UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo dell'interoperabilità e del linting in relazione a GeoJSON nel framework Qt

Design and development in Qt framework of GeoJSON interoperability and linting

Relatore:
Prof. Adriano Mancini

Tesi di Laurea di:
Juljan Sherollari

Anno Accademico 2018-2019

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

Non si può mai attraversare l'oceano se non si ha il coraggio di perdere di vista la riva.

You can never cross the ocean if you're not brave enough to lose sight of the shore.

Cristoforo Colombo

Abstract

Il presente elaborato è basato sul progetto presentato per il Google Summer of Code 2018, relativo alle librerie geospaziali del framework Qt. L'obiettivo del progetto è stato quello di rendere la libreria Qt Location del framework interoperabile con lo standard GeoJSON.

Il modulo Qt Location è un componente del framework Qt che offre funzionalità che riguardano la rappresentazione di luoghi geografici su una mappa. I dati sulla posizione di questi luoghi, espressi in coordinate geografiche, vengono memorizzati in classi che rappresentano specifiche forme geometriche.

GeoJSON è uno standard aperto, che utilizza il sistema di riferimento di coordinate geografiche WGS84, per rappresentare geometrie spaziali i cui attributi sono descritti attraverso JavaScript Object Notation. Secondo il documento che lo definisce, RFC 7946[27], le geometrie rappresentabili sono punti, linee spezzate, poligoni, o collezioni multiple di queste tipologie. Per riuscire a importare ed esportare correttamente file GeoJSON in Qt e rendere i due sistemi interoperabili è stato necessario rendere rappresentabili biunivocamente le geometrie dello standard con quelle del framework. Per il conseguimento della equivalenza tra le caratteristiche supportate dalle geometrie GeoJSON e da quelle presenti nella libreria Qt Location sono state svolte 3 azioni principali:

1. **Feature parity:** Sono stati modificati i file sorgenti di Qt Location, dal momento che esso supporta solo parzialmente lo standard GeoJSON, in particolare è stato aggiunto il supporto ai buchi nelle geometrie poligono.

2. **Importing:** È stata realizzata una funzione importer che permette di leggere e rappresentare un documento GeoJSON in una struttura dati opportunamente modellata, contenente le geometrie Qt Location equivalenti a quelle presenti nel documento GeoJSON di partenza.

3. **Exporting:** È stata realizzata una funzione exporter che permette di leggere la struttura dati generata dell'importer ed estrarre un documento GeoJSON valido.

Operativamente è stata sviluppata una nuova classe QGeoJson[31] dotata di due metodi che implementano le funzioni esposte nei punti 2 e 3.

Nella documentazione realizzata a dotazione della classe, è stata ampiemente illustrata l'organizzazione dalla struttura dati contenente le geometrie importate (nel caso dell'importer) o le geometrie da esportare (nel caso dell'exporter); inoltre sono stati sviluppati due ulteriori metodi, uno per validare il file da importare e uno per per visualizzare la struttura dati importata (scopo di debug).

Per prima cosa sono state analizzate le strutture Qt più adatte all'equivalenza con gli oggetti GeoJSON. In particolare, il lavoro sul codice già esistente, è consistito nell'aggiunta del supporto ai buchi per le geometrie di tipo poligono, al fine di ottenere l'equivalenza con gli oggetti Polygon e MultiPolygon di GeoJSON.

Corrispondenze tra Qt Location e GeoJSON		
#	Oggetti GeoJSON	Classi Geometrie Qt
1	Point	QGeoCircle
2	LineString	QGeoPath
3	Polygon	QGeoPolygon
4	MultiPoint	QVariantList(QGeoCircle)
5	MultiLineString	QVariantList(QGeoPath)
6	MultiPolygon	QVariantList(QGeoPolygon)
7	GeometryCollection	QVariantList(Varie Geometrie)
8	Feature	Geometria + attributo "properties"
9	FeatureCollection	QVariantList(Feature)

Sviluppo della classe QGeoJson

La classe QGeoJson ha 4 metodi:

1. Un **importer** che accetta un QJsonDocument e restituisce una QVriantList.

2. Un **exporter** che accetta la QVarianList di cui sopra e restituisce un QJsonDocument.

3. Un **linter** per la validazione del QJsonDocument, secondo lo standard GeoJSON.

4. Un metodo per la visualizzazione della QVariantList a scopi di debug.

Prototipi di funzione

```
// This method importe a GeoJSON file to QVariantList:
static QVariantList importGeoJson(const QJsonDocument &doc);

// This method exporte a GeoJSON file from a QVariantList:
static QJsonDocument exportGeoJson(const QVariantList &list);

// This method performs validation on the input:
static bool isValidGeoJson(const QJsonDocument &geojson,
QJsonParseError *err = nullptr);

// This method prints the content of the imported QVariantList:
static QString toString(const QVariantList &importedGeoJson);
```

Importer del GeoJSON

L'importer accetta un QJsonDocument dal quale viene estratto un singolo oggetto, poichè secondo l'RFC[27] un documento GeoJSON è sempre costituito da un singolo oggetto JSON, e restituisce una QVariantList per mantenere la compatibilità con il QML. La QVariantList contiene sempre una singola QVariantMap, quest'ultima è la vera geometria importata. La map contiene sempre almeno due coppie (chiave, valore), la prima avente come chiave una QString "type" e come corrispondente valore una QString identificante gli oggetti GeoJSON definiti nell'RFC (vedi tabella), la seconda coppia ha come chiave una QString "data" e contiene il valore delle geometrie Qt ottenute dall'importazione, e strutturate in un modo ben specifico per garantire la visualizzazione delle geometrie in una mappa in modo semplice. Per le singole geometrie (Point, LineString, Polygon) il valore corrispondente alla chiave "data" è un QGeoShape. Per le geometrie multiple omogenee (MultiPoint, MultiLineString, MultiPolygon) il valore corrispondente alla chiave "data" è una QVariantList. Ogni elemento della QVariantList è una QVariantMap delle singole geometrie, tutte dello stesso tipo. La "GeometryCollection" è una composizione eterogenea di gemetrie. Il valore corrispondente alla chiave "data" è una QVariantList popolata da QVariantMaps delle precedenti geometrie e anche di GeometryCollection stessa. L'oggetto di tipo "Feature" include una delle precedenti entità più il membro "properties". Il valore di questo membro è una QVariantMap. L'unico modo per distinguere una Feature da una geometria è controllare se è presente il nodo "properties" nella QVariantMap. Infatti la presenza del membro properties, anche se vuoto, è obbligatorio in un oggetto Feature GeoJSON. La FeatureCollection è una composizione di Feature.

Il valore corrispondente alla chiave "data" è una QVariantList popolata da QVariantMaps.

Exporter del GeoJSON

L'exporter accetta in ingresso una QVariantList strutturata come spiegato sopra, e restituisce un QJsonDocument. Come è facile comprendere, nel caso in cui venisse processata attraverso l'exporter la QVariantList prodotta dall'importazione di un dato documento GeoJSON, il medesimo exporter restituirebbe un documento equivalente all'originale dal punto di vista dei dati contenuti, ma non necessariamente della forma con cui questi vengono esposti.

Validazione del GeoJSON

Il linter è un metodo che prende in ingresso un QJsonDocument e restituisce un valore booleano "true", nel caso di un documento GeoJSON corretto, e "false" nel caso di un documento invalido. Qt offre la possibilità di validare un file JSON e di segnalare l'eventuale errore mediante l'utilizzo di una enum, lo stesso modello è stato adottato in fase di progettazione del metodo di validazione.

Debug della struttura dati

Il metodo toString consiste in un secondo exporter creato a scopo di debug. Esso restituisce una rappresentazione conforme allo standard JSON, della QVariantList generata dall'importer. Questa funzione ci consente di avere una rappresentazione leggibile della struttura dati costituita da QVariantList e QVariantMap nidificate molteplici volte. Nella rappresentazione JSON ad ogni QVariantList corrisponde un "array" e a ogni QVariantMap un "object".

Tutti i metodi illustrati, tranne il metodo che si occupa della validazione del GeoJSON, sono stati pubblicati in una nuova classe inserita all'interno della libreria QtLocation, e inclusi nel framework con qualifica di classe sperimentale dalla verione 5.13. Oltre allo sviluppo della Classe di importazione ed esportazione, e alla modifica della classe QGeoPolygon esistente, sono stati realizzati gli opportuni autotest per la verifica del funzionamento, ed è stata scritta una app di test in QML per visualizzazione di elementi GeoJSON su una mappa in QtLocation.

Test di esempio

È stata creata un app di esempio, allo scopo di illustrare il corretto funzionamento della classe sviluppata. L'applicazione permette di visualizzare su una mappa le geometrie importate da un file GeoJSON, utilizzando la funzione di importer ed una funzione che, partendo da una visualizzazione di geometrie su una mappa generata utilizzando il framework Qt, utilizzando l'exporter restituisca la stessa struttura dati.

Contents

Chapter 1

Introduction

The project of this thesis had its start with the Google program called Google Summer of Code. While looking for an opportunity to improve my skills as a developer I enrolled the Google Summer of Code program for the year 2018.

1.1 Google Summer of Code

Google Summer of Code, often abbreviated to GSoC [21], is an international annual program, first held from May to August 2005[22]. The program is open to university students and it is focused on bringing more students into open source software development. Students work on a three month programming project with an open source organization.

The project follows a very specific timeline which contemplates open source projects to apply to be mentor organizations. Once accepted, organizations discuss possible ideas with students and then decide on the proposals they wish to mentor for the summer. They provide mentors to support each student through the program. Existing contributors with the organizations can choose to mentor a student project. Mentors and students work together to determine appropriate milestones and requirements for the summer. Students contact the open source organizations they want to work with and write up a project proposal for the summer. If accepted, students spend a month integrating with their organizations prior to the start of coding. Students then have three months to code while meeting the deadlines agreed upon with their mentors.

1.2 Mentor Organization

Among other organization, Qt[20] has been chosen. The Qt Company is a global software company focused on the development of a cross-platform framework, compatible with a large set of operating systems and hardware which enables a single software code across many operating systems, platforms and screen types, from desktops and embedded systems. Qt is present in more than 70 industries and its technology is used by approximately one million developers worldwide. The company's net sales in year 2018 totaled 45.6 MEUR[2] and it achieves this through its cross-platform software framework for the development of apps and devices, under both commercial and open source licenses. The Qt framework, and Qt toolset is being developed with the aim to allow software teams to develop faster, by supporting the full cycle of Prototyping – Development – Testing – Deployment. Qt also supports regulatory and compliance efforts through its internal resources or its industry-leading partner network. The choice of Qt offered many more benefits, from the opportunity to engage with the use of powerful version control and collaboration tools, in fact the whole development process of the framework is managed through git and Gerrit Code Review tools. On top of all this Qt offered the chance to deal with the development of commercial grade APIs. Between many different ideas proposed by Qt for the GSoC project, my choice fell on the one which required to work on Qt Location, the Qt Framework geographic library. The goal of the project was to make Qt Location interoperable with GeoJSON by adding support for loading geometries from GeoJSON into Qt Location and exporting geometries from QtLocation to GeoJSON. Before doing this, required features should have been added to Qt Location, to reach feature parity with GeoJSON. By supporting GeoJSON import/export it would have become much easier to distribute feature files to end users, and to backup user-defined items. After the Project with google was completed, emerged the usefulness of a tool for GeoJSON validation, thus a linter was designed beyond the scope of the Google Summer of Code. The structure of this work begins in chapter 2 with a detailed illustration of what Qt Framework is and of its architecture, with a special focus on the Qt geographical modules in chapter 3. Then, in chapter 4, I will summarily discuss about JSON and GeoJSON. Chapter 5 will present the design of the feature parity between GeoJSON and the discussed Qt geographical libraries, while chapter 6 will get across the development phase of the project. Chapter 7 is a brief conclusion to this work.

Chapter 2

Reference Framework

Qt (pronounced as "cute", not "cu-tee") is a cross-platform application development framework for desktop, embedded and mobile. Qt is not a programming language on its own. It is a framework written in C++. Development of Qt was started in 1990 by the Norwegian programmers Eirik Chambe-Eng and Haavard Nord [15]. Their company, Trolltech, that sold Qt licenses and provided support, went through several acquisitions over the years. Today former Trolltech is named The Qt Company and is a wholly owned subsidiary of Digia Plc., Finland. Although The Qt Company is the main driver behind Qt, Qt is now developed by a bigger alliance: The Qt Project[20]. It consists of many companies and individuals around the globe and follows a meritocratic governance model. Everyone who wants to, individuals and companies, can join the effort. There are many ways one can contribute to the Qt Project, e.g. by writing code or documentation for the framework, reporting bugs, helping other users on the forum or maintaining pages on the wiki. Qt is available under various licenses: The Qt Company sells commercial licenses, but Qt is also available as free software under several versions of the GPL.

2.1 Qt framework

Qt framework is usually used as a graphical toolkit, although it is also possible to create command line applications. The framework[12] is structured in modules. The base modules are called "Qt Essentials" and define the foundation of Qt on all platforms. They are available on all supported development platforms and on the tested target platforms. The following table lists the most relevant Qt essentials modules[11]:

Qt Essentials Modules	
Module	**Description**
Qt Core	Core non-graphical classes used by other modules
Qt GUI	Base classes for graphical user interface (GUI) components. Includes OpenGL
Qt Multimedia	Classes for audio, video, radio and camera functionality
Qt Widgets	Widget-based classes for implementing multimedia functionality
Qt Network	Classes to make network programming easier and more portable
Qt QML	Classes for QML and JavaScript languages.
Qt Quick	A declarative framework for building highly dynamic applications with custom user interfaces
Qt Quick Controls	Provides lightweight QML types for creating performant user interfaces for desktop, embedded, and mobile devices. These types employ a simple styling architecture and are very efficient
Qt Quick Dialogs	Types for creating and interacting with system dialogs from a Qt Quick application
Qt Quick Layouts	Layouts are items that are used to arrange Qt Quick 2 based items in the user interface
Qt Quick Test	A unit test framework for QML applications, where the test cases are written as JavaScript functions
Qt SQL	Classes for database integration using SQL
Qt Test	Classes for unit testing Qt applications and libraries
Qt Widgets	Classes to extend Qt GUI with C++ widgets

Essential modules are general and useful for a majority of Qt applications. A module that is used for a special purpose is considered an add-on module even if it is available on all supported platforms. Qt Add-On modules bring additional value for specific purposes. These modules may only be available on some development platform. Many add-on modules are either feature-complete and exist for backwards compatibility, or are only applicable to certain platforms. Each add-on module specifies its compatibility promise separately.

The following table lists some of the Qt add-ons:

Add-On Modules	
Module	**Description**
Qt Location	Displays map, navigation, and place content in a QML application
Qt Positioning	Provides access to position, satellite and area monitoring classes
Qt 3D	Functionality for near-real-time simulation systems with support for 2D and 3D rendering
Qt Bluetooth	Provides access to Bluetooth hardware
Qt Concurrent	Classes for writing multi-threaded programs without using low-level threading primitives
Qt Purchasing	Enables in-app purchase of products in Qt applications
Qt NFC	Provides access to Near-Field communication (NFC) hardware
Qt OpenGL	Deprecated in favor of the QOpenGL* classes in the Qt GUI module
Qt Sensors	Provides access to sensor hardware and motion gesture recognition
Qt WebSockets	Provides WebSocket communication compliant with RFC 6455

Qt XML	C++ implementations of SAX and DOM
Qt Help	Classes for integrating documentation into applications, similar to Qt Assistant

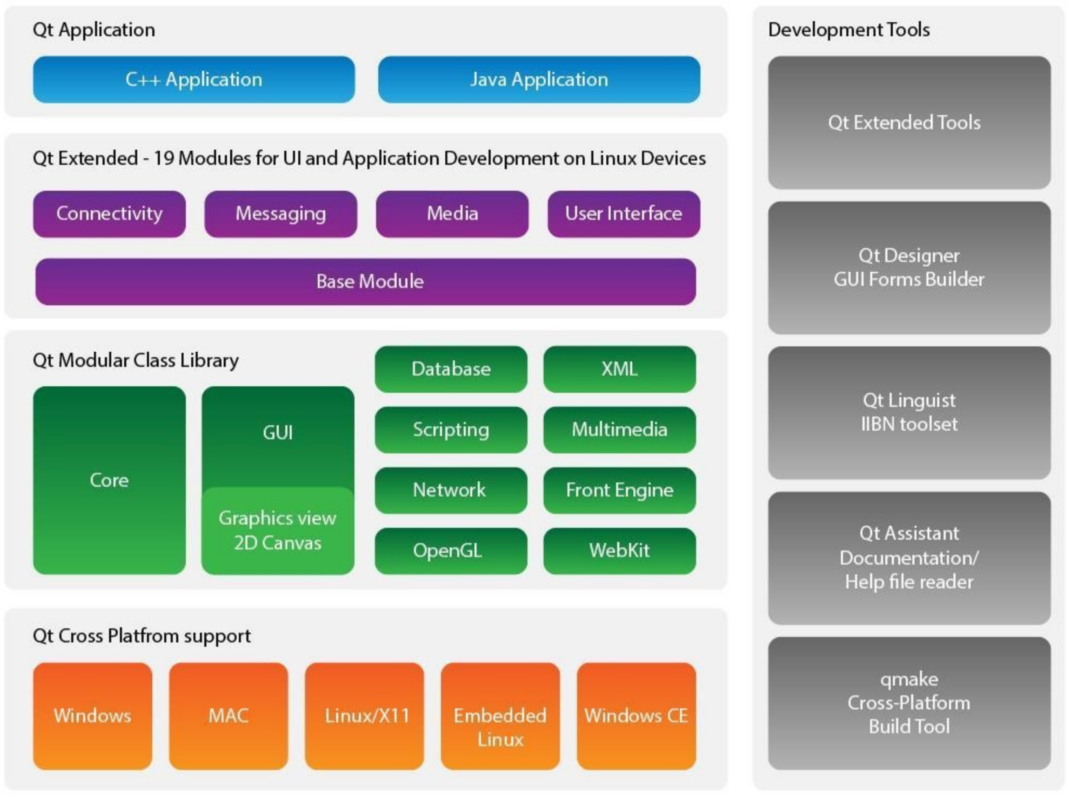
In addition to the modules released as part of Qt 5, the following modules and tooling build on top of the Qt libraries to provide additional value. They have their own release schedule, and are available under the commercial license.

Value-Add Modules	
Feature	**Description**
Qt Automotive Suite	A collection of software components and tools to enable development of In-Vehicle-Infotainment systems
Qt for Automation	Libraries and tools for automation related domains, such as KNX, OPC UA, and MQTT
Qt for Device Creation	Tools for fast, easy, and fully-integrated embedded device application development. Included in most other value-add solutions

To facilitate the development and design of application on all the supported development platforms, Qt offers a rich set of development tools called Qt Tools. The most important of these tools is the Qt Creator Integrated Development Environment (IDE). Qt Creator provides tools for accomplishing tasks throughout the whole application development life-cycle, from creating a project to deploying the application on the target platforms. Some interesting tools are integrated into Qt Creator, Qt Designer for designing and building graphical user interfaces (GUIs) from Qt widgets in a visual editor; qmake for building applications for different target platforms; Qt Linguist for localizing applications, which contains tools for the roles typically involved in localizing applications: developers, translators; Qt Assistant for viewing Qt documentation. You can also view documentation in Qt Creator.

2.2 Architecture

The architecture of Qt class library is based on a custom Object Model. The standard C++ object model provides very efficient runtime support for the object paradigm. But its static nature is inflexible in certain problem domains. Graphical user interface programming is a domain that requires both runtime efficiency and a high level of flexibility. The Qt Object Model has been developed to achieve this flexibility without giving up the speed of C++.



Qt adds these features to C++:

- a very powerful mechanism for seamless object communication called signals and slots;
- queryable and designable object properties;
- powerful events and event filters;
- contextual string translation for internationalization;
- interval driven timers that make it possible to elegantly integrate many tasks in an event-driven GUI;
- hierarchical and queryable object trees that organize object ownership;
- guarded pointers (QPointer) that are automatically set to 0 when the referenced object is destroyed, unlike normal C++ pointers which become dangling pointers when their objects are destroyed;
- a dynamic cast that works across library boundaries.

Many of these Qt features are implemented using standard C++ techniques, based on inheritance from QObject. Others, like the object communication mechanism (like signals and slots) and the dynamic property system, require the Meta-Object System provided by Qt's own Meta-Object Compiler (MOC). The MOC (Meta-Object Compiler), is a preprocessor, used to extend the C++ language with features. Before the compilation step, the MOC parses the source files written in Qt-extended C++ and generates standard compliant C++ sources from them. Thus the framework itself and applications/libraries using it can be compiled by any standard C++ compiler like Clang, GCC, ICC, MinGW and MSVC[3]. Some of the added features listed above for the Qt Object Model, ground on the idea that Qt Objects are meant to be thought as identities, not values. Values are copied or assigned; identities are cloned. Cloning means to create a new identity, not an exact copy of the old one. Since a Qt Object might have a unique `QObject::objectName()`, we can have problems if we try to copy a Qt Object. A Qt Object has a specific location in an object hierarchy, plus it can be connected to other Qt Objects to emit signals to them or to receive signals emitted by them. It can have new properties added to it at runtime that are not declared in the C++ class. For these reasons, Qt Objects should be treated as identities, not as values. Identities are cloned, not copied or assigned, and cloning an identity is a more complex operation than copying or assigning a value. Therefore, `QObject` and all subclasses of `QObject` (direct or indirect) have their copy constructor and assignment operator disabled.

The meta-object system is a C++ extension that makes the language better suited to true component GUI programming. Although templates can be used to extend C++, the meta-object system provides benefits using standard C++ that cannot be achieved with templates.

The meta-object system is based on three things:

- the `QObject` class provides a base class for objects that can take advantage of the meta-object system;
- the Q_OBJECT macro inside the private section of the class declaration is used to enable meta-object features, such as dynamic properties, signals and slots;
- the Meta-Object Compiler supplies each `QObject` subclass with the necessary code to implement meta-object features.

The MOC tool reads a C++ source file. If it finds one or more class declarations that contain the Q_OBJECT macro, it produces another C++ source file which contains the meta-object code for each of those classes. This generated source file is either #include'd into the class's source file or, more usually, compiled and linked with the class's implementation.

Qt's meta-object system provides the signals and slots mechanism for inter-object communication, run-time type information, and the dynamic property system. Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks. Signals and slots are made possible by Qt's meta-object system.

In GUI programming, when one widget changes, is often required another widget to be notified. More generally, it would be useful for objects of any kind to be able to communicate with one another. For example, if a user clicks a close button, we probably want the window's `close()` function to be called.

Other toolkits achieve this kind of communication using callbacks. A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback when appropriate. While successful frameworks using this method do exist, callbacks can be not-intuitive and may suffer from problems in ensuring the type-correctness of callback arguments.

In Qt, there is an alternative to the callback technique: the use of signals and slots. A signal is emitted when a particular event occurs. Qt's widgets have many predefined signals, but we can always subclass widgets to add our own signals to them. A slot is a function that is called in response to a particular signal. Qt's widgets have many pre-defined slots, but it is common practice to subclass widgets and add your own slots so that you can handle the signals that you are interested in.

Qt also provides a sophisticated internal property system similar to the ones supplied by some compiler vendors. However, as a compiler and platform independent library, Qt does not rely on non-standard compiler features like `__property` or `[property]`[8]. The Qt solution works with any standard C++ compiler on every platform Qt supports. It is based on the Meta-Object System that, like said, also provides inter-object communication via signals and slots. To declare a property, use the Q_PROPERTY() macro in a class that inherits QObject.

2.3 Qt Quick

One of the most recent and most important Qt modules is Qt Quick. First introduced in Qt 4.7 and in Qt Creator 2.1, is a high-level UI technology that allows developers and UI designers to work together to create animated, touch-enabled UIs and lightweight applications. It includes new tools in the Qt Creator IDE, including a visual editor that allows UI designers and developers to cooperate, working on the same code in an iterative approach. It also includes QtDeclarative, a new module in the Qt library that enables a new declarative programming approach, and it includes QML, Qt Meta-Object Language, an easy to use, declarative language. No C++ programming skills are needed to use Qt Quick, yet it is totally based on Qt and can be extended from C++.

2.4 Qt QML Module

The Qt QML module provides a framework for developing applications and libraries with the QML language. It defines and implements the language and engine infrastructure, and provides an API to enable application developers to extend the QML language with custom types and integrate QML code with JavaScript and C++. The Qt QML module provides both a QML API and a C++ API. Note that while the Qt QML module provides the language and infrastructure for QML applications, the Qt Quick module provides many visual components, model-view support, an animation framework, and much more for building user interfaces.

2.5 Qt Meta-Object Language

QML is a multi-paradigm language based on JavaScript, built to create highly dynamic applications. With QML, application building blocks such as UI components are declared and various properties set to define the application behavior. Application behavior can be further scripted through JavaScript, which is a subset of the language. In addition, QML heavily uses Qt, which allows types and other Qt features to be accessible directly from QML applications. As multi-paradigm language, QML enables objects to be defined in terms of their attributes and how they relate and respond to changes in other objects. In contrast to purely imperative code, where changes in attributes and behavior are expressed through a series of statements that are processed step by step, QML's declarative syntax integrates attribute and behavioral changes directly into the definitions of individual objects. These attribute definitions can then include imperative code, in the case where complex custom application behavior is needed.

```
import QtQuick 2.3

Rectangle {
    width: 300
    height: 100
    color: "red"

    Text {
        anchors.centerIn: parent
        text: "Hello,␣World!"
    }
}
```

Listing 2.1: Example of QML code

QML Object Types

A QML object type is a type from which a QML object can be instantiated. In syntactic terms, a QML object type is one which can be used to declare an object by specifying the type name followed by a set of curly braces that encompasses the attributes of that object. This differs from basic types, which cannot be used in the same way. For example, Rectangle is a QML object type: it can be used to create Rectangle type objects. This cannot be done with primitive types such as `int` and `bool`, which are used to hold simple data types rather than objects. Custom QML object types can be defined by creating a .qml file that defines the type, or by defining a QML type from C++ and registering the type with the QML engine. Note that in both cases, the type name must begin with an uppercase letter in order to be declared as a QML object type in a QML file.

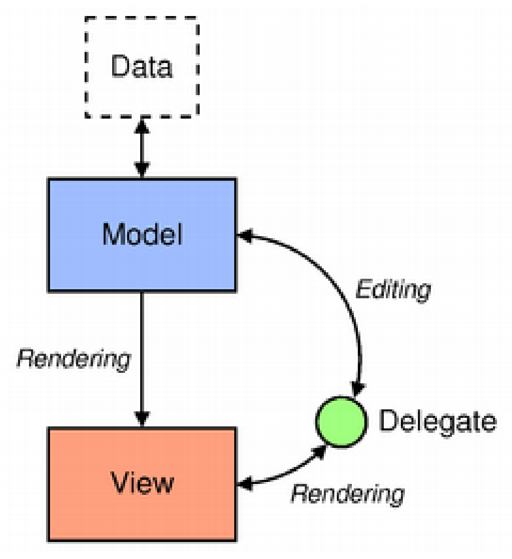
The QML module contains types for defining data models in QML.

QML Types	
Types	**Description**
DelegateModel	Encapsulates a model and delegate
DelegateModelGroup	Encapsulates a filtered set of visual data items
Instantiator	Dynamically creates objects
ItemSelectionModel	Instantiates a QItemSelectionModel to be used in conjunction with a QAbstractItemModel and any view supporting it
ListElement	Defines a data item in a ListModel
ListModel	Defines a free-form list data source
ObjectModel	Defines a set of items to be used as a model
Package	Specifies a collection of named items

Experimental QML Types	
Types	**Description**
DelegateChoice	Encapsulates a delegate and when to use it
DelegateChooser	Allows a view to use different delegates for different types of items in the model
TableModel	Encapsulates a simple table model
TableModelColumn	Represents a column in a model

2.6 Model/View Programming

Qt contains a set of item view classes that use a model/view architecture to manage the relationship between data and the way it is presented to the user. The separation of functionality introduced by this architecture gives developers greater flexibility to customize the presentation of items, and provides a standard model interface to allow a wide range of data sources to be used with existing item views.



The Model/View architecture

Model-View-Controller (MVC) is a design pattern originating from Smalltalk[34] that is often used when building user interfaces. MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input.

Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse. If the view and the controller objects are combined, the result is the Model/View architecture. This still separates the way that data is stored from the way that it is presented to the user, but provides a simpler framework based on the same principles. This separation makes it possible to display the same data in several different views, and to implement new types of views, without changing the underlying data structures. To allow flexible handling of user input, Qt introduces the concept of the delegate. The main advantage of using a delegate is the possibility to customize data rendering and editing. The model communicates with a source of data, providing an interface for the other components in the architecture. The nature of the communication depends on the type of data source, and the way the model is implemented. The view obtains indexes from the model; these are references to items of data. By supplying indexes to the model, the view can retrieve items of data from the data source. In standard views, a delegate renders the items of data. When an item is edited, the delegate communicates with the model directly using model indexes.

Generally, the model/view classes can be separated into the three groups: models, views, and delegates. Each of these components is defined by abstract classes that provide common interfaces and, in some cases, default features implementations. Abstract classes are meant to be sub-classed in order to provide the full set of functionality expected by other components and also allows specialized components to be written.

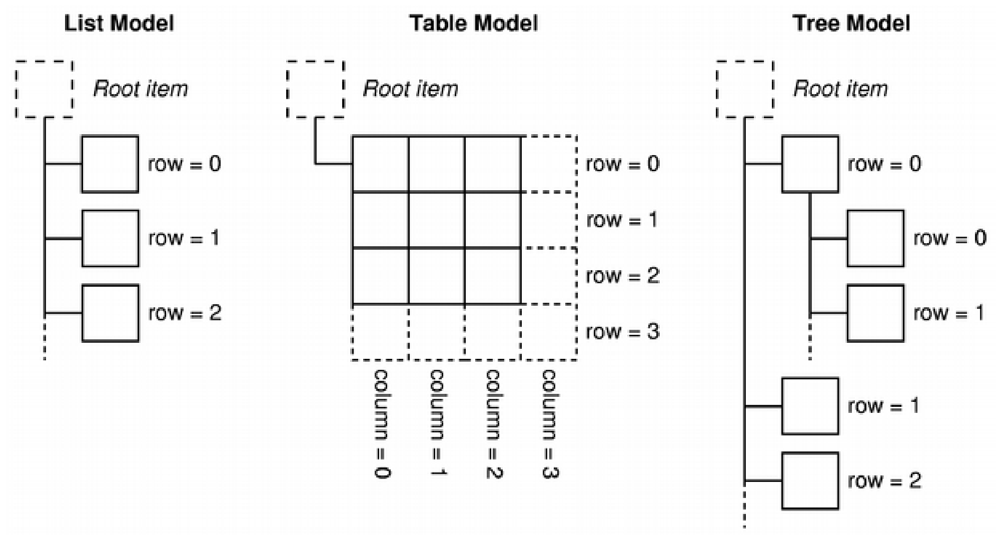


Figure 2.1: Basic concepts surrounding models.

Chapter 3

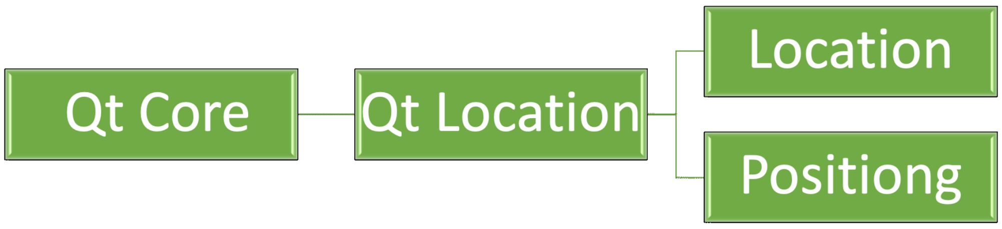
Qt Geographical APIs

Essential Qt modules are general and useful for a majority of Qt applications. A module that is used for a special purpose is considered an add-on module even if it is available on all supported platforms. Qt Location and Qt Positioning , are the geographical add-on modules of the Qt framework[11].

1. Location API provides a library for mapping, navigation and place information.

2. Positioning module provides positioning information via QML and C++ interfaces.

Qt Loction API is split into sub-modules, which provide QML and C++ interfaces for specific purposes. The required position data can be retrieved using the Qt Positioning module.



3.1 Qt Location API

The Qt Location API offers the classes needed to create mapping solutions using the data available from some of the most popular location services[13]. It allows access to map data and presenting it, Qt Location allows the user to add additional geometric layers to the map, query for a specific geographical location and route[13]. The API is split into sub-modules, which focus mainly on Map and Place information:

Location APIs	
Maps and Navigation	Displaying maps and finding routes
Places	Searching for and managing points of interest
Geoservices Plugin	Implement geoservices and positioning plugins

Currently it is not possible to handle user interaction with maps data via C++, the only available interface is the Maps and Navigation (QML) API.

3.1.1 Maps and Navigation API

The module provides the QML and C++ alternatives for maps and navigation. The C++ alternative provides utility classes to get geocoding (finding a geographic coordinate from a street address) and navigation (including driving and walking directions) information, whereas its QML counterpart provides UI components to render the information.

Maps and Navigation (QML)

Maps and Navigation provides Qt Quick user interface types for displaying geographic information on a map, as well as allowing user interaction with map overlay objects and the display itself. It also contains utilities for geocoding and navigation. Maps can also contain map overlay objects, which are used to display information on its surface. There is a set of basic pre-defined map overlay objects, as well as the ability to implement custom map overlay objects using the `MapQuickItem` type, which can contain any standard Qt Quick item. To automatically generate map overlay objects based on the contents of a

Qt Quick model (for example a `ListModel` item), the `MapItemView` type is available. It accepts any map overlay object as its delegate, and can only be created within a Map.

Displaying Maps

Displaying a map is done using the Map QML types. The Map type supports user interaction through the `MapGestureArea` QML type. The Map object draws the map on-screen using OpenGL, allowing for hardware-accelerated rendering where available. Provided that a position has been obtained, the Qt Location module can add a Map with Places of Interest (POI) and Places. The user can be made aware of nearby features and related information, displayed on the map. These features can be places of business, entertainment, and so on. They may include paths, roads, or forms of transport, enabling navigation optimization and assistance. The Map QML types enable the information contained in Place objects to be displayed, panned, zoomed, and so on. In the following table are listed the relevant Map QML Types

QML Types	
MapItemView	Populates a Map with map overlay objects based on the data provided by a model
MapCircle	Define a geographic circle (all points at a set distance from a center), optionally with a border
MapPolyline	Define a polyline made of an arbitrary list of coordinates
MapPolygon	Define a polygon made of an arbitrary list of coordinates
MapRectangle	Define a rectangle whose top left and bottom right points are specified as coordinate types, optionally with a border
MapQuickItem	Turns any arbitrary Qt Quick Item into a map overlay object

3.1.2 Places API

This API allows users to discover places/points of interest and view details about them such as address and contact information; some places may even have rich content such as images and reviews. The Places API also facilitates management of places and categories, allowing users to save and remove them. A `QPlace` object represents a place by acting as a container for various information about that place. Besides the source information, the API provides information about the location, size, and other related information. The Places API can also retrieve images, reviews, and other content related to a place[14]. To access data from REST servers or places from a local database Plugins are used. A Plugin is an abstraction for a backend. In the following table there is a list of the Qt Location Map plugins.

Qt Location Plugin	
Esri	Uses Esri for location services
HERE	Uses the relevant services provided by HERE
Items Overlay	Provides an empty map intended to be used as background for an overlay layers for map items
Mapbox GL	Uses Mapbox GL for location services
Mapbox	Uses Mapbox for location services
Open Street Map	Uses Open Street Map and related services

The QML Places API is built around the notion of models, views and delegates as illustrated in the Chapter 2. The Model holds data items and maintains their structure. It is also responsible for retrieving the items from a data source. The View is a visual container that displays the data and manages how visual items are shown such as in a list or a grid. It may also be responsible for navigating the data, for example, scrolling through the visual items during a flicking motion[17]. The Delegate defines how individual data elements should appear as visual items in the view. The models expose a set of data roles and the delegate uses them to construct a visual item. It may also define behaviour such as an operation to invoke when a visual item is clicked.

3.2 Qt Positioning API

The Positioning module is the natural complement to the Places submodule of Qt Location. This API gives developers the ability to determine a position by using a variety of possible sources, including satellite, or wifi, or text file, and so on. That information can then be used to for example determine a position on a map. In addition satellite information can be retrieved and area based monitoring can be performed. Positioning includes all the functionality necessary to find and work with geographic coordinates. It can use a variety of external sources of information, including GPS. This provides us with a coordinate and altitude for the device with additional features such as speed and direction. This provides the fundamental location information used in the API[18].

C++ Positioning API

`QGeoShape` class defines a geographic area. This class was introduced in Qt 5.2. and its part of Positiong API. For the sake of consistency, subclasses should describe the specific details of the associated areas in terms of `QGeoCoordinate` instances and distances in meters. It can be directly used from C++ and QML[5].

C++ Positioning Classes	
QGeoLocation	Represents basic information about a location
QGeoCoordinate	Defines a position on the surface of the Earth
QGeoCircle	Defines a circular geographic area
QGeoPath	Defines a geographic path
QGeoPolygon	Defines a geographic polygon
QGeoRectangle	Defines a rectangular geographic area
QGeoShape	Defines a geographic area

QML Positioning API

The QML position is stored in a coordinate which contains the latitude, longitude and altitude of the device. The Location contains this coordinate and adds an address, and also has a bounding box which defines the recommended viewing region when displaying the location. A `geoShape`type represents an abstract geographic area. This type is a QML representation of `QGeoShape` which is an abstract geographic area. It includes attributes and methods common to all geographic areas[6].
The `geoCircle` type is a `geoShape` that represents a circular geographic area. It is a direct representation of a `QGeoCircle` and is defined in terms of a coordinate which specifies the center of the circle and a `qreal` which specifies the radius of the circle in meters.
The `geopath` type is a `geoshape` that represents a geographic path. It is a direct representation of a `QGeoPath` and is defined in terms of a path which holds the list of geo coordinates in the path.
The `geopolygon` type is a `geoshape` that represents a geographic polygon. It is a direct representation of `QGeoPolygon` and is defined in terms of a path which holds a list of geo coordinates in the polygon. When integrating with C++, any `QGeoCircle`, `QGeoPath` and `QGeoPolygon` values passed into QML are automatically converted into the corrispectives `geoCircle`, `geoPath`, `geoPolygon`, and vice versa[6].

QML Positioning Types	
geoLocation	Represents basic information about a location
geoCoordinate	Defines a position on the surface of the Earth
geoShape	Defines a geographic area
geoCircle	Defines a circular geographic area
geoPath	Defines a geographic path
geoPolygon	Defines a geographic polygon
geoRectangle	Defines a rectangular geographic area

The Qt Location QML module depends on the Qt Positioning QML module. Therefore every QML application that imports the Qt Location QML module must always import the Qt Positioning module as well, since `MapPolygon`, `MapPolyline` and `MapCircle` type displays polygons, lines and circles on a Map, specified in terms of an ordered list of coordinates, list of cordinates and cordinates.

C++ Classes and QML Basic Types		
Positioning (C++)	**Positioning (QML)**	**Location (QML)**
QGeoShape::QGeoCircle	geoShape::geoCircle	MapCircle
QGeoShape::QGeoPath	geoShape::geoPath	MapPolyline
QGeoShape::QGeoPolygon	geoShape::geoPolygon	MapPolygon
QGeoShape::QGeoRectangle	geoShape::geoRectangle	MapRectangle

This Basic Types can find direct correspondence to the GeoJSON geometries that will be illustrated in the following chapter. To be noted here, till Qt 5.12, neither `QGeoPolygon`, `geoPolygon` nor `MapPolygon` types supported holes.

Chapter 4

Geographic JSON (GeoJSON)

GeoJSON is a format for encoding data about geographic features using JavaScript Object Notation (JSON)[]. Geographic features need not be physical things; any thing with properties that are bounded in space may be considered a feature. GeoJSON provides a means of representing both the properties and spatial extent of features[28].

4.1 JavaScript Object Notation (JSON)

According json.org[23] and JSON RFC[25]'s paper. JSON is a data interchange format derived from JavaScript, widely used as a data exchange format on the internet. JSON is easy for humans to read and write, plus it is easy for machines to parse and generate. It is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++ and C#. These properties make JSON an ideal data-interchange language. JSON objects are written in key/value pairs. Keys must be strings, and values must be a valid JSON data type. A boolean value is represented by the strings true or false in JSON; a value can have any of the types in the figure 4.1. A string can be any valid unicode string; an array is a list of values. An object is a collection of key/value pairs, all keys in an object are strings, and an object cannot contain any duplicate keys. In various languages, a collection of name/value pairs, defines an object (or record, struct, dictionary, hash table, keyed list, associative array), while

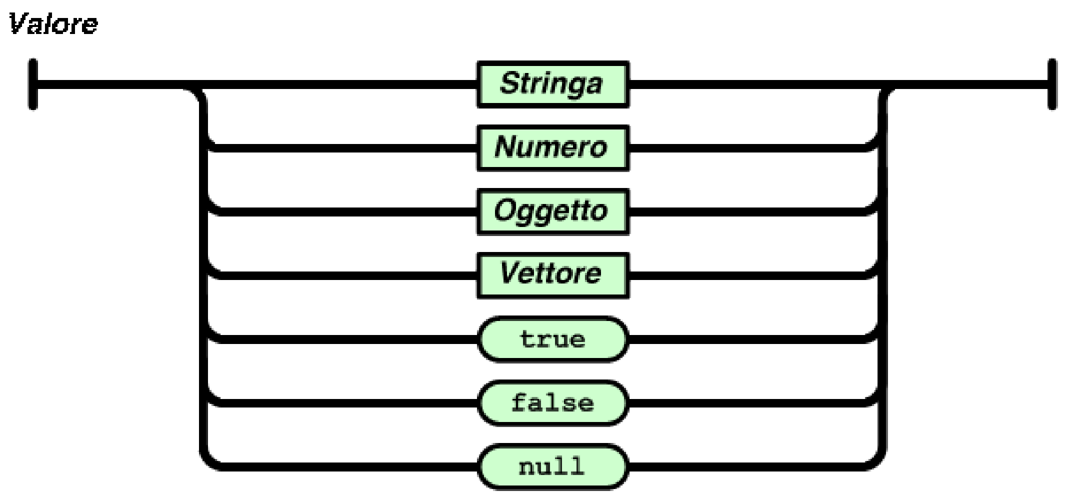


Figure 4.1: valid JSON data type

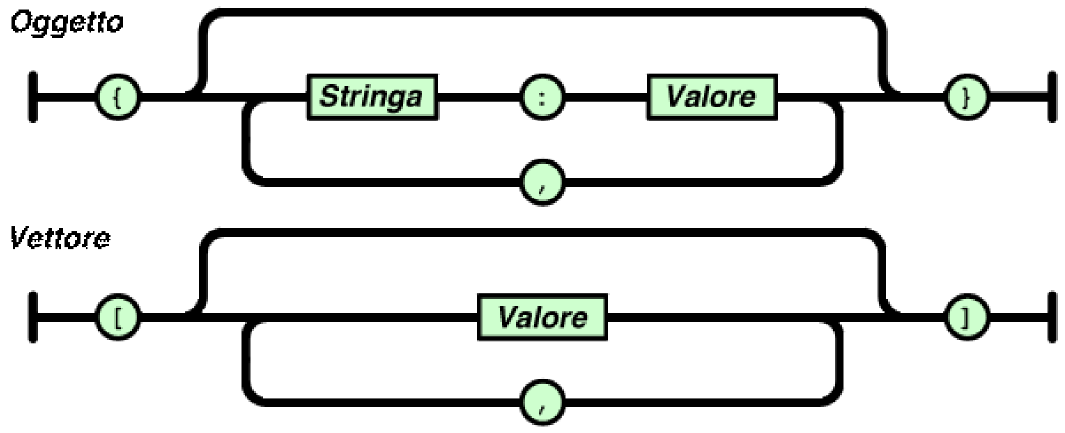


Figure 4.2: representation of JSON encloses arrays in square brackets ([...]) and objects in curly brackets ({ ... }). Entries in arrays and objects are separated by commas and separator between keys and values in an object is a colon.

an ordered list of values defines an array (or vector, list, or sequence). These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

4.2 GeoJSON

GeoJSON is a geospatial data interchange format based on JSON. The GeoJSON format specification was published in 2008 and It defines several types of JSON objects, and the manner in which they can be combined to represent geographic features, their properties, and their spatial extents[27]. GeoJSON uses the geographic coordinate reference system WGS84 and today, it plays an important and growing role in many spatial databases, web APIs, and open data platforms. Consequently the implementers increasingly demand formal standardization, improvements in the specification, guidance on extensibility, and the means to utilize larger GeoJSON datasets.

GeoJSON objects represent geographic features only, and do not specify associations between geographic features and particular devices, users, or facilities. When a GeoJSON object is used in a context where it identifies the location of a device, user, or facility, it becomes subject to the architectural, security, and privacy considerations in RFC 6280[24]. The GeoJSON Working Group worked on a format for a streamable sequence of GeoJSON texts based on RFC 7464[26] (JSON Text Sequences) to address the difficulties in serializing very large sequences of features or feature sequences of indeterminate length. Note that GeoJSON does not provide privacy or integrity services. If sensitive data requires privacy or integrity protection, those must be provided by the transport – for example, Transport Layer Security (TLS) or HTTPS. There will be cases in which stored data need protection, which is out of scope.

GeoJSON can be used to represent a geometry, or collection of geometries, a feature, or a collection of features. The geometry types supported in GeoJSON are Point, LineString, Polygon, MultiPoint, MultiLineString, MultiPolygon and GeometryCollection. Features are geometries with additional properties, and FeatureCollection are sets of features[33]. A GeoJSON text is a JSON text and consists of a single GeoJSON object. The object has a member with the name "type", the value of the member must be one of the GeoJSON types:

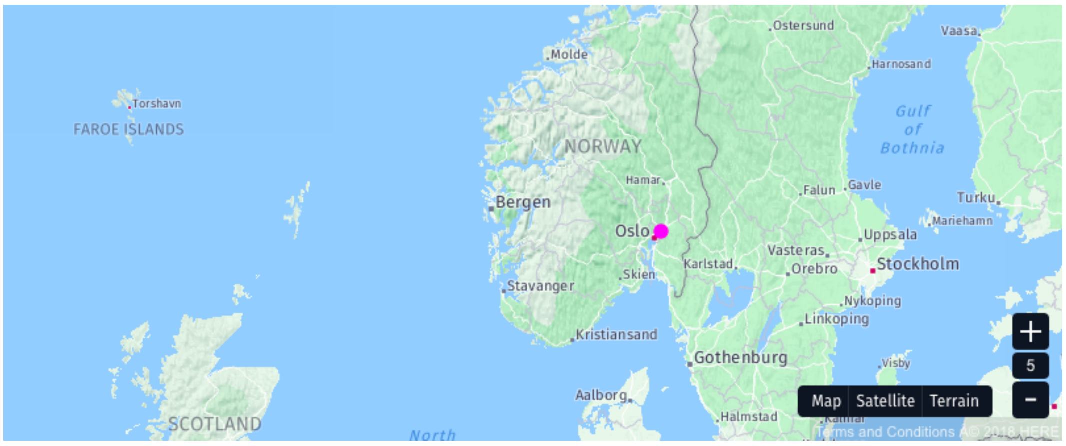
GeoJSON Types	
1	**Point**
2	**LineString**
3	**Polygon**
4	**MultiPoint**
5	**MultiLineString**
6	**MultiPolygon**
7	**GeometryCollection**
8	**Feature**
9	**FeatureCollection**

A Position is a fundamental geometric construct. It's an array of 2 or 3 numbers. The first two elements are longitude and latitude, precisely in that order and using decimal numbers. And the third (optional) number represents altitude or elevation. So, a position is basically the array [longitude, latitude, altitude]. GeoJSON Geometry object (to be noted: Feature and FeatureCollection are not geometry types) of any type other than "GeometryCollection" has a member with the name "coordinates" and value member is an array. The structure of the elements in this array is determined by the type of geometry:

- one position in the case of a "Point" geometry;
- an array of positions in the case of a "LineString" or "MultiPoint" geometry;
- an array of "LineString" coordinates in the case of a "Polygon" or "MultiLineString" geometry;
- an array of "Polygon" coordinates in the case of a "MultiPolygon" geometry.

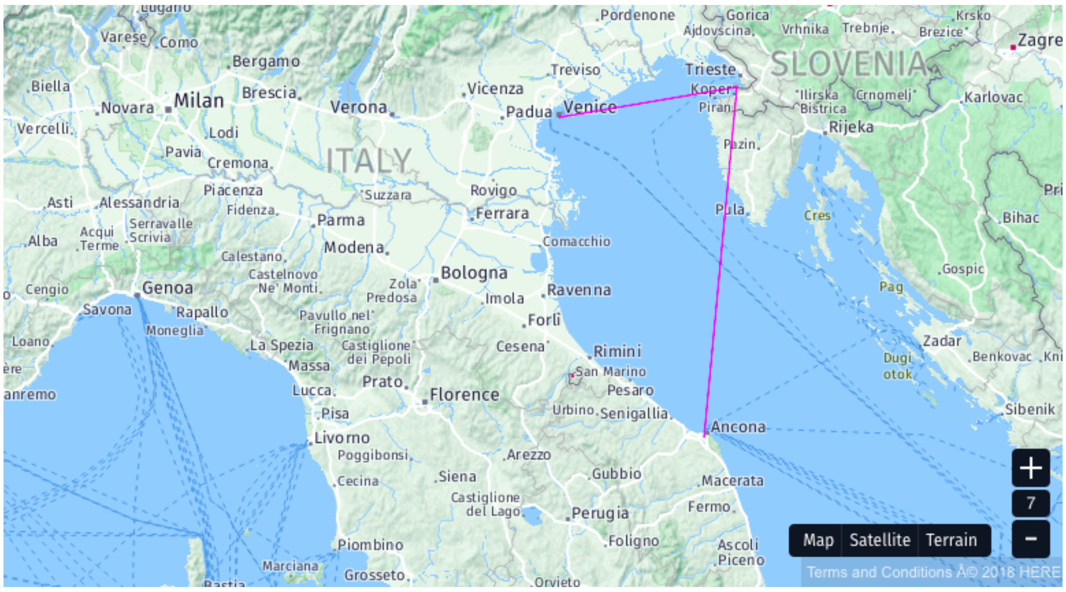
4.2.1 Single Geometry objects

Point, LineString and Polygon shapes are single type Geometry objects. And their structures are illustrated in the following examples.



```
{
    "type": "Point",
    "coordinates": [11,60]
}
```

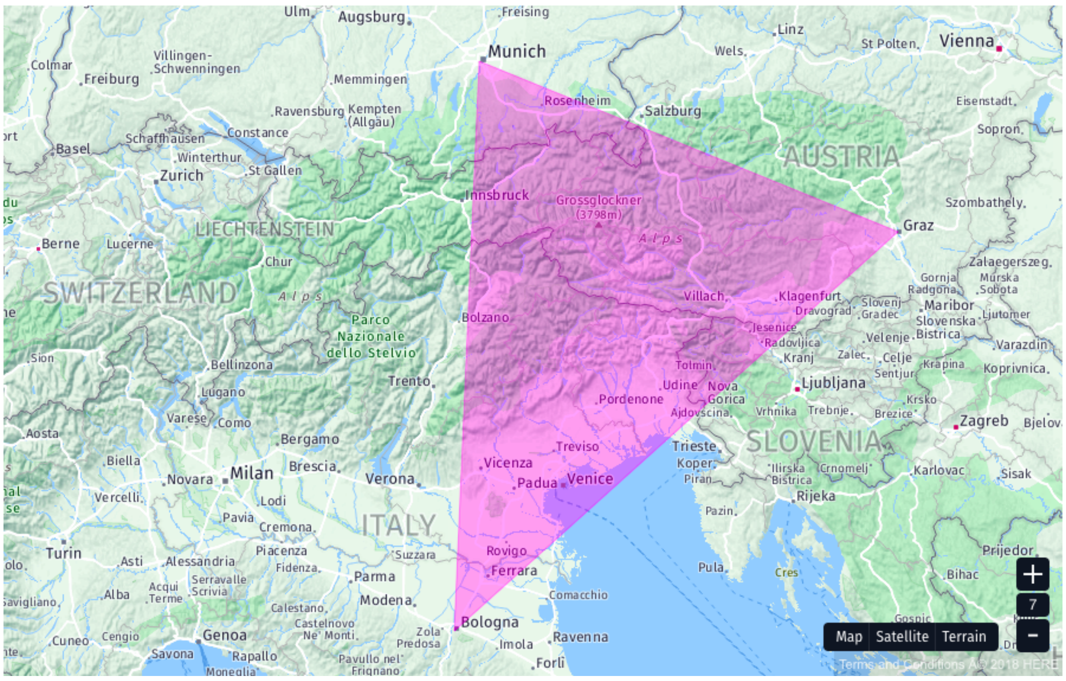
Listing 4.1: Example of a GeoJSON Point



```
{
  "type": "LineString",
  "coordinates": [
          [12.33, 45.42],
        [13.75, 45.59],
          [13.49, 43.59]
    ]
}
```

Listing 4.2: Example of a GeoJSON LineString

Polygons in GeoJSON are built around the concept of "linear ring": a linear ring is a closed LineString with four or more positions, the first and last positions contain identical values. A linear ring is the boundary of a surface or the boundary of a hole in a surface. It follows the right-hand rule with respect to the area it bounds: exterior rings are counterclockwise, and holes are clockwise. For backwards compatibility reasons to the 2008 GeoJSON standard, parsers should not reject Polygons that do not follow the right-hand rule. For type "Polygon", the "coordinates" member has to be an array of linear ring coordinate arrays. For Polygons with more than one of these rings, the first has to be intended as the exterior perimeter, and any others are interior rings. The exterior ring bounds the surface, and the interior rings bound holes within the surface.



```
{
  "type": "Polygon",
  "coordinates": [
    [
      [11.55,48.13],
      [11.32,44.48],
      [15.44,47.06],
      [11.55,48.13]
    ]
  ]
}
```

Listing 4.3: Example of a GeoJSON Polygon

```
{
    "type": "Polygon",
    "coordinates": [
        [[35, 10], [45, 45], [15, 40], [10, 20], [35, 10]],
        [[20, 30], [35, 35], [30, 20], [20, 30]]
    ]
}
```

Listing 4.4: Example of a GeoJSON Polygon with holes

4.2.2 Homogeneously multipart Geometries

MultiPoint, MultiLineString, and MultiPolygon are multipart Geometry objects. They are defined as compositions of single type geometry objects (Point, LineString and Polygon).

4.2.3 Heterogeneous composition of Geometries

A GeoJSON object with type "GeometryCollection" has a member with the name "geometries". The value of "geometries" is an array, each element of this array is a GeoJSON Geometry object. Although a GeometryCollection object has no "coordinates" member, it does have coordinates: the coordinates of all its parts belong to the collection. The "geometries" member of a GeometryCollection describes the parts of this composition. To maximize interoperability, implementations should avoid nested GeometryCollections. Furthermore, according to the GeoJSON RFC, GeometryCollections composed of a single part or a number of parts of a single type should be avoided.

4.2.4 Spatially bounded entities with properties

A Feature object represents a spatially bounded thing. Every Feature object is a GeoJSON object no matter where it occurs in a GeoJSON text. It has a "type" member with the value "Feature"[1]. A Feature object has a member with the name "geometry", the value of this member shall be either a Geometry object as defined above or, in the case that the Feature is unlocated, a JSON null value. The main difference between the heterogeneous composition is that a Feature object has a member with the name "properties" with an object as a value. It can have an identifier, this identifier should be included as a member of the Feature object with the name "id", and the value of this member is either a JSON string or number.

```
{
    "type":"Feature",
    "id":"Poly",
    "properties": {
        "text":"This␣is␣a␣Feature␣with␣a␣Polygon"
    },
    "geometry": {
        "type": "Polygon",
        "coordinates": [
            [
                [17.13, 51.11],
                [30.54, 50.42],
                [26.70, 58.36],
                [17.13, 51.11]
            ],
            [
                [23.46, 54.36],
                [20.52, 51.91],
                [26.80, 54.36],
                [23.46, 54.36]
            ]
        ]
    }
}
```

Listing 4.5: Example of a GeoJSON Feature

The last GeoJSON object type is "FeatureCollection". It is defined as a sets of features. FeatureCollection has a member "features", and its value is a JSON array. Each element of the array is a Feature object as defined above.

Chapter 5

Design of Qt Location Interoperability

Aim of the project was to make Qt Location interoperable with GeoJSON[32]. This translates into adding support for loading geometries from GeoJSON and exporting geometries to GeoJSON. The fist step to achieve this result was to verify whether Qt geometries were already suited to represent all of the GeoJSON objects and to contain all the informations these objects can carry. To define the 3 basic GeoJSON objects it has been decided to use 3 child classes of the QGeoShape class, (as seen in chapter 3), `QGeoCircle` has been used to represent the Point geometry, `QGeoPath` for the LineString and `QGeoPolygon` for Polygon geometry. QGeoRectangle has not been used since rectangles are classified as Polygons in GeoJSON.

GeoJSON Point

To define a Point GeoJSON object, the `QGeoCircle` class has been used. This class defines a circular geographic area. Since The GeoJSON Point only needs a single position, and given that Qt Location does not implements a single position geospatial class, it has been decided to use the center of a QgeoCircle as reference for the coordinates of a GeoJSON Point. Visually a point would be reppresented by a small cricle.

GeoJSON LineString

To define a LineString it has been used the `QGeoPath` class without any changes. This class already supported all of the features needed to describe a GeoJSON

LineString. The path is defined by an ordered list of `QGeoCoordinates`. Each two adjacent elements in the path are intended to be connected together by the shortest line segment of constant bearing passing through both elements.

GeoJSON Polygon

To define a Polygon the `QGeoPolygon` class has been used. This class defines a geographic polygon. The Polygon is defined by an ordered list of `QGeoCoordinates` representing its perimeter. Each two adjacent elements in this list are intended to be connected together by the shortest line segment of constant bearing passing through both elements. The `QGeoPolygon` class does not support holes, thus, the first step to achieve feature parity was to upgrade the Qt Location and Qt Positiong APIs to add holes support in polygons.

The Container Class: QVariant

Because C++ forbids unions from including types that have non-default constructors or destructors, most interesting Qt classes cannot be used in unions. For this reason it has been decided to use `QVariant` class to store the data of all GeoJSON objetcs. A `QVariant` object holds a single value of a single `type()` at a time. (Some types are multi-valued, for example a string list.) You can find out what type, T, the variant holds, convert it to a different type using convert(), get its value using one of the `toT()` functions (e.g., `toSize()`) and check whether the type can be converted to a particular type using `canConvert()`[9].

Here is some example code to demonstrate the use of `QVariant`:

```
QDataStream out(...);
QVariant v(123);            // The variant now contains an int
int x = v.toInt();          // x = 123
out << v;                   // Writes a type tag and an int to out
v = QVariant("hello");      // The variant now contains a QByteArray
v = QVariant(tr("hello"));  // The variant now contains a QString
int y = v.toInt();          // y = 0, v cannot be converted to an int
QString s = v.toString();   // s = tr("hello")  (see QObject::tr())
out << v;                   // Writes a type tag and a QString to out
...
QDataStream in(...);        // (opening the previously written stream)
in >> v;                    // Reads an Int variant
```

```
int z = v.toInt();            // z = 123
qDebug("Type␣is␣%s",          // prints "Type is int"
        v.typeName());
v = v.toInt() + 100;          // The variant now hold the value 223
v = QVariant(QStringList());
```

Qt provides the following sequential containers: `QList`, `QLinkedList`, `QVector`, `QStack`, and `QQueue`. For most applications. `QList<T>` is one of Qt's generic container classes. It stores items in a list that provides fast index-based access and index-based insertions and removals. Qt also provides these associative containers: `QMap`, `QMultiMap`, `QHash` and `QSet`. The containers conveniently support values associated with a single key. `QMap<Key, T>` is one of Qt's generic container classes. It stores (key, value) pairs and provides fast lookup of the value associated with a key. The `QMap` class is a template class that provides a red-black-tree-based dictionary[7].

It is possible to use the `QVariant` container class with both `QList` and `QMap`:

QVariantMap is a synonym for:

```
QMap<QString, QVariant>
```

and QVariantList is a synonym for:

```
QList<QVariant>.
```

The QML engine provides automatic type conversion between `QVariantList` and JavaScript arrays, and between `QVariantMap` and JavaScript objects. Without `QVariant`, this would have been a problem.

To design the new interoperability class, `QVariantMap` and `QVariantList` have been extensively used to store the geometries imported from GeoJSON data. In this way it has been possible, by nesting QVariantLists and QVariantMaps at various levels, to build arbitrarily complex data structures of arbitrary types. This approach was very powerful and versatile, and allowed to maintain very simple compatibility with both QML and C++ applications.

GeoJSON homogeneous and Heterogeneous multiple geometries

Given its simple interoperability between C++ and QML, `QVariantList` has been chosen To represent GeoJSON homogeneously typed (MultiPoint, MultiLineString, MultiPolygon) and heterogeneously typed (GeometryCollection) multiple geometries.

GeoJSON objects including geometries and properties

GeoJSON Feature and FeatureCollection Objects represents a geometry of any of the above lists types, equipped with a properties member. The best Qt data structure to represent these objects is the `QVariantMap`. A `QVariantMap` with 2 memebers allows to port any Feature object in Qt. The FeatureCollection Object has been represented with a `QVariantList` of many `QVariantMap`. In the following table it is possible to see the comparison between GeoJSON objects and Qt Location classes.

Feature-Parity Check		
GeoJSON Objects	**Qt Classes**	**Parity**
Point	QGeoCircle	Partial (1)
LineString	QGeoPath	Yes
Polygon	QGeoPolygon	No (2)
MultiPoint	QVariantList(QGeoCircle)	Yes
MultiLineString	QVariantList(QGeoPath)	Yes
MultiPolygon	QVariantList(QGeoPolygon)	Yes
GeometryCollection	QVariantList(QGeoShape)	Yes
Feature	QVariantMap	Yes
FeatureCollection	QVariantList(QVatiantMap)	Yes

(1) Need to store point coordinates in `QGeoCircle` center property.

(2) Need to implement holes support in `QGeoPolygon`.

5.1 QGeoJson class

To achieve the interoperability, a new class named `QGeoJson` was designed. The `QGeoJson` class had been used to convert GeoJSON document and a proper Qt data structures to be used both in C++ and QML. The class would have featured an importer method and an exporter method. To ease the development and debugging a third member function was designed to print in a readable format the imported data structure.

```
// This method imports a GeoJSON file to QVariantList:

static QVariantList importGeoJson(const QJsonDocument &doc);

// This method exports a GeoJSON file from a QVariantList:

static QJsonDocument exportGeoJson(const QVariantList &list);

// This method prints the content of the imported QVariantList:

static QString toString(const QVariantList &importedGeoJson);
```

After the GSoC project was completed one more member function was designed to operate the linting of a GeoJSON document.

```
 // This method performs validation on the input

static bool isValidGeoJson(const QJsonDocument &geojson,
QJsonParseError *error = nullptr);
```

5.1.1 Importing GeoJSON

The method `importGeoJson()` accepts a `QJsonDocument` from which it extracts a single JSON object, since the GeoJSON RFC expects that a valid GeoJSON Document has in its root a single JSON object. The importer returns a `QVariantList` containing a single QVariantMap. This map has always at least 2 (key, value) pairs. The first one has type as key, and the corresponding value is a string identifying the GeoJSON type. This value can be one of the GeoJSON object types: Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, GeometryCollection, FeatureCollection. The second pair has data as key, and the corresponding value can be either a `QGeoShape` or a `QVariantList`, depending on the GeoJSON type. The Feature type is converted into the type of the geometry contained within, with an additional (key, value) pair, where the key is properties and the value is a `QVariantMap`. Thus, a feature Map is distinguishable from the corresponding geometry, by looking for a properties member.

5.1.2 Exporting GeoJSON

The exporter accepts the `QVariantList` returned by the importer, and returns a JSON document. The exporter is complementary to the importer because it executes the inverse action.

5.1.3 Validation of the GeoJSON data

As JSON data is often output without line breaks to save space, it can be extremely difficult to spot errors, Qt provides support for dealing and validating JSON data (in particular it makes easy to use C++ API to parse, modify and save this data in a binary format that is directly "mapp"-able and very fast to access). Since GeoJSON is a subset of JSON and since Qt offers JSON linting, in order to ensure the correct parse of GeoJSON before importing, it has been extended the validation also to GeoJSON file. The design of `toValidGeoJson` member function, which it works as GeoJSON syntax checker and flag programming errors according to the description set out geojson.org and its RFC[27].

Chapter 6

Development of feature parity and QGeoJSON class

The process of patching and upgrading Qt Library follows a very specific path which contemplates the use of the Gerrit Codereview platform. There is a very active community around the development of the Qt libraries, Qt related tools, and add-ons, called the Qt Project. Contributors have to follow strict guidelines. The version control tool Git is integrated with Gerrit, a web based code review tool. The patches, coded following very strict guidelines[10] whose rules extend from indentation to coding style, once published on Qt code review platform, have to pass the automated check of a bot and the review of multiple Qt developers to become candidate for merging to the Qt Project code.

The GSoC Project produced 3 patches to the Qt code, which went trough the review process before being accepted and merged. Two of these patches attained the support of holes in polygon geometries, while the third patch contained the importer/exporter class.

1	**Add QGeoPolygon holes support to LocationSingleton[29]**
2	**Add hole support in QGeoPolygon[30]**
3	**Add QGeoJson: a GeoJSON parser[31]**

Table 6.1: The 3 patches merged into Qt framework

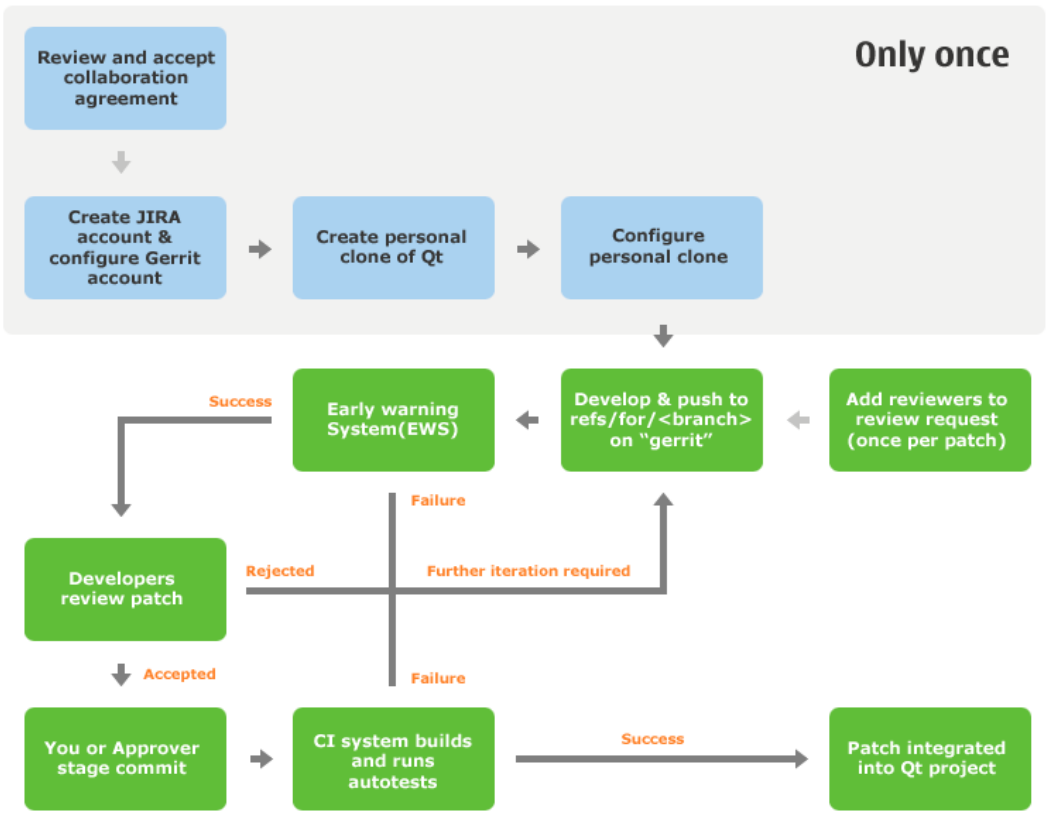

Only once
Review and accept collaboration agreement
Create JIRA account & configure Gerrit account
Create personal clone of Qt
Configure personal clone
Success
Early warning System(EWS)
Develop & push to refs/for/<branch> on "gerrit"
Add reviewers to review request (once per patch)
Failure
Developers review patch
Rejected
Further iteration required
Accepted
Failure
You or Approver stage commit
CI system builds and runs autotests
Success
Patch integrated into Qt project

Figure 6.1: The Qt open source contribution process

6.1 Adding holes support to Geographic Polygons

Like already mentioned there was only partial compatibility between the GeoJSON Polygon object and the QtLocation `QGeoPolygon` class. The Qt class was unable to offer the setting of holes inside the surface of the defined polygon. Thus, the first step to obtain feature parity, it has been to support holes in `QGeoPolygon` class. To achieve this results it has been necessary to modify the following source files in 2 different patches:

1	**qgeopath_p.h**
2	**qgeopath.cpp**
3	**qgeopolygon.h**
4	**qgeopolygon.cpp**
5	**locationsingleton.h**
6	**locationsingleton.cpp**

File 5 and 6 contain the implementation of Qt Poisitioning QML geoPolygon class. Header files with the "_p" suffix contain private class members prototypes and private properties. Implementation of C++ `QGeoPolygon` class spans on both `QGeoPath` class source files (1 and 2 on the above list) and `QGeoPolygon` class source files (3 and 4). Infact QGeoPolygon Private functions, are included in `QGeoPath` source files, while `QGeoPolygon` public APIs are included in specific class files.

Filename	Description
qgeopath_p.h	Includes private member functions prototype for both `QGeoPath` and `QGeoPolygon` classes
qgeopath.h	Includes public API member functions prototype for `QGeoPath` class. Not modified
qgeopath.cpp	Includes public API implementation for `QGeoPath` class and private implementation for both Q`GeoPath` and `QGeoPolygon` classes
qgeopolygon.h	Includes public API member functions prototype file for `QGeoPolygon` class
qgeopolygon.cpp	Includes public API implementation for `QGeoPolygon`

6.1.1 C++ Private API development

Private member functions prototypes were added to the `qgeopath_p.h` header file to add and "remove" holes, "read" a hole, and a function to "count" holes number in the polygon. An attribute was also added to store coordinates of holes perimeters.

src/positioning/qgeopath_p.h:

```
// Sets the holePath for a hole inside the polygon.
// The hole is a QVariant containing a QList<QGeoCoordinate>
void addHole(const QList<QGeoCoordinate> &holePath);

// Returns the holePath at a given index
const QList<QGeoCoordinate> holePath(int index) const;

// Removes a single hole from the polygon
void removeHole(int index);

// Returns the number of holes
int holesCount() const;

// Stores coordinates of holes perimeters
QList<QList<QGeoCoordinate>> m_holesList;
```

In the following source file are included both public and private function members for both `QGeoPath` and `QGeoPolygon` classes. Here was added the code for the private functions whose prototypes were included in the `qgeopath_p.h` header file. The `polygonContains()` function is used to render a polygon surface on a map and makes use of the clipper library[4]. It has been modiefied too to support holes.

src/positioning/qgeopath.cpp

```cpp
// Modified version of polygonContains with holes support

bool QGeoPathPrivate::polygonContains(const QGeoCoordinate &coordinate)
    const
{
    if (m_clipperDirty)
        const_cast<QGeoPathPrivate *>(this)->updateClipperPath();

    /*
        Iterates the holes list checking whether the point
        is contained inside the holes
    */

    for (const QVariant &hole : m_holesList) {

        QList<QGeoCoordinate> holePath = hole.value<QList<QGeoCoordinate>>()
    ;
        QGeoPolygon holePolygon;
        holePolygon.setPath(holePath);
        QGeoPath holeBoundary;
        holeBoundary.setPath(holePath);

        if (holePolygon.containsCoordinate(coordinate) && !(holeBoundary.
    containsCoordinate(coordinate)))
            return false;
    }
    QDoubleVector2D coord = QWebMercator::coordToMercator(coordinate);
    double tlx = QWebMercator::coordToMercator(m_bbox.topLeft()).x();

    if (coord.x() < tlx)
        coord.setX(coord.x() + 1.0);
    IntPoint intCoord = QClipperUtils::toIntPoint(coord);

    return c2t::clip2tri::pointInPolygon(intCoord, m_clipperPath) != 0;
}
```

```
/*
    Sets the path for an Hole inside the polygon.
    The hole has to be provided in QList <QGeoCoordinate> type
*/

void QGeoPathPrivate::addHole(const QVariant &holePath)
{
    m_holesList.append(holePath);
}

/*!
    Return a QVariant containing a QList<QGeoCoordinate>
    containing the hole at index, have to find a way to return the QVariant.

*/

const QVariant &QGeoPathPrivate::holePath(int index) const
{
    return  m_holesList.value(index);
}

// Removes element at position \a index from the holes QList
void QGeoPathPrivate::removeHole(int index)
{
    if (index < 0 || index >= m_holesList.size())
        return;

    m_holesList.removeAt(index);
}
```

6.1.2 C++ Public API development

The `qgeopolygon.h` header file contains the prototypes for the public API member functions. The `Q_INVOKABLE` macro applied to declarations of member functions allows them to be invoked via the meta-object system in QML code.

src/positioning/qgeopolygon.h:

```
    Q_INVOKABLE void addHole(const QVariant &holePath);
                void addHole(const QList<QGeoCoordinate> &holePath);
    Q_INVOKABLE const QVariantList hole(int index) const;
                const QList<QGeoCoordinate> holePath(int index) const;
    Q_INVOKABLE void removeHole(int index);
    Q_INVOKABLE int holesCount() const;
```

The `qgeopolygon.cpp` file includes the implementations of the public API member functions. The `addHole()` method is overloaded and accepts both `QGeoCoordinate` and `QVariant` parameter, to maximize the compatibility with QML applications. For the same reason, there are 2 different functions to "write" a new hole inside a polygon.

src/positioning/qgeopolygon.cpp:

```
/*
    Sets the path for a hole inside the polygon.
    The hole is a QVariant containing a QList<QGeoCoordinate>
*/
void QGeoPolygon::addHole(const QVariant &holePath)
{
    Q_D(QGeoPolygon);
    QList<QGeoCoordinate> qgcHolePath;
    if (holePath.canConvert<QVariantList>()) {
        const QVariantList qvlHolePath = holePath.toList();
        for (const QVariant &vertex : qvlHolePath) {
            if (vertex.canConvert<QGeoCoordinate>())
                qgcHolePath << vertex.value<QGeoCoordinate>();
        }
    }
    return d->addHole(qgcHolePath);
}
```

```
/*
    Overloaded method. Sets the path for a hole inside the polygon.
    The hole is a QList<QGeoCoordinate>
*/
void QGeoPolygon::addHole(const QList<QGeoCoordinate> &holePath)
{
    Q_D(QGeoPolygon);
    return d->addHole(holePath);
}
```

```
/*
    Returns a QVariant containing a QVariant containing a
    QList<QGeoCoordinate> which represents the hole at index
*/
const QVariantList QGeoPolygon::hole(int index) const
{
    Q_D(const QGeoPolygon);
    QVariantList holeCoordinates;
    for (const QGeoCoordinate &coords: d->holePath(index))
        holeCoordinates << QVariant::fromValue(coords);
    return holeCoordinates;
}
```

```
// Returns a QList<QGeoCoordinate> which represents the hole at \a index
const QList<QGeoCoordinate> QGeoPolygon::holePath(int index) const
{
    Q_D(const QGeoPolygon);
    return d->holePath(index);
}
```

```
// Removes element at position \a index from the holes QList.
void QGeoPolygon::removeHole(int index)
{
    Q_D(QGeoPolygon);
    return d->removeHole(index);
}

// Returns the number of holes.
int QGeoPolygon::holesCount() const
{
    Q_D(const QGeoPolygon);
    return d->holesCount();
}
```

6.1.3 QML API Development

The `locationsingleton.h` file contains the prototypes for the member functions needed to extend the C++ `QGeoPolygon` class to QML.

src/imports/positioning/locationsingleton.h:

```
    Q_INVOKABLE QGeoPolygon polygon(const QVariantList &perimeter, const
    QVariantList &holes) const;
```

The `locationsingleton.cpp` file contains the function implementation for methods to be used in QML.

src/imports/positioning/locationsingleton.cpp:

```
// Constructs a polygon from coordinates for perimeter and inner holes

QGeoPolygon LocationSingleton::polygon(const QVariantList &perimeter, const
     QVariantList &holes) const
{
    QList<QGeoCoordinate> internalCoordinates;
    for (int i = 0; i < perimeter.size(); i++) {
        if (perimeter.at(i).canConvert<QGeoCoordinate>())
            internalCoordinates << perimeter.at(i).value<QGeoCoordinate>();
    }
    QGeoPolygon poly(internalCoordinates);

    for (int i = 0; i < holes.size(); i++) {
        if (holes.at(i).type() == QVariant::List) {
            QList<QGeoCoordinate> hole;
            const QVariantList &holeData = holes.at(i).toList();

            for (int j = 0; j < holeData.size(); j++) {
                if (holeData.at(j).canConvert<QGeoCoordinate>())
                    hole << holeData.at(j).value<QGeoCoordinate>();
            }

            if (hole.size())
                poly.addHole(hole);
        }
    }
    return poly;
}
```

6.2 QGeoJson class

Once achieved the full feature parity between Qt Location and GeoJSON, it has been possible to move forward to the development of the importer and the exporter. Both of them have been created as methods of the new `QGeoJson` class. The QGeoJson class can be used to convert between a GeoJSON document and a data structure built as a `QVariantList` of `QVariantMaps`. The elements in this list are ready to be used as model in a `MapItemView`.

The following list includes all the methods developed in `QGeoJson` class. The class has no private API.

src/location/labs/qgeojson.h:

```
// This method importe a GeoJSON file to QVariantList:

static QVariantList importGeoJson(const QJsonDocument &doc);

// This method exporte a GeoJSON file from a QVariantList:

static QJsonDocument exportGeoJson(const QVariantList &list);

// This method prints the content of the imported QVariantList:

static QString toString(const QVariantList &importedGeoJson);
```

After the GSoC project was completed one more member function was designed to operate the linting of a GeoJSON document.

```
 // This method performs validation on the input:

static bool isValidGeoJson(const QJsonDocument &geojson,
QJsonParseError *err = nullptr);
```

6.2.1 Importing GeoJSON

The `importGeoJson()` method accepts a `QJsonDocument` from which it extracts a single JSON object, since the GeoJSON RFC expects that a valid GeoJSON Document has in its root a single JSON object. This method doesn't perform any validation on the input. The importer returns a `QVariantList` containing a single `QVariantMap`. This map has always at least 2 (key, value) pairs. The first one has type as key, and the corresponding value is a string identifying the GeoJSON type. This value can be one of the GeoJSON object types: Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, GeometryCollection, FeatureCollection. The second pair has data as key, and the corresponding value can be either a `QGeoShape` or a list, depending on the GeoJSON type. The next section provides details about this node. The Feature type is converted into the type of the geometry contained within, with an additional (key, value) pair, where the key is properties and the value is a `QVariantMap`. Thus, a feature Map is distinguishable from the corresponding geometry, by looking for a properties member.

Structure of the "data" node

For the single type geometry objects (Point, LineString, and Polygon), the value corresponding to the data key is a `QGeoShape`. When the type is Point, the data is a `QGeoCircle` with the point coordinates stored in the center property.

For example, the following GeoJSON document contains a Point geometry:

```
{
    "type" : "Point",
    "data" : [50.0, 11.0]
}
```

Listing 6.1: GeoJSON Point Geometry

It is converted to a `QVariantMap` with the following structure:

```
{
    type : Point
    data : QGeoCircle({50.000, 11.000}, -1)
}
```

Listing 6.2: Example

When the type is LineString the data is a `QGeoPath`. For example, the following GeoJSON document contains a LineString geometry:

```
{
    "type" : "LineString",
    "coordinates" : [[13.5, 43],[10.73, 48.92]]
}
```

Listing 6.3: Example of a GeoJSON LineString Geometry

It is converted to a `QVariantMap` with the following structure:

```
{


    type : LineString,
    data : QGeoPath([{43.000, 13.500}, {48.920, 10.730}])


}
```

When the type is Polygon, the data is a `QGeoPolygon` (holes are supported). For example, the following GeoJSON document contains a Polygon geometry:

```
{
    "type" : "Polygon",
    "coordinates" : [
        [[17.13, 41.11],
        [10.54, 40.42],
        [16.70, 48.36],
        [17.13, 41.11]]
    ],
}
```

Listing 6.4: GeoJSON Polygon Geometry

It is converted to a `QVariantMap` with the following structure:

```
{
    type : Polygon
    data : QGeoPolygon([{41.110, 17.130}, {40.420,10.540}, {48.360, 16.700},
     {41.110, 17.130}])
}
```

For the homogeneously typed multipart geometry objects (MultiPoint, MultiLineString, MultiPolygon) the value corresponding to the data key is a `QVariantList`. Each element of the list is a `QVariantMap` of one of the above listed types. The elements in this list will be all of the same GeoJSON type: For example, When the type is MultiPoint, the data is a List of Points.

The following GeoJSON document contains a MultiPoint geometry:

```
{
    "type" : "MultiPoint",
    "coordinates" : [
        [11,50],
        [5.5,40.3],
        [5.7,48.90]
    ]
}
```

Listing 6.5: GeoJSON MultiPoint Geometry

It is converted to a `QVariantMap` with the following structure:

```
{
    type : MultiPoint
    data : [
        {
        type : Point
        data : QGeoCircle({50.000, 11.000}, -1)
        },
        {
        type : Point
        data : QGeoCircle({40.300, 5.500}, -1)
        },
        {
        type : Point
        data : QGeoCircle({48.900, 5.700}, -1)
        }
    ]
}
```

The MultiLineString and MultiPolygon cases follow the same structure, where the data member is respectively a list of LineStrings or a list of Polygons.

The GeometryCollection is a heterogeneous composition of other geometry types. In the resulting `QVariantMap`, the value of the data member is a `QVariantList` populated by `QVariantMap` of various geometries, including the GeometryCollection itself.

```
{
    "type" : "GeometryCollection",
    "geometries" : [
        {
            "type" : "MultiPoint",
            "coordinates" : [
                [11,60], [5.5,60.3], [5.7,58.90]
            ]
        },
        {
            "type" : "MultiLineString",
            "coordinates": [
              [[13.5, 43], [10.73, 59.92]],
              [[9.15, 45], [-3.15, 58.90]]
            ]
        },
        {
            "type" : "MultiPolygon",
            "coordinates" : [
                  [
                    [
                      [19.84, 41.33],
                      [30.45, 39.26],
                      [17.07, 30.10],
                      [19.84, 41.33]
                    ]
                ]
            ]
        }
    ]
}
```

Listing 6.6: GeoJSON GeometryCollection

It is converted to a `QVariantMap` with the following structure:

```
{
  type : GeometryCollection
  data : [
    {
      type : MultiPolygon
      data : [
        {
          type : Polygon
          data : QGeoPolygon({41.330, 19.840, -1}, {39.260, 30.450, -1},
    {30.100, 17.070, -1}, {41.330, 19.840, -1}, )
        }
      ]
    }
    {
      type : MultiLineString
      data : [
        {
          type : LineString
          data : QGeoPath({45.000, 9.150, -1}, {58.900, -3.150, -1}, )
        }
        {
          type : LineString
          data : QGeoPath({43.000, 13.500, -1}, {59.920, 10.730, -1}, )
        }
      ]
    }
    {
      type : MultiPoint
      data : [
        {
          type : Point
          data : QGeoCircle({58.900, 5.700, -1}, 20000)
        }
        {
          type : Point
          data : QGeoCircle({60.300, 5.500, -1}, 20000)
        }
        {
          type : Point
          data : QGeoCircle({60.000, 11.000, -1}, 20000)
        }
      ]
    }
  ]
}
```

The Feature object, which consists of one of the previous geometries together with related attributes, is structured like one of the 7 above mentioned geometry types, plus a properties member. The value of this member is a `QVariantMap`. The only way to distinguish a Feature from the included geometry is to check if a properties node is present in the `QVariantMap`.

For example, the following Feature:

```
{
    "type" : "Feature",
    "id" : "Poly",
    "properties" : {
        "name" : "Poly",
        "text" : "This␣is␣a␣Feature␣with␣a␣Polygon",
        "color" : "limegreen"
    },
    "geometry" : {
        "type" : "Polygon",
        "coordinates" : [
            [
                [11.55, 48.13],
                [11.32, 44.48],
                [15.44, 47.06],
                [11.55, 48.13]
            ],
            [
                [12.46, 47.36],
                [12.52, 46.19],
                [13.25, 47.10],
                [12.46, 47.36]
            ],
            [
                [13.75, 46.99],
                [12.76, 46.13],
                [14.25, 46.51],
                [13.75, 46.99]
            ]
        ]
    }
}
```

Listing 6.7: GeoJSON Feature Object

```
{
  type : Polygon
  data : QGeoPolygon({48.130, 11.550, -1}, {44.480, 11.320, -1}, {47.060,
    15.440, -1}, {48.130, 11.550, -1}, )
  properties : {
    color : limegreen
    name : Poly
    text : This is a Feature with a Polygon
  }
}
```

The FeatureCollection is a composition of Feature objects. The value of the data member in a FeatureCollection is a `QVariantList` populated by Feature. For example, the following FeatureCollection:

```
{
    "type" : "FeatureCollection",
    "properties" : {
        "color" : "crimson"
    },
    "features" : [
        {
            "type" : "Feature",
            "id" : "Poly",
            "properties" : {
                "text" : "This␣is␣a␣Feature␣with␣a␣Polygon"
            },
            "geometry" : {
               "type" : "Polygon",
               "coordinates" : [
                   [
                       [9.13, 41.11],
                       [20.54, 40.42],
                       [16.70, 48.36],
                       [9.13, 41.11]
                   ],
                   [
                       [13.46, 41.36],
                       [10.52, 41.91],
                       [16.80, 43.36],
                       [13.46, 41.36]
                   ]
               ]
           }
       },
       {
           "type" : "Feature",
           "id" : "MultiLine",
           "properties" : {
               "text" : "This␣is␣a␣Feature␣with␣a␣MultiLineString",
               "color" : "deepskyblue"
           },
           "geometry" : {
              "type" : "MultiLineString",
              "coordinates" : [
                [[2.5, 43], [10.73, 39.92]],
                [[9.15, 45], [-3.15, 45.90]]
              ]
           }
       }
    ]
}
```

Listing 6.8: GeoJSON FeatureCollection Object

It is converted to a `QVariantMap` with the following structure:

```
{
  type : FeatureCollection
  data : [
    {
      type : MultiLineString
      data : [
        {
          type : LineString
          data : QGeoPath({45.000, 9.150, -1}, {45.900, -3.150, -1}, )
        }
        {
          type : LineString
          data : QGeoPath({43.000, 2.500, -1}, {39.920, 10.730, -1}, )
        }
      ]
      properties : {
        color : deepskyblue
        text : This is a Feature with a MultiLineString
      }
    }
    {
      type : Polygon
      data : QGeoPolygon({41.110, 9.130, -1}, {40.420, 20.540, -1}, {48.360,
     16.700, -1}, {41.110, 9.130, -1}, )
      properties : {
        text : This is a Feature with a Polygon
      }
    }
  ]
}
```

6.2.2 Exporting GeoJSON

The exporter accepts the `QVariantList` returned by the importer, and returns a JSON document. The exporter is complementary to the importer because it executes the inverse action.

6.2.3 Linting GeoJSON

This method performs validation on the input.

```
// Method to validate GeoJSON file

/*
 * The QGeoJsonParseError class is used to report errors during GeoJSON
    parsing.
 * WARNING! This member function is part of Qt, thus not stable API, it is
    part
 * of the experimental components of Qt Location.
 * Until it is promoted to public API, it may be subject to
 * source and binary-breaking changes.
 *
*/

bool isValidGeoJson(const QJsonDocument &geojson, QJsonParseError *error =
    nullptr)
{
    // Load the JSON file using Qt's API
    QJsonParseError err;
    QJsonDocument loadDoc(QJsonDocument::fromJson(loadFile.readAll(), &err))
    ;
    if (err.error) {
         qWarning() << "[1]:␣Error␣parsing␣the␣JSON␣document:␣" << err.
    errorString();
         return false;
    }
```

```
    // Checking whether the GeoJSON has only a JSON object
    if (!loadDoc.isObject()) {
        qWarning() << "[2]:␣Error␣parsing␣not␣a␣GeoJSON␣object␣";
        return false;
    }
    // Extract the JSON object
    QJsonObject loadObject = loadDoc.object();
    // Extract the map using Qt's API
    QVariantMap root = loadObject.toVariantMap();

    // Checking whether the JSON object has a "type" member
    QVariant keyMap = root.value(QStringLiteral("type"));
    if (keyMap == QVariant::Invalid) {
        qWarning() << "[3]:␣Error␣parsing␣\"type\"␣check␣failed␣";
        return false;
    }
    QString valueMap = keyMap.value<QString>();
```

```
// This is an array of string with all GeoJSON objects (RFC)
QString geojsonObject[] = {
    QStringLiteral("Point"),
    QStringLiteral("MultiPoint"),
    QStringLiteral("LineString"),
    QStringLiteral("MultiLineString"),
    QStringLiteral("Polygon"),
    QStringLiteral("MultiPolygon"),
    QStringLiteral("GeometryCollection"),
    QStringLiteral("Feature"),
    QStringLiteral("FeatureCollection")
};
```

```
// Checking whether the "type" member has a GeoJSON admitted value
for (int i = 0; i < geoObject.size(); ++i) {
    if (valueMap == geoObject[i]) {
        break;
    } else if (i == geoObject.size()) {
    qWarning() << "[4]:␣Error␣parsing␣invalid␣value␣\"type\"␣";
    return false;
    }
}
// Checking if the GeoJSON geometries has a "coordinates" member
QVariant keyCrd = root.value(QStringLiteral("coordinate");
for (int i = 0; i < geoObject[5]; ++i) {
    if (keyCrd == QVariant::Invalid) {
        qWarning() << "[5]:␣Error␣parsing␣no␣coordinates␣key␣to␣
Geoometry␣Objects␣";
        return false;
    }
}
```

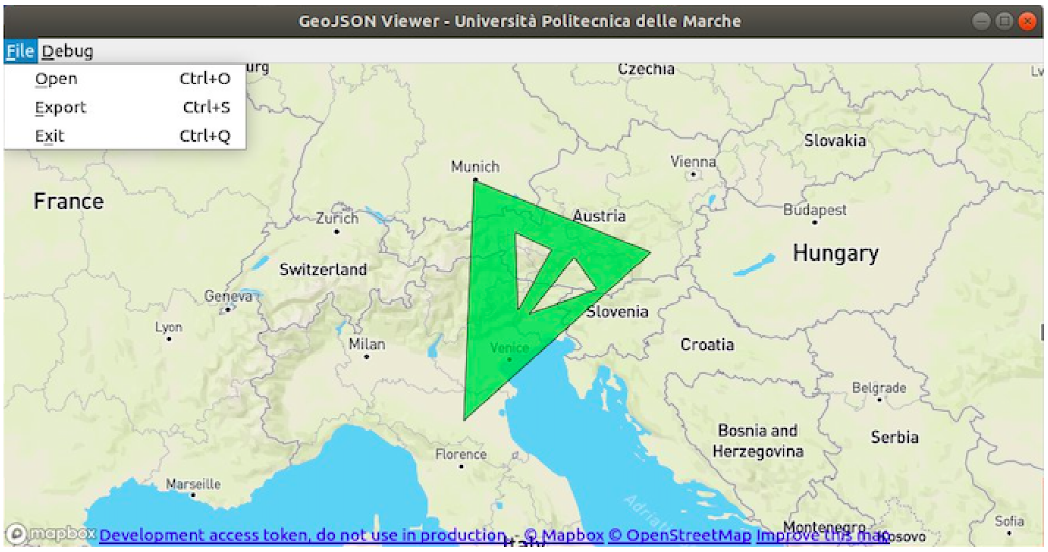
6.2.4 Debug tool for QGeoJson class

To make debug easier, the member function toString has been developed. It accepts the `QVariantList` structured like described in section Importing GeoJSON, and returns a string containing the same data in a readable form. The toString outputs, for debugging purposes, the content of a `QVariantList` structured like `importGeoJson()` does, to a `QString` using a prettyfied format.

6.3 Test and Example

Together with the class, a very simple test and example app has been developed and published,[31] with the goal of illustrating how to apply the model/view paradigm using `QGeoJson` class in QML.

The app can import a GeoJSON file using the "open" menu entry, and displays the geometries on a map using a `Delegate` and a `MapItemView`[16]. After opening a file the app can export it to a new GeoJSON file extracting the geometries from the view. A Debug Menu allows to print in a readable format the imported data structure, using the `QGeoJson::toString()` method.



The example features a ready to use delegate, which is meant to operate recursively on a `QVariantList`. This is why the importer returns a `QVariantList` with a single element. When the element is a single geometry, the delegate passes it to the `MapItemVIew` to be displayed, when it is a multiple geometry, acts recursively on the data member of the `QVariantMap`.

The app also features a C++ function to extract the geometries from the QML `MapItemView`. This extractor is used in combination with the exporter member function of `QGeoJson::exportGeojson()`.

In the patch are also included the files to run the proper autotests[19].

6.3.1 Online resources about the project

Add hole support in QGeoPolygon:

Add QGeoPolygon holes support to LocationSingleton:

Add QGeoJson: a GeoJSON parser:

GitHub:

Chapter 7

Conclusion

Research in the field of geographic notation is, without any doubt, one of the most promising sector of Information Technology in terms of possible applications. These applications can span from improving the daily life to military use, from Artificial Intelligence to environmental studies. Like often happens in the Information Technology, most likely the best perspectives have yet to be thought. It becomes then clear the importance of having flexible yet scalable and complete standards, and powerful tools to work with this standards. GeoJSON and the open source Qt framework is a good example of a potential successful synergy. Without any doubt there is still wide margins for improving the tools available to developers, to work with Qt and GeoJSON data, yet in the design and development described in this work, together and before the most obvious objectives, there was a single perspective point: enabling developers to realize their "Geographic" projects in the most efficient and straightforward way. It happens very often that a good idea is crippled by difficulties in implementation. This is why I thought that putting together an open standard like GeoJSON with a powerful Open Source framework I could have paved the way for a number of ideas to get realized, and maybe, for one of them, to be decisive for a step towards a better world.

Bibliography

[1] S. Chamberlain. GeoJSON Specification. Available at https://cran.r-project.org/web/packages/geojsonio/vignettes/geojson_spec.html.

[2] T. Q. Company. About Qt. Available at https://www.qt.io/company.

[3] T. Q. Company. Adding Compilers. Available at https://doc.qt.io/qtcreator/creator-tool-chains.html.

[4] T. Q. Company. Clipper Polygon Clipping Library. Available at https://doc.qt.io/qt-5/qtlocation-attribution-clipper.html.

[5] T. Q. Company. Geographic shape class (C++). https://doc.qt.io/qt-5/qgeoshape.html.

[6] T. Q. Company. Geographic shape class (QML). https://doc.qt.io/qt-5/qml-geoshape.html.

[7] T. Q. Company. Introduction to Qt's container classes. Available at https://doc.qt.io/qt-5/containers.html.

[8] T. Q. Company. Properties. Available at https://doc.qt.io/qt-5.12/properties.html.

[9] T. Q. Company. Qt Container class. Available at https://doc.qt.io/qt-5/qvariant.html.

[10] T. Q. Company. Qt Contribution Guidelines. Available at https://wiki.qt.io/Qt_Contribution_Guidelines.

[11] T. Q. Company. Qt Essentials Modules. Available at https://doc.qt.io/qt-5/qtmodules.html.

[12] T. Q. Company. Qt Framework. Available at https://www.qt.io/qt-frameworkhttps://wiki.qt.io/Qt{_}for{_}Beginners.

[13] T. Q. Company. Qt Location API. Available at https://doc.qt.io/qt-5/qtlocation-index.html.

[14] T. Q. Company. Qt Location C++ Documentation. Available at https://doc.qt.io/qt-5/location-places-cpp.html.

[15] T. Q. Company. Qt Location MapItemView. Available at https://wiki.qt.io/Qt_History.

[16] T. Q. Company. Qt Location MapItemView. Available at https://doc.qt.io/qt-5/qml-qtlocation-mapitemview.html.

[17] T. Q. Company. Qt Location QML Documentation. Available at https://doc.qt.io/qt-5/qtlocation-places-example.html.

[18] T. Q. Company. Qt Posistiong API. Available at https://doc.qt.io/qt-5/qtpositioning-index.html.

[19] T. Q. Company. Running Autotests. Available at https://doc.qt.io/qtcreator/creator-autotest.html.

[20] T. Q. Company. The Qt Company. Available at https://www.qt.io/company.

[21] T. Q. Company. How does Google Summer of Code it works?, 2018. Available at https://wiki.qt.io/Google_Summer_of_Code/Processes.

[22] Google. History of Google Summer of Code. 2005. Available at https://google.github.io/gsocguides/student/history-of-gsoc.

[23] Internet Engineering Task Force (IETF). Available at https://www.json.org/json-en.html.

[24] Internet Engineering Task Force (IETF). An Architecture for Location and Location Privacy in Internet Applications. Available at `https://tools.ietf.org/html/rfc6280`.

[25] Internet Engineering Task Force (IETF). JavaScript Object Notation. Available at `url{https://www.json.org/json-en.html}`.

[26] Internet Engineering Task Force (IETF). Rfc7464: Javascript object notation (json) text sequences, 2015. Available at `https://tools.ietf.org/html/rfc7464`.

[27] Internet Engineering Task Force (IETF). RFC 7946: The GeoJSON Format, 2016. Available at `https://tools.ietf.org/html/rfc7946`.

[28] Internet Engineering Task Force (IETF). The GeoJSON Working Group, 2016. Available at `https://datatracker.ietf.org/wg/geojson/charter`.

[29] J. Sherollari. Add QGeoPolygon holes support to LocationSingleton. https://codereview.qt-project.org/c/qt/qtlocation/+/237321.

[30] J. Sherollari. Add hole support in QGeoPolygon, 2018. Available at `https://codereview.qt-project.org/c/qt/qtlocation/+/232285`.

[31] J. Sherollari. Add QGeoJson: a GeoJSON parser, 2018. Available at `https://codereview.qt-project.org/c/qt/qtlocation/+/236253`.

[32] J. Sherollari. Google Summer of Code 2018, 2018. Available at `https://summerofcode.withgoogle.com/archive/2018/projects/5180918117433344/`.

[33] Tim Schaub, Allan Doyle, Martin Daly, Sean Gillies and Andrew Turner. GeoJSON definitions and examples. Available at `http://wiki.geojson.org/GeoJSON_draft_version_6`.

[34] Wikipedia.org. Smalltalk Object-Oriented Programming Language. Available at `https://en.wikipedia.org/wiki/Smalltalk`.