the injector of Wisconsin *Incident* layer. The tag "MSG_WisDOT_ITS" is the beginning tag of the feed, from where the event updates start.

Parse the XML Feed

Since we already introduced the *XML::Parser* module in previous sections, we will only discuss some minor but important features of the module together with extra parsing strategies. Basically the structure of the XML is recognized by the parser by identifying the start tag and close tag in pairs. And the useful data is extracted by recognizing the non-markups in between the tags. All of those are realized by the following three important handlers [4].

Start (Expat, Element [, Attr, Val [,...]]). When an XML start tag is recognized, the start event will be generated immediately. Parameter *Expat* is the event based parser.

Element is the name of the XML element type that is opened with the start tag. The *Attr* and *Val* pairs are generated for each attribute in the start tag. It needs to be noticed that the database records are to be deleted when the start tag of the whole feed is recognized. A data feed file contains many events, the variables that are used to hold the needed records should be cleared each time when the start tag of an event is recognized.

End (Expat, Element). Similarly an event will be generated whenever an end tag named as *Element* is recognized. The end tag of an event needs particular attention. When it is recognized, it is the time to pass the data of extracted items to the variables which are supposed to hold the needed records. Also, most database activities such as new data insertions happen here. In most cases, the local server timestamp is used as the event updating time if the original event updating time is not provided. The code below is the method of grabbing the local server timestamp.

```
#get the local server timestamp
($second, $minute, $hour, $day_of_month, $month,
$year_offset, $day_of_week, $day_of_year,
$daylight_savings_flag) = localtime();
$year = 1900 + $year_offset;
#db date format for time stamp is mm/dd/yyyy hh:mm:ss
$month=$month+1;
$mes_server_timestamp="'$month/$day_of_month/$year
$hour:$minute:$second'";
```

Char (Expat, String). This event is generated when non-markup is recognized. This is the place where most data is extracted. Since not all those needed items in the feed is ready to use after being extracted, proper trimming and editing is indeed necessary. For example, the data input in the feed may in the format of *mmddyy* while the expected

output format is *mm/dd/yyyy*. Then the following code is necessary to perform the format transformation.

```
$mm = substr($str, 0, 2);
$dd = substr($str, 2, 2);
$yyyy = substr($str, 4, 4);
$end_time_date = "$mm/$dd/$yyyy";
```

Geometry Data Generation

An important component of the injector is the generation of the geometry data. Given the coordinates or mileposts we would need to translate them into the geometry type data that is ready to store in the geodatabase. At this step the coordinates or mileposts are already extracted, what needs to be done next is querying the database based on the provided parameters. And then pass the results of the query to the location variable. This is the case of generating road segments. However, if the location information of the event is only a point, then what needs to be done is simply wrapping the coordinates with the well-known text (WKT). The code listed below shows the process of generating road segments from milepost data.

```
Sub hdl_end{
      my($p, $elt)=@_;
      $currentTag=$elt;
      if($currentTag eq "SEGMENT"){
            $milepost=$beg_mp . "-" . $end_mp;
            if($signed_route eq "I-90"){
                  $sth=$dbh->prepare("SELECT location FROM
      mt_milepost_i90 where milepost='$milepost'");
            }elsif($signed_route eq "I-94"){
```

```
            $sth=$dbh->prepare("SELECT location FROM
      mt_milepost_i94 where milepost='$milepost'");
            }
      }

      $sth->execute() or die $DBI::errstr;
      While(my $ref=$sth->fetchrow_hashref()){
            $location=$ref->('location');
      }
      ##more lines of code##
}
```

The code above shows how to get the road segments on interstate 90 and 94 provided the begin milepost and end milepost, for the layer of *Road Condition* from Montana. And the database table mt_milepost_i90 and mt_milepost_i94 store the mileposts and corresponding segments in between.

At the end of each event in the data feed, it is about to update database tables once all records are parsed. If a single feed contains multiple layers, flags should be used to identify different layers together with extra parameters such as event id, organization id, update timestamp in order to keep well track of each update.

At last, when the parser reaches the end of the whole feed, database connection should be closed.

SUMMARY

This paper presents the approach to ingest the heterogeneous data from the Departments of Transportation (DOT) of the eight states that are involved with the N/WP corridor, and to ingest other heterogeneous data, from non-DOT sources, such as weather information, fuel station information, and recreational locations information. Most states provide XML or XML based data feeds, however, the data format and data structure from each state varies. Third-party data source provide various formatted data that ranges from XML, CSV, TXT, etc. This effort details the work performed to design the data injectors. The goal of the data injector is to process data from various sources periodically, of which the time interval is dependent on the source data updating frequency. With the data injector, heterogeneous data is fetched from various sources and is parsed on the server regardless of data structures and formats and then is pushed into the target database.

This work presents the functionalities of the data injectors and presents the strategies that the data injector uses to integrate heterogeneous data for future use of OTIIS project. By designing and implementing a common data processing framework based on industry standards, we developed a common data format that at least these eight member states could use to exchange and share their data. In addition, the developed common format will not only support the website but will also support the mobile app.

FUTURE WORK

A short term objective is to develop the mobile application version of OTIIS platform. A long term objective of future work is to encourage and facilitate data sharing between the states, and to encourage more states to join the OTIIS platform.

Specifically for the injectors, future work includes doing more complicated semantic analysis since sometimes semantic analysis is the only possible solution when the data is missing. Also, parsing speed of the injectors could further be improved, especially if more states are joining the OTTIS platform.

REFERENCES CITED

1. *National Digital Forecast Database*. Retrieved from NOAA National Weather Service: http://graphical.weather.gov/xml/

2. *Internet Engineering Task Force.* (2014, July). Retrieved from XML Media Types, RFC 7303.

3. *ArcGIS Resources*. Retrieved from ArcGIS Help 10.2.: http://resources.arcgis.com/en/help

4. CPAN. *XML-Parser-2.36*. Retrieved from http://search.cpan.org/~msergeant/XML-Parser-2.36/Parser.pm

5. Engineers, I. o. Retrieved from Traffic Management Data Dictionary: http://www.ite.org/standards/TMDD/accessed 4/16/10

6. SRID. Retrieved from wiki-srid: http://en.wikipedia.org/wiki/SRID

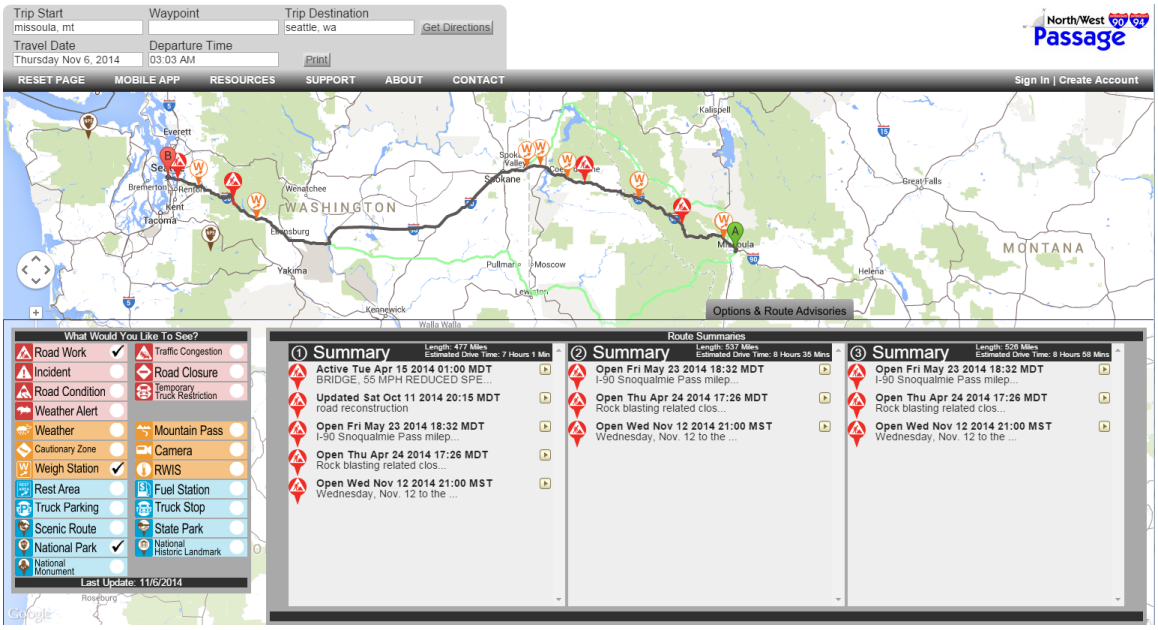
7. *W3C SOAP page*. Retrieved from http://www.w3.org/TR/soap/

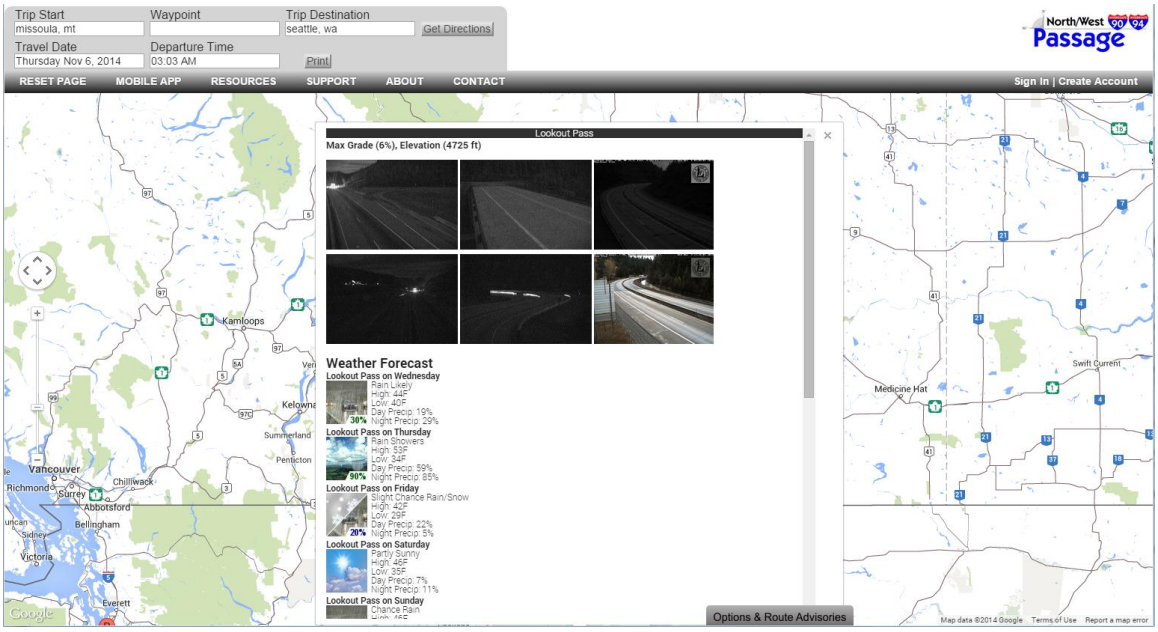
8. *W3C XML homepage*. Retrieved from http://www.w3.org/XML/

9. *wiki-wkt*. Retrieved from http://en.wikipedia.org/wiki/Well-known_text

APPENDIX A

Screenshots of OTIIS website


Trip Start
missoula, mt
Waypoint
Trip Destination
seattle, wa
Get Directions
Travel Date
Thursday Nov 6, 2014
Departure Time
03:03 AM
Print
North/West Passage
RESET PAGE
MOBILE APP
RESOURCES
SUPPORT
ABOUT
CONTACT
Sign In | Create Account
Options & Route Advisories
What Would You Like To See?
Road Work
Traffic Congestion
Incident
Road Closure
Road Condition
Temporary Truck Restriction
Weather Alert
Weather
Mountain Pass
Cautionary Zone
Camera
Weigh Station
RWIS
Rest Area
Fuel Station
Truck Parking
Truck Stop
Scenic Route
State Park
National Park
National Historic Landmark
National Monument
Last Update: 11/6/2014
Route Summaries
① Summary
Length: 477 Miles
Estimated Drive Time: 7 Hours 1 Min
Active Tue Apr 15 2014 01:00 MDT
BRIDGE, 55 MPH REDUCED SPE...
Updated Sat Oct 11 2014 20:15 MDT
road reconstruction
Open Fri May 23 2014 18:32 MDT
I-90 Snoqualmie Pass milep...
Open Thu Apr 24 2014 17:26 MDT
Rock blasting related clos...
Open Wed Nov 12 2014 21:00 MST
Wednesday, Nov. 12 to the ...
② Summary
Length: 537 Miles
Estimated Drive Time: 8 Hours 35 Mins
Open Fri May 23 2014 18:32 MDT
I-90 Snoqualmie Pass milep...
Open Thu Apr 24 2014 17:26 MDT
Rock blasting related clos...
Open Wed Nov 12 2014 21:00 MST
Wednesday, Nov. 12 to the ...
③ Summary
Length: 526 Miles
Estimated Drive Time: 8 Hours 58 Mins
Open Fri May 23 2014 18:32 MDT
I-90 Snoqualmie Pass milep...
Open Thu Apr 24 2014 17:26 MDT
Rock blasting related clos...
Open Wed Nov 12 2014 21:00 MST
Wednesday, Nov. 12 to the ...
Everett
Seattle
Bremerton
Renton
Tacoma
Olympia
Wenatchee
WASHINGTON
Ellensburg
Yakima
Kennewick
Walla Walla
Pullman
Moscow
Lewiston
Spokane
Kalispell
Great Falls
MONTANA
Helena
Missoula
Roseburg
Google


Trip Start
missoula, mt
Waypoint
Trip Destination
seattle, wa
Get Directions
Travel Date
Thursday Nov 6, 2014
Departure Time
03:03 AM
Print
North/West Passage
RESET PAGE
MOBILE APP
RESOURCES
SUPPORT
ABOUT
CONTACT
Sign In | Create Account
Lookout Pass
Max Grade (6%), Elevation (4725 ft)
Weather Forecast
Lookout Pass on Wednesday
Rain Likely
High: 44F
Low: 40F
Day Precip: 19%
Night Precip: 29%
Lookout Pass on Thursday
Rain Showers
High: 53F
Low: 34F
Day Precip: 59%
Night Precip: 85%
Lookout Pass on Friday
Slight Chance Rain/Snow
High: 42F
Low: 29F
Day Precip: 22%
Night Precip: 5%
Lookout Pass on Saturday
Partly Sunny
High: 46F
Low: 35F
Day Precip: 7%
Night Precip: 11%
Lookout Pass on Sunday
Chance Rain
Options & Route Advisories
Kamloops
Kelowna
Summerland
Penticton
Vancouver
Richmond
Surrey
Chilliwack
Abbotsford
Bellingham
Victoria
Everett
Swift Current
Medicine Hat
Google
Map data ©2014 Google
Terms of Use
Report a map error