Location-based services with MongoDB

Final Report

Daniel Huntley: c1544004

Supervisor: Alia I Abdelmoty
Moderator: Chris B Jones
Module: CM3203
CM3203 One Semester Individual Project 40 Credits

Abstract

Geospatial data on the web is a core component of many of the services businesses offer including maps, geo-tagging and the storing of locations, making it an advantageous tool for developers to adopt and use. Being able to create these queries requires an in-depth knowledge of the database program which can be overwhelming and confusing to new developers. Additionally, no solutions currently exist which offer intuitive query creation for all the different MongoDB operators; and the subset of those supporting query creation are limited in functionality and supporting user documentation.

In this project, I have produced a program that aids developers in the creation and understanding of MongoDB geospatial queries. It allows users to connect and visualise a representation of their geospatial data on a map and through intuitive interaction, helping improve their understanding of the different operators in MongoDB.

Acknowledgements

I would like to thank my supervisor Alia I Abdelmoty for her continued support and guidance throughout the project, as well as those who invested time to participate in user testing and provide valuable insights into the value of the application.

Contents

1 Introduction . . . 7
- 1.1 Project Overview . . . 7
- 1.2 Aims and Objectives . . . 8

2 Background . . . 9
- 2.1 Context . . . 9
- 2.2 Geospatial Data . . . 9
- 2.3 MongoDB . . . 11
- 2.4 Existing partial solutions . . . 16
- 2.5 Likely stakeholders in the application . . . 20
- 2.6 Tools and Software Used . . . 21
- 2.7 Alternative Tools . . . 22

3 Specification and Design . . . 24
- 3.1 Functional Requirements . . . 24
- 3.2 Non Functional Requirements . . . 26
- 3.3 User Interface Design . . . 28
- 3.4 User Interface Design Choices . . . 29
- 3.5 Workflow . . . 33
- 3.6 Heuristic Evaluation . . . 39
- 3.7 High Level Architecture . . . 43
- 3.8 Class Diagram . . . 44
- 3.9 Node.JS Rest API best practices . . . 47

4 Implementation . . . 49
- 4.1 Project Infrastructure . . . 49
- 4.2 Backend . . . 49

4.3 Frontend 53
4.4 Challenges faced 56
5 Testing, results and evaluation 59
5.1 Requirements Evaluation 59
5.2 Test Cases 63
5.3 User-Testing 64
6 Future Work 68
7 Conclusion 70
8 Reflection on Learning 71
9 Appendices 74
9.1 Test Cases 74
9.2 User interface Test Cases 85

List of Figures

1 Yelp Business Sample MongoDB Document 11
2 Compass MongoDB instance connectivity 17
3 Compass MongoDB instance connectivity 18
4 GeoJSON.io Interface 19
5 Deck.gl Interface 22
6 Iterative design of filtering panel 28
7 Filtering Panel: Autocomplete Search 29
8 Notification Icon 29
9 Progress Tracker indicator 30
10 Single item selection list 30
11 Authentication Dropdown List 31
12 Query Panel 32
13 Database Connection Successful 32
14 Connecting and visualisation of MongoDB geospatial data workflow . . . 34
16 Filtering to view a subset of the geospatial data workflow 37
17 Generation of a geospatial query workflow 38
18 High Level System Architecture 43
19 Class Diagram 44
20 Class Diagram: Frontend 45
21 Class Diagram: Backend 46
22 Map API function 50
23 MongoDB establish connection function 51
24 Tiled Map Example [1] 53
25 Query Window Functionality 54

26 Drawing bounding box example for the $geoWithin operator 55
27 D3 versus Leaflet.js Comparison 57
28 Node.js MongoDB Regex Error 58
29 SUS Results Summary 66

1 Introduction

1.1 Project Overview

Location-based data drives many of the tools and applications on the web; these include maps, geo-tagging and the storing of location. These services help gauge new insights and improve end-user experience.

One of the most popular NoSQL database programs supporting the storing and querying of geospatial data on the web is MongoDB; making it an advantageous tool for developers. However, creating and querying geospatial data can be confusing and requires previous knowledge of the database program together with an understanding of the stored geospatial data. For existing tools such as Compass or MongoDB terminal interaction, there is little or no support in creating geospatial queries or understanding the stored location-based data.

The application I have developed for this project aims to address this and support new developers in querying there MongoDB geospatial collections using a map interface. Users data is visualised and plotted on a map enabling them to observe how their data is connected and provides an intuitive way to generate geospatial queries with little understanding of MongoDB and geospatial operators. In addition to further improve the end users understanding an explanation of the query is displayed explaining each component.

For the development of this project, the agile development methodology has been utilised to employ an iterative approach to design, development and testing. Doing so has enabled prioritisation of valuable features including the user interfaces usability; which has undergone multiple iterations of enhancements based on feedback of the current designs and collected results testing conducted.

The scope of this application is limited to the four MongoDB geospatial operators and is under the assumption that developers have access to a MongoDB instance containing geospatial records that can be queried.

1.2 Aims and Objectives

Aims

The aim of this project is to design and develop a web-based application that allows users to intuitively build and test MongoDB geospatial queries for their connected MongoDB datasets.

To generate queries users will interact with a map interface, visualising their collection allowing for intuitive query creation. Additionally, an information side panel will clearly explain the different MongoDB operators, as well as output query results.

Objectives

Study and research different tools and techniques used to visualise, interact and understand geospatial data. Including research into,

- Different languages and libraries best suited to develop the application
  - Languages: NodeJs and Python.
  - Libraries: Leaflet.js, D3 and Mapbox.
- Existing papers and work relating to this topic. Including,
  - geojson.io.
- Datasets which can be used for testing, including the,
  - Yelp open Dataset.
  - MongoDB sample Geojson datasets.
  - Ordnance Survey datasets inc. Points of interest and Open Roads
  - GeoJson available through Data.gov
- User-interface standards and best practice.
- Techniques which can be used to visualise large amounts of Geospatial data.

Design and develop a solution for new developers to create and understand geospatial queries.

- Provide a web-based interface allowing users to connect and visualise there MongoDB geospatial data.
- Allow users to interact with a map interface to generate the queries and display the results.
- Explain each component of the query generated using the map, supporting the users understanding of geospatial queries.

Test and evaluate the usability of the system

- Test the application using different datasets sourced during research, e.g.the Yelp Open Dataset.
- Evaluate the usability of the application, testing with a sample of real novice developers.

2 Background

2.1 Context

This chapter contains a detailed explanation of the background topics needed to understand this report and emphasises the aims of this project. This includes introducing important concepts for representing location-based data on the web (2.2), and how the database program MongoDB is well suited for storing this (2.3). After which the section evaluates existing solutions and there limited support in "aiding new developers create and understand MongoDB geospatial queries"; as well as an insight into the core tools and software used to produce the completed application.

2.2 Geospatial Data

Geospatial data is information that has a geographic component to it and can be linked to some location on the earth. It is the foundation of many of the services offered by applications on the web and for this project is used to allow developers to connect large sets of geospatial data so that it can be visualised and utilised.

The terms spatial and geospatial data are often used interchangeably. However spatial data refers to the relation of a space, where geospatial refers to the relation of a geographical location.

Expanding on this to reference geospatial data to a location on the earth, most systems use geographic coordinates of longitude and latitude to uniquely identify a specific point on the earth. A single pair of coordinates (longitude and latitude) are good for representing a single location or point on the earth. However, for regions larger than a single point such as roads, buildings etc. a series of geospatial points will be stored in succession to create these shapes.

With the vast amount of large open source geospatial datasets available including Yelp, Ordnance Survey and data.gov datasets; it has allowed us to improve our understanding, analysis and visualise the world around us today. Furthermore, the availability of these large geospatial datasets which encompass tens of thousands of geospatial objects has improved our understanding of location-based data through the visualisation and allow us to locate patterns and trends in the data.

Geospatial shapes covered in sections 2.3 and 6
Point: Defined by X, Y coordinate pair Example: type: "Point", coordinates: [40, 5]
LineString: Defined by a sequence of points with linear interpolation between the points (x1, y1,...xn, yn) Example: type: "LineString", coordinates: [[40, 5], [41, 6]]
Polygon: Defined by a closed list of points Example: type: "Polygon", coordinates: [[[0 , 0] , [3 , 6] , [6 , 1] , [0 , 0]]]

2.3 MongoDB

MongoDB is a document-oriented database program which stores semi-structured JSON like data, allowing for a flexible storage solution. In addition, MongoDB uses the BASE consistency model [2] which in brief allows for high availability of data but at the cost of potential data inconsistency, which could result in queried data being out of sync. However, these characteristics of MongoDB as well as support for horizontal scaling of data across machines "address the performance and scalability requirements of web-based applications which cannot be addressed by traditional relational databases", making it popular NoSQL database for storing real-time and big data.

Additionally, with the increasing use of location-based data in applications, a common use case for MongoDB is the storing and querying of geospatial data. This supports the storing of geospatial data in GeoJSON objects [3], which is an open standard format designed for representing geographical features such as the basic geospatial shapes as discussed above and is stored in JSON like format, which can be seen in the examples in figure 1 [4] [5].

MongoDB does not require documents to conform to a strict schema for the containing fields or data type, except for the storing of geospatial data, which is required to be contained within an embedded document.

Figure 1 demonstrates a sample document from the Yelp Business dataset which has been utilised for testing the application and contains over 19000 documents of real business information. The collection has been adapted to conform to this requirement of embedding geospatial data within an embedded document[6] (geospatial data is contained within the "location" attribute) to enable its use in MongoDB.

```
{
        "_id" : ObjectId("5caf84d79a57df481d213c8d"),
        "business_id" : "lxnuq9wJiwLOPJ4uZU2ljg",
        "name" : "Las Vegas Motorcars",
        "address" : "3650 N 5th, Ste 100",
        "city" : "North Las Vegas",
        "latitude" : 36.2258515,
        "longitude" : -115.1328004,
        "review_count" : 3,
        "is_open" : 1,
        "attributes" : {
                "BusinessAcceptsCreditCards" : "True"
        },
        "categories" : "Automotive, Car Dealers",
        "location" : {
                "type" : "Point",
                "coordinates" : [
                -115.1328004,
                36.2258515
                ]
        }
}
```

Figure 1: Yelp Business Sample MongoDB Document

Several notable real-world use cases exist for the storing and querying of geospatial data (GeoJSON) in MongoDB. Allowing us to understand the data around us better and draw new knowledge from our data. Two example use cases include,

- The locating of points of interest near a destination.
  - For example: What is the closest post office to Cardiff University?
- The locating of points or polygons intersecting or within an area.
  - For example: What restaurants are within Cardiff?

Both of which are expressions which can be queried using the different geospatial operators in MongoDB listed below.

Understanding and querying GeoJSON objects in MongoDB can be difficult, and this project aims to support developers in its usage. The application allows users to connect an instance of their MongoDB dataset and visualise and interact with the stored GeoJSON to produce queries and understand the different tools available.

To query geospatial data in MongoDB requires the creation of a "2dsphere" index to allow the querying of data on an earth-like sphere. This is done to prevent the large performance overhead of querying geospatial data and prevent a full collection scan to search for query matches. This result is achieved by indexing the documents thus limiting the number of documents which need to be scanned.

Note: MongoDB uses the WGS84 coordinate referencing system for storing geospatial data.

The four main geospatial query operators in MongoDB which are covered in this paper are Near, NearShere, Intersects and within. Below is a brief breakdown of there functionality and use.

MongoDB Geospatial Query Operators

$geoNear
Specifies a point for which a geospatial query returns the documents from nearest to farthest.

Sample Query:

The following sample query locates geospatial objects in the Yelp Business dataset that are further than 1000 metres from the specified longitude and latitude coordinates (-73.9667, 40.78) and that are within 5000 metres.

```
db.yelp_business.find(
{
        location:
                { $near :
                        {
                                $geometry: { type: "Point",  coordinates: [ -73.9667,
                                    40.78 ] },
                                $minDistance: 1000,
                                $maxDistance: 5000
                        }
                }
        }
)
```

$NearSphere:
Same as $geoNear except when using legacy coordinates where it will calculate legacy points on a sphere rather than a flat plane [7].

$geoIntersects:
Selects geospatial objects which intersects within a specified GeoJSON object.

Sample Query:

The following sample query locates geospatial objects that are fully contained or intersects the specified GeoJSON coordinates.

```
db.yelp_business.find(
        {
                loc: {
                        $geoIntersects: {
                                $geometry: {
                                type: "Polygon" ,
                                coordinates: [
                                [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ]
                                        ]
                                }
                        }
                }
        }
)
```

$geoWithin:
Selects geospatial objects which are within a specified GeoJSON object.

Sample Query:

The following sample query locates geospatial objects from the Yelp Business dataset that are fully contained within the specified GeoJSON coordinates.

```
db.yelp_business.find(
        {
                loc: {
                        $geoIntersects: {
                        $geometry: {
                        type: "Polygon" ,
                        coordinates: [
                        [ [ 0, 0 ], [ 3, 6 ], [ 6, 1 ], [ 0, 0 ] ]
                                ]
                        }
                }
                }
        }
)
```

2.4 Existing partial solutions

Several partial solutions exist relevant to the aims and objectives to this project. These have been outlined below with the reasons for there unsuitability to fulfil the objectives of the project.

Compass

Compass is a GUI for MongoDB, created by the company as an intuitive way to interact with users databases [8].

It allows users to "visualise, understand, and work with their data through an intuitive GUI" and supports CRUD operations amongst other features. In the beta build of the application, support for generating and visualising geospatial data has been added.

However, interaction using the geospatial map in Compass is currently limited in its ability to generate queries using the different spatial query operators. At present interaction using the GUI is limited to generating queries within a circle polygon, using the $geoWithin operator. No explanation or guidance is provided on how to generate this or any other queries using the map interface in Compass.

The visualisation of geospatial data in Compass currently only exists in the 1.3.0-beta build onwards and is limited in its support of the different geospatial operators and provides no support for $geoIntersects, $near and $nearSphere. As well as the ability to draw different query regions other than a circle and documentation on how to use the functionality.

Figure 3 demonstrates the workflow for query creation in Compass and exhibits its limited capabilities in creating geospatial queries and supporting the users understanding of the topic.

The workflow demonstrates the steps taken to,

- connect a user's MongoDB instance to Compass,
- visualising the Yelp Business dataset and
- creating a query for the only supported geospatial operator ($geoWithin).

To summarise query creation functionality is limited in Compass and provides minimal supporting documentation for the different MongoDB operators and specifiers. The application developed for this project aims to address this and provide a more intuitive and comprehensive interface to "support developers in the creation and understanding of MongoDB geospatial queries".

Compass Workflow: Connecting a user's MongoDB instance

Connecting a user's MongoDB instance to Compass is intuitive and easy to do. The connection wizard guides you through the connectivity of the MongoDB instance and is consistent in design, styling and displays appropriate error messages on input. The following workflow demonstrates this for a hosted MongoDB instance containing the Yelp Business collection.

(a) MongoDB instance connection details input

(b) MongoDB Database selection

(c) MongoDB Collection selection

Figure 2: Compass MongoDB instance connectivity

Visualising and generating a $geoWithin query within Compass

The MongoDB collection used for visualisation and query generation is the Yelp Business dataset (Fig 1). The workflow demonstrates a step by step process for creating a $geoWithin query in Compass and discusses the limitations and usability of the interface.

Compass: Analysing Schema
Map and geo query functionality is visible under the Schema tab within Compass.

However, Locating this functionality is not easy with no clear guidance in the user interface or supporting documentation of the tools online. Notably, this functionality is still in beta and may not be fully integrated in the product's workflow but in its current state is difficult to find.

Compass: Visualised Collection data
After analysis of the Yelp Business dataset is complete, a map interface is displayed visualising the analysed schema. Notably, there is no indication of the number of geospatial objects displayed on the map or any way to determine the visualised documents. Both of which help support the users understanding of the visualised collection. Furthermore, the instructions on query creation are obscured on the bottom left corner of the map making it difficult for the user to determine how to do this.

Compass: Query Creation
To create a $geoWithin query in Compass press "Shift" and drag, after which the query region is drawn on the map.

Map interactivity to create the query and select the query region is intuitive. However, it is limited for drawing different shape regions, creating queries for all the different MongoDB geospatial operators and lastly its ability to filter the collection without inputting a MongoDB query.

Figure 3: Compass MongoDB instance connectivity

geojson.io

geojson.io is a web-based application that allows users to visualise their GeoJSON [9]. It is a very basic tool which allows users to paste their GeoJSON directly into the website. It is suitable for experienced developers to be able to quickly grasp how their data is connected and interact with the map by drawing markers and polygons, it was not designed to support the connectivity of MongoDB or any other database and does not offer any support for the generation of MongoDB queries.

Figure 4: GeoJSON.io Interface

2.5 Likely stakeholders in the application

The primary stakeholders in this application are new and existing developers in addition to MongoDB Inc.

The primary focus for this application is to support new developers in their understanding of the different geospatial tools in MongoDB. However, existing developers with knowledge of the subject may also have an invested interest in this application as they can visualise and determine patterns in geospatial data and identify relationships between objects.

Likewise, MongoDB Inc. and other third-party GUI tools for MongoDB such as Robo 3T [10] may have an invested interest as they have the opportunity to enhance their current geospatial querying methods by implementing or reproducing similar functionality in their GUI tools for MongoDB and enhance the user's experience.

2.6 Tools and Software Used

Several tools and methods have set the foundations to this application including MapBox, Leaflet and the MongoDB node.JS driver.

MapBox

The first tool is MapBox which handles the rendering of the map by seamlessly joining individual images to draw the map [11]. Doing so allows for the level of detail for the target area to be adjusted based on the user's zoom and is much less computationally demanding to use image tiling, compared to working with a single large map image.

Leaflet

Building on this to allow the plotting and drawing of geospatial objects on the map a library called Leaflet provides this functionality [12]. In addition, it also handles the clustering of geospatial objects which helps improve performance and only displays the individual points at a higher level of zoom.

To render geospatial objects Leaflet requires the data to be in GeoJSON format. The MongoDB datasets I have been working with are natively not stored in this standardised format and have had to be converted to GeoJSON to support this.

MongoDB Node.js Driver

Finally to allow connectivity between the backend of the application and the users databases, the MongoDB Node.js Driver has been used to provide this functionality.

2.7 Alternative Tools

MapTiler, Mango (MapBox Alternative)

MapBox is the most widely used cloud mapping service; however other competitors exist in this space including, Mango [13] and MapTiler [14]. For the scope of this project, there is no difference between the different services and is only impactful in the different map styles and the minimum uses of a free license. However, with Mapbox being the most widely adopted, there is a broader collection of documentation and guidance for the MapBOX service.

deck.gl (Leaflet alternative)

Deck.gl is a data visualisation framework [15] designed to support large datasets. Similar to Leaflet.js it supports visualisation of large two-dimensional GeoJSON datasets but additionally supports more complex overlays including the rendering of 3d layers; which can be useful in representing the mapping of cities and viewing data at a higher dimension.

However, support for querying using the different MongoDB geospatial operators is limited to solely two-dimensional data.

So although the deck.gl framework has a wider suite of visualisation tools in comparison to the chosen Leaflet.js library. There is a lack of supporting documentation available for this framework, to produce interactive maps. Furthermore, as discussed MongoDB currently does not support the querying of more than two-dimensional data, meaning for the use cases of this project, Leaflet.js provides all the necessary functionality needed to deliver the visualisation and map drawing tools.

Figure 5: Deck.gl Interface

D3 (Leaflet Alternative)

D3 is a data-driven Javascript library that is used to "produce dynamic, interactive data visualisations in the web browser"[16]. For early implementation development, D3 was used in place of Leaflet to handle map visualisation (Section: 4.4).

The library supports a wide plethora of data visualisation uses, among which includes map visualisation, which is extensively documented with detailed code examples available. But, as discovered the library lacks support for specific web mapping functionality including interactivity tools such as pan and zoom and the ability to draw on the map (Section: 4.4).

3 Specification and Design

3.1 Functional Requirements

The following section outlines a detailed set of functional and non-functional requirements for the application.

Requirement
The user must be able to connect there MongoDB instance using the application's user interface.
Acceptance Criteria:

- Using the application's web-based user interface the user must be able to establish a connection to their MongoDB instance.
- The system should be tolerable to invalid entry and inform the user.

Requirement
The user must be able to visualise multiple collections of Geospatial data on the map from their connected MongoDB instance.
Acceptance Criteria:

- The system should handle visualisation of a subset of 1000 geospatial objects from the connected MongoDB instance.
- The map visualisation should be adjusted based on collection filters applied.
- The map visualisation should adjust based on the zoom and pan of the map. For example, zooming to view a smaller region of the map should change the visualised geospatial objects to that of this region.

Requirement
The user must be able to generate MongoDB Geospatial queries through interaction of the map and user interface. Including query selectors,

- $geoIntersects (Returns geometries that intersect other GeoJson objects)
- $geoWithin (Returns geometries entirely within a specified shape)
- $near (Return Geospatial Geometrics objects in proximity to a point)
- $nearSphere (Return Geospatial Geometrics objects from nearest to farthest using spherical geometry)

And Geometry Specifiers,

- $maxDistance Specifies a maximum distance to limit the results.
- $minDistance Specifies a minimum distance to limit the results.

Acceptance Criteria:

- Using the user interface the user should be able to generate valid MongoDB geospa-

tial queries for each of the different operators.

Requirement
The user must be able to test the generated MongoDB query against the connected collection.
Acceptance Criteria:

- The tested query must display a subset of the first five results from the query in the user interface.
- The user interface must allow for the entry of a query or for the query to be modified and for results to be displayed.
- The system should be tolerable of invalid query input and display an appropriate error message in the user interface.

Requirement
The user must be able to visualise the query together with an explanation of each component of the generated query.
Acceptance Criteria:

- The user interface must explain each component of the generated query.

3.2 Non Functional Requirements

Requirement
The applications user interface must be responsive when working with large datasets.
Acceptance Criteria:

- The user interface should be responsive for collections with ten thousand Geospatial Elements.
- The map should be visualised within five seconds.
- Map interactivity tools including pan and zoom should be responsive.

Requirement
The application should be intuitive and easy to use.
Acceptance Criteria:

- The application must be tested with real novice developers to assess the effectiveness of the project in supporting users to generate and understand geospatial queries.
- The application will adhere to Nielsen's 10 Usability principles.

Requirement
The application must be tested and evaluated with multiple MongoDB datasets of varying size.
Acceptance Criteria:

- The application must function as intended including all map and query generation functionality, with different MongoDB datasets including,
  - MongoDB restaurants sample dataset
  - Yelp Open Business dataset
  - Ordnance survey Points of interest and open Roads

Requirement
The applications web interface should be compatible with varying browsers including Chrome (v73+), Firefox (v65+) and Microsoft Edge (v44+).
Acceptance Criteria:

- The application's functionality including all map and query generation functionality must be compatible with,
  - Chrome (v73+),
  - Firefox (v65+) and
  - Microsoft Edge (v44+).

Requirement
The application's user interface should adhere to usability standards.

Acceptance Criteria:

- A heuristic evaluation must be completed to ensure usability standards of Nielsen's 10 Usability principles are followed.

Requirement
The application must be tested to ensure all paths of the application function correctly.
Acceptance Criteria:

- Manual test cases need to be completed to verify the different paths of the application work as intended.
- Automated test cases need to be in place to ensure the stability of the code.
- User testing will test different paths of the application's functionality work correctly.

3.3 User Interface Design

With the focus for this project being on "aiding developers in the creation and understanding of MongoDB geospatial queries"; having an intuitive, easy to use and aesthetically pleasing user interface has been an essential focus for the application. Considerations into usability guidelines and design choices such as layout and design of widgets have driven user interface development. Furthermore, the design has undergone multiple iterations as a result of,

- the heuristic evaluation of Nielsen's usability principles[17],
- analysis of existing solutions,
- functional and non-functional requirements as well as
- feedback on the current designs.

The purpose of doing so has been important in meeting the objectives of the project and avoid creating a more complicated tool than otherwise exists, such as Compass or MongoDB terminal interface.

Figure 6 demonstrates this with a comparison between the initial filtering panel contrasting the final mockup; as evident analysis and completion of a heuristic evaluation has had a massive impact on the final design.

(a) Filtering Panel: Initial Mockup

(b) Filtering Panel: Final Mockup

Figure 6: Iterative design of filtering panel

3.4 User Interface Design Choices

A significant focus during the design of the application has been on providing users with a familiar and intuitive GUI to create and understand MongoDB queries. The following section explains decisions into the layout, controls and components of the interface to create an easy to use interface.

These include considerations into input control factors [18], CARP principles [19] and use of familiar components which have all been influential factors in the design.

User Interface Components

Input Fields

Input fields are one of the most recognisable user interface components and offer flexibility in form input. Despite this, it is easy to make mistakes on entry as a result of confusion or simply entry mistakes. To prevent this typeahead functionality has been added where possible (Fig: 7) in addition to placeholders show expecting input. Overall input entry has been included where necessary with modifications to address the high chance of input error.

Figure 7: Filtering Panel: Autocomplete Search

Notification Icon

Similarly to keep the user informed of the applications state a notification icon has been included to alert the user of the number of current active MongoDB collection filters. This is helpful as from the map alone it may not be clear a subset of data is being visualised and consequently will result in a different generated query and results.

Figure 8: Notification Icon

Progress Tracker
Displayed beneath the MongoDB connection wizard is a progress tracker indicating the current step of the user in the wizard. Including this has been beneficial in keeping the user informed of their current position, what they have completed and what steps are left.

Figure 9: Progress Tracker indicator

Single Selection List
A single selection list is effective at displaying a large number of choices simultaneously in comparison to other UI components for example dropdowns. However, because of this it takes up a large amount of screen space and has only been used in one instance for the selection of the users MongoDB instance; where it has allowed users to see a large number of collections from their selected database at once.

Figure 10: Single item selection list

Dropdowns

Inclusion of dropdowns has been limited where possible in favour of other entry forms such as table selection so that all items are visible. Despite this dropdowns are a familiar interface component and take up less screen space than alternative selection methods such as radio buttons or table selection.

Figure 11 demonstrates the single use of this component in the application's workflow.

(a) Authentication Dropdown

(b) Authentication Dropdown: Expanded

Figure 11: Authentication Dropdown List

Layout

The layout of the user interface has been carefully designed to produce a consistent and intuitive application, with a large emphasis on the visibility of essential functionality for query creation, such as the visualised data (map) and tools to create geospatial queries. To achieve this considerations into the

- use of familiar components [18],
- Positioning of elements and
- CARP Principles [19] (Contrast, Alignment, repetition and Proximity).

To achieve this styling of the background, user interface components and layout have been designed to be invariable to create a sense of unity and familiarity in the application; which is demonstrated in Fig 14 and 16, showcasing the different workflows of the application.

To keep the UI uncluttered information has been encapsulated within collapsible elements or paged to avoid overwhelming the user with a large amount of information. For example, the Query Creation panel has been subdivided into three collapsible paged elements and displays only content relevant to the specific MongoDB geospatial operator.

Figure 12: Query Panel

To draw the user's attention high contrasting colours have been used for user interface components and text to make them stand out and easier to read. An example of this for error message popups are displayed in a highly contrasting colour as well as positioned in the centre of the screen to focus the user's attention.

Figure 13: Database Connection Successful

3.5 Workflow

Illustrated beneath is a workflow of three core use cases in the application. These include,

- Connecting and visualisation of MongoDB geospatial data
- Filtering to view a subset of the geospatial data
- Generation of a geospatial query

Connecting and visualisation of MongoDB geospatial data

Immediately upon launching the application, the user is presented with a landing page wizard, which helps guide the user through connecting their MongoDB instance; which is necessary for the visualisation of their geospatial data and provides a platform to execute their geospatial queries.

In much greater detail section 3.6 discusses considerations of the colour scheme, design and consistency of application widget such as inputs and buttons in each of the panels and pages of the application have been reflected upon to ensure the usability of the interface and overall supporting the user in generating geospatial queries.

To not overwhelm the user, form information has been divided into pages, helping the user process each new piece of information including hostname, port, authentication details, database and collection name.

Dividing the connection interface up was decided upon after review of the Compass workflow which displays all connection information on a single interface. For experienced developers, this allows for faster input of connection details but can be overwhelming for new developers and offers limited space to display help and guidance. As a result, the content has been divided into pages to make it easier to process, and allow for a more in-depth explanation of connection elements.

(a) Landing Page (Hostname and port input)

(b) Authentication Details

(c) Authentication Details Expanded

(d) Database Name

(e) Database Connection Successful

(f) Collection selection

Figure 14: Connecting and visualisation of MongoDB geospatial data workflow

Generation of a geospatial query

Once a connection has been established to the users MongoDB instance, the user is redirected to the map interface displaying a visual representation of their geospatial data within their selected MongoDB collection. The interface has been divided into distinct subsections including the visual representation of the map, tools and filters side panels for query generation and the generated query displayed beneath.

The following workflow demonstrates the process for creating a MongoDB query to select all geospatial objects within a 5km radius of Cardiff University. The workflow has been designed to solve issues with existing solutions such as Compass's query creation workflow, by guiding the user and explaining the different geospatial operators and specifiers.

Map Visualisation
Once a connection has been established to the users MongoDB instance a visual representation of their geospatial data will be displayed on the map. This is together with details of the collection visualised and the number of objects visible which is shown above the map to inform the user to the data visualised.

Query Generation side panel
Located to the right of the map is the Query Generation panel which provides intuitive tools to construct queries and filter the visualised data.

To keep the user-interface uncluttered query generation is divided into multiple panels as not to overwhelm the user with new information. Demonstrated in the mockup a short description of the query operator has been provided to support the users understanding and find out more information before progressing to create the query.

Near query generation

After the user has selected to create a query the query creation panel is displayed for the chosen query builder tool.

In this example, the Near operator provides the user with the ability to either mark a point on the map or find an existing geospatial object within their MongoDB collection.

Near query generation: Existing point selection

To make it easier for the user a typeahead dropdown listing geospatial objects from their collection is provided so that the user can easily select an existing point.

Near query generation: Point map selection

After selecting an existing point, the map will recenter on the point and display a marker at the location.

The user will be able to edit the query generated in the query generated box and make any neccassary ammendments in the side panel by navigating using the dots indicating which slide they are on.

Near query generation: Additional parameters

As mentioned to avoid overwhelming the user, content has been subdivided into panels. For the Near query operator, additional parameters such as min and max distance have been provided on the following page. Again to avoid overwhelming the user less common parameters such as min distance have been collapsed by default.

Figure 16: Filtering to view a subset of the geospatial data workflow

Filtering to view a subset of the geospatial data

Once a connection has been established, the user can interact with the map to better understand the visualised data and generate queries. To refine the displayed data and visualise a subset of the collection can be achieved using the "Filters" tab.

In the following workflow a demonstration of how the collection can be filtered to visualise a subset of geospatial objects which are in the "city" "Cardiff". Doing so offers an intuitive way to hone in on a subset of the collection and can be used in query generation, for example, searching for the nearest post office in Cardiff. The functionality to filter on attributes provides an easier alternative compared to existing tools of writing a MongoDB query to filter the collection.

(a) Filtering Panel

(b) Filtering Panel: Filtering on Attribute

(c) Filtering Panel: Autocomplete Search

(d) Filtering Panel: Apply Filters

Figure 17: Generation of a geospatial query workflow

3.6 Heuristic Evaluation

To assess and identify usability issues in the user interface a heuristic evaluation has been completed to identify any opposing interface design patterns, based on best practices. For the evaluation and to assess compliance, the assessment will be completed against Nielsen's usability principles [17] which is one of the most-used heuristics for usability design.

To assess Nielsen's usability principles, the interface has been evaluated using a numeric scale measuring the extent of compliance and significance of issues found for each heuristic.

Nielsen's usability principles

Outlined beneath is a summary of Nielsen's Ten Usability Heuristics for User Interface Design [17].

- Visibility of system status
- User control and freedom
- Consistency and standards
- Error prevention
- Recognition rather than recall
- Flexibility and efficiency of use
- Aesthetic and minimalist design
- Help users recognize, diagnose, and recover from errors
- Help and documentation

Severity Ratings

To rank the severity of each heuristic, a ranking system from 0 to 4, assessing the frequency, impact and persistence of the issue have been used. Doing so highlights the importance of fixing the problem and provides a measurable metric for the testing [20].

Severity	Rating Scale
0	I don't agree that this is a usability problem at all
1	Cosmetic problem only: need not be fixed unless extra time is available on project
2	Minor usability problem: fixing this should be given low priority
3	Major usability problem: important to fix, so should be given high priority
4	Usability catastrophe: imperative to fix this before product can be released

Heuristic Evaluation results

Heuristic	Severity
Visibility of system status The system status of the connected collection and number of visualised geospatial objects are visible above the map which is informative to the user. However, connection details including, the host, port and database are not visible other than when first input. This could be improved upon to meet this heuristic and communicate better the current state of the application. Furthermore, the system status is displayed appropriately to the user, when the connection is being established or loading, which is done in the form of a loading animation.	2
Match between system and the real world The application uses real-world language and aims to avoid confusing the user with terms they are unsure of by providing explanations of complex terminology where needed. For example, specific MongoDB query terminology has been explained, and for the generated MongoDB query output, a natural language expression has been provided to "speak the users' language". [21]	0
User control and freedom Throughout the application, the user is not limited and has the freedom to navigate with no restrictions. Furthermore, every action is reversible, and the user can return to any previous state. For example, the input of the users MongoDB connection details can be reversed or changed after input.	0
Consistency and standards The program's design is consistent for both elements and terminology. For widgets including, buttons, inputs and error messages are invariable in their styling and terminology. However, there is a minor change in navigation button styling which may confuse the user. The styling of the buttons remains consistent; however, there is a variance in text and colouring which would be better if kept constant.	1

Heuristic	Severity
Error prevention To prevent errors from occurring confirmation modals have been added asking the user to confirm their action before proceeding. For example, when redrawing a query selection bounding box, the user is asked to confirm that this was the action they wished to make. Where possible to prevent errors from happening, typeahead inputs have been added to avoid input error. However, for invalid or empty input, this could be better addressed by highlighting the error before submission of the page. For example, when establishing a connection to the MongoDB instance, it would have been more appropriate to highlight the fields after invalid input rather than on page submission.	1
Recognition rather than recall On the map interface, the user can see which collection they are connected to. However, the connected hostname, port and database are not displayed after input of these details. Making these visible these would be massively beneficial to the user instead of having to recall what connection details they inputted or to validate the correct host and database are connected.	3
Flexibility and efficiency of use For new users who have never used the application before it is clear how to use and navigate with added support for new developers in the form of placeholder, tooltips and suggestions. For example, a placeholder for hostname and port have been added to show example input and tooltips provided on the map interface explaining functionality. Additionally, for expert users who require additional functionality, expansion of panels and further options have been added, e.g. additional options in the Near query panel. Again these have been collapsed (hidden) by default to keep the interface uncluttered and displaying only the essential functionality.	0
Aesthetic and minimalist design No unnecessary content is displayed throughout all of the different user interfaces and follows the same aesthetic design throughout each of the applications states.	0
Help users recognize, diagnose, and recover from errors If the user was to makes an invalid action or entry, feedback in the form of an error message is displayed to the user; helping them "recognise, diagnose, and recover from errors". An example of this occurs when attempting to connect to a non-existent host, where the user is provided with an appropriate error message and allow them to correct it.	0
Help and documentation Where possible help, documentation and useful links have been added to make all of the application's functionality easier to understand. This includes references to the appropriate MongoDB documentation, which is extremely useful as a secondary source of information for understanding geospatial queries in MongoDB.	0

Heuristic Evaluation Summary

Completing a heuristic evaluation has been valuable in providing confidence in the usability of the program and revealed a number of opportunities to enhance the user interface.

A significant usability issue uncovered was the visibility of the users MongoDB instance hostname, port and database details which were not visible once submitted. Doing this prevents the user from having to recall the connection details they inputted and could be resolved by including the details alongside the visualised collection name on the map interface.

The remaining enhancements discovered from the heuristic evaluation are minor cosmetic issues which are easily solvable. These consisted of an inconsistency in the styling of navigation buttons and the enhancement of certain inputs to include typeahead functionality.

3.7 High Level Architecture

Below Figure 18 displays a high level overview of the system architecture.

The user will interact using the web Frontend interface; which is responsible for communicating to the backend of the application, using the necessary Rest API's to retrieve information relevant to the interface and the user's request, such as the retrieval of map data.

The Node.JS backend of the application is entirely responsible for the processing and handling of data from the users MongoDB client (instance); where it will then be passed to the frontend of the application.

For example for the visualisation of map data, the relevant Rest API will request the map data, where the Node.js application will query the user's collection to retrieve this information. Once returned it will be processed and forwarded onto the application's frontend where it can be visualised to the user.

Figure 18: High Level System Architecture

3.8 Class Diagram

Expanding on the high-level architecture, the class diagram outlines a more granular level of the application's class structure.

The class structure for this application adheres to design best practices to make maintainability, testing and addition of new features straightforward. This includes considerations into the grouping of related functionality (high cohesion) and separation of class functionality from one another (low coupling). Sections 3.8 and 3.8 outline the design of the front and backend of the program and how functionality has been separated into relevant classes, additionally making agile development easier. For example, the containment of all Query Builder functionality is contained within QueryBuilder.js and can function independently from other classes.

Figure 19: Class Diagram

Web Frontend

The frontend of the application which consists of the top two levels in figure 19 is made up of frontend technologies,

- HTML: index.ejs, map.ejs
- CSS: stylesheet.css
- JS: createConnection.js, map.js, mapTools.js, mapFilters.js, mapQuery.js

The frontend of the application is responsible for the visualisation of the user interface and handles all of the necessary requests to the backend of the application and subsequently the user's MongoDB instance.

A single stylesheet has been used to handle the complete styling of the application, to enable reusability for all of the different widgets. Doing so makes sure the UI is consistent throughout.

The underlying javascript files are each responsible for processing user interface interaction on the frontend of the program such as map and query creation panel interaction. And for making the necessary API requests to the backend of the program to retrieve important information such as map data and the generated MongoDB geospatial query. Outlined in section 4.2 is a breakdown of different API's in the application as well as a clear description of their use.

Figure 20: Class Diagram: Frontend

Node.JS backend

The backend of the application is responsible for the communication between the users MongoDB client and the processing of incoming requests from the frontend of the application. The backend of the application is handled via Node.js which an open source JavaScript run-time environment.

The server is instantiated using the index.js file where the application's middleware consisting of error handling (errors.js), routing of incoming HTTP API requests 4.2 (routes.js) and the applications configuration file (config.js).

The routes.js file defines all the API routes in the application. For incoming HTTP requests, it maps where the corresponding request is to be processed and responded. In this application, it acts as the glue connecting the frontend of the application to the different rest API's. This includes the routing of the following RESTful API's, to their corresponding API route.

- /api/monogDB
- /api/mongoDB/collection
- /api/map
- /api/query
- /api/query/breakdown

Furthermore, as explained in the high-level architecture design, the backend of the application is also responsible for the communication between the users MongoDB client to retrieve data and test queries.

Figure 21: Class Diagram: Backend

3.9 Node.JS Rest API best practices

APIs provide access to features or data of a service and are used in this solution to retrieve information from the backend of the application and subsequently MongoDB. When designing the different API's needed for this project, best practices were considered to ensure the maintainability of the endpoints. Ensuring API usage is clear will make future project maintainability or extension simpler, as developers will have a clearer understanding of the purpose of the different APIs in the application [22].

For this application, we have used REST as our API architectural style, which takes advantage of HTTP to read, post and delete data. It is well optimised for web-based applications and permits a response with the payload format of JSON; making it advantageous for retrieving MongoDB documents which are stored in a JSON like format (BSON) — making it a great choice in comparison to other popular API protocols such as SOAP, which uses XML for message exchange.

For REST APIs there are no official guidelines on how they should be defined; however, there is a consensus of best practices [22] [23]. Below I have outlined an analysis of these with examples on how they have impacted the design of the endpoints in this project.

Singular noun of documents

For individual resources, the API needs to echo its naming convention to reflect this. For this application, endpoints have been defined accordingly to adhere to this best practice, as demonstrated in the map data endpoint.

- /api/map ✓
- /api/maps ✗

Plural noun for collections

For resources within a collection (contain more than one item), the API needs to reflect this in its naming convention. As demonstrated in the example below, MongoDB documents can comprise of multiple attributes and to adhere to this will need to use the plural noun for the resource.

- /api/MongoDB/collection/attributes ✓
- /api/MongoDB/collection/attributes ✗

Appropriate error handling

Constructing robust APIs with appropriate error handling is essential in system design so the cause of the error is clear and can be acted upon. This can be addressed through the returning of a response that describes the error in detail, including the status code and body containing a description of the error.

In the example beneath you can see a modelled response for an API testing an invalid MongoDB connection; the response returns a valid status code of 400 as well as an appropriate error message.

```
Response:
  Status: 400

{
        "error": {
                "message": "Authentication failed."
        }
}
```

4 Implementation

This section describes the specifics and structure of the implementation and is divided into subtopics, first outlining the infrastructure of the project, backend and frontend of the application; whereafter more specific problems are described in detail.

4.1 Project Infrastructure

The applications infrastructure uses Node.js which is an open-source Javascript run-time environment used to handle both server-side API requests and the visualisation of the program's frontend.

Previously discussed in section 3.8, the frontend of the application communicates through the use of Ajax API requests to the backend of the application to retrieve data. The Node.js backend is exclusively responsible for listening, and processing of incoming API requests from the frontend of the application and communication between the users connected MongoDB instance.

Node.js is best suited for real-time web applications, where performance and speed are crucial. In this application, we are processing and visualising large MongoDB datasets of thousands of geospatial objects; consequently making performance a critical factor in enabling the application to be responsive with large datasets.

In comparison to other alternatives such as Python and PHP, Node.js has a steeper learning curve and thus has resulted in a larger time investment spent on understanding the environment. However, Node.js is extensively documented and has a vast range of supporting Node.js modules which have been valuable in the applications delivery. A more extensive analysis of these has been discussed in section [24].

4.2 Backend

The express.js node module is responsible for the routing and creation of robust APIs. For incoming API requests from the frontend of the application, express.js listens for the specified endpoints, where the relevant callback function is triggered, processed and returned with the appropriate response and status code. Outlined below is a list of all of the key endpoints in this application.

/api/map

The map API is responsible for querying the users MongoDB instance and returning collection data in a prepared geoJSON format which can be used for visualisation. The API request supports filtering in two ways.

Firstly the databases collection data can be filtered to those which match query criteria for specific attributes in the collection — for example, finding map data which

matches a particular attribute in full or partially such as name: "Cardiff Castle" or name: "Cardiff*".

Secondly to handle the visualisation of large datasets a query window can be specified to set the bounds of the visible map (fig: 25); thus limiting the quantity of data that needs to be retrieved, processed and visualised. For example, with the map focused on the UK region, it does not need to retrieve data outside of this window.

After each map adjustment in the form of pan and zoom, the map API is called with the updated query window, and data is fetched for the new query region. Doing so helps resolve the most significant limitation in the application of visualising large datasets, which can overload the system and cause the interface to be unresponsive and slow. A restriction of 1000 geospatial elements has been applied to prevent this. However, retrieving data for only the visible map region focuses data selection to a limited set and allows the visualisation of more data; especially when focusing on a smaller region of the world.

Once data has been retrieved from the users MongoDB collection, incoming data is preprocessed and converted into "GeoJSON; which is an open standard format used to represent simple geographical features"[4]. This is achieved through the geojson node.js module and is required so that the data is in a standardised format and can be visualised on the frontend.

```
router.get('/', async (req, res) => {
        let collection = req.query.collection;

        let filters = filterBuilder(req.query.filterCollection);
        let mapBounds = mapBoundsBuilder(req.query.mapBounds);
        let findParam = Object.assign({},filters, mapBounds);

        const client = await mongoDBConnection.establishConn(req.query);
        const db = client.db(req.query.database);
        db.collection(collection).find(findParam).limit(config.mapLimit).toArray(
            function(err, result) {
                if (err) throw err;
                res.status(200).send(
                        parseJson(result)
                );
        });
});
```

Figure 22: Map API function

/api/mongoDB/test-connection

The test-connection endpoint adds the functionality to establish and test the connection to the users MongoDB instance. Upon connection, the instance is validated to verify that one or more collections exist. Whereafter a valid status code and message are responded, informing the status of the connection.

This API is primarily used to establish that a connection exists for the instance upon the input of connection details when connecting the users MongoDB instance.

The core code (function) responsible for establishing a connection to the user's instance is utilised by every endpoint which requires a connection to the user MongoDB instance and is required before retrieval or processing can take place.

```
function establishConn(req) {

        let url = req.url;
        let username = req.username;
        let password = req.password;
        let database = req.database;

        /*  Auth or no Auth */
        if (username !== '' & password !== '') {
                url = 'mongodb://' + username + ':' + password + '@' + url + '/' +
                    database;
        } else {
                url = 'mongodb://' + url + '/';
        }

        return new Promise((resolve, reject) => {
                MongoClient.connect(decodeURIComponent(url), {
                useNewUrlParser: true
                }, function(err, client) {
                        if (err) {
                                console.log('Error ' + err);
                                reject(err);
                        } else {
                                console.log('connected to ' + req.url);
                                resolve(client);
                        }
                })
        });
}
```

Figure 23: MongoDB establish connection function

/api/query

The core focus of this project is to "aid developers in the creation and understanding of MongoDB geospatial queries" where as a result, one of the core pieces of functionality for this application is the creation of MongoDB queries using the map interface.

The query API builds the query based on passed parameters and outputs it into a format processable by MongoDB.

/api/query/breakdown

In addition to supporting developers in creating MongoDB geospatial queries, the application aims to improve their understanding of the queries.

On the frontend of the application, the different query operators and explained, however, the generated MongoDB query output can be difficult to understand and process. To facilitate this, a natural language alternative to the generated query is provided; which explains the query in a human-readable format.

/api/mongoDB/execute-query

This API facilitates the testing of MongoDB queries and the retrieval of a subset of results, thus enabling users to test, execute and capture results; which is important in validating that the query is returning desired results.

The main use cases for the API is to test and retrieve results for the generated query. However, users can modify the generated query and input and test custom geospatial queries. For ill-formed queries, the endpoint is tolerable and will return a valid error message with the result.

Error Handling

Operational errors are not bugs in the program but problems experienced by correctly functioning programs, such as connectivity issues to the users MongoDB instance. These run-time problems need to be handled correctly so that the system can continue to operate uninterrupted from these errors.

Outlined in section 3.9, discusses the importance of API error handling best practices, including error response and status code. To ensure this error handling has been externalised through the use of a middleware function[1]. The function ensures that the response returned is in a consistent format with a valid status code.

[1]Middleware are functions which have access to the API request object.

4.3 Frontend

The frontend of the application is displayed to the user through a web-based interface, made up of, HTML, Javascript and CSS as the underlying technologies. In addition to several important Javascript libraries, Jquery, Leaflet and D3 which has eased implementation and provided essential front-end infrastructure.

Visualisation of the map

For this application, a map interface provides intuitive interaction to create geospatial queries and better understand visualised MongoDB geospatial data.

To achieve this the open source library Leaflet.js provides the web mapping, interactivity and layering of geoJSON for the map. However, for visualisation of the underlying world map Leaflet.js relies upon a tiling service, for the requested map; where the individual tiles are seamlessly joined together to draw the map. As discussed in "Tools and Software Used" the benefit to doing this is it is computationally less demanding in comparison to the use of a single high detailed image and allows for a much higher level of detail with high zoom levels.

Figure 24: Tiled Map Example [1]

Additionally, Leaflet.js is used to visualise users MongoDB collection data on top of the underlying map. Data is retrieved through an Ajax API request to the /api/map endpoint defined above. On retrieval, the data is appended and represented as a GeoJSON layer on top of the underlying world map.

Each geospatial object retrieved from the collection is appended as a new HTML DOM element, resulting in a potentially large performance overhead. Because of this visualisation is restricted to displaying no more than 1000 objects at any one time, to ensure the interface remains usable. To overcome this limitation, users can filter their MongoDB collection to visualise geospatial objects of interest such as those that contain the attribute name, "Cardiff". In conjunction with displaying only geospatial objects within the current view window. As the user refines the search to a smaller query window

(zooms in to a smaller region), an increasing percentage of collection data is visible, where the first thousand geospatial objects will be retrieved.

One of the issues with retrieving the first thousand documents which match the search is after panning or zooming the map it can create the illusion of objects appearing which can be confusing to the user. The reason for this is as the user refines the search more elements can be retrieved and as a result appear on the map.

One strategy proposed to resolve this was the sampling of points throughout the collection rather than retrieving the first thousand documents. The reasoning for was the distribution of points would be more even and as a result less confusing with less large regions of points appearing, as tested datasets were typically ordered and appeared from left to right. To achieve this, the MongoDB $sample operator was tested, which "randomly selects the specified number of documents". The outcome of this was a more even distribution; however, upon map interactivity, it resulted in new points being sampled every iteration and points constantly changing upon map movement.

For this proposal to work, consistent sampling of MongoDB data needs to occur, however with no existing functionality and other solutions such as sampling every nth row being detrimental to performance. The current solution continues to sample the first thousand documents with potential for future improvement.

(a) Querying full map
(Retrieval of all data)

(b) Querying specific region
(Retrieval of subset)

Figure 25: Query Window Functionality

To promote intuitive interaction for query creation, users can define the geoJSON query region for the different geospatial query operators using the map, for example, drawing the bounding region for $geoWithin queries. Doing so developers can visually distinguish the selected documents and quickly assess whether the query is behaving as needed.

- $geoNear: Drawing of marker to specify a point for which documents are returned from nearest to farthest.
- $geoIntersects and $geoWithin: Drawing of a bounding box to specify documents which either intersect or are contained within.

Drawing markers and regions on the map is achieved using an extension to Leaflet.js

called Leaflet draw [25]. Latitude and longitude coordinates are captured for each point of the drawn shape and passed as a parameter to the two query builder APIs (api/query, api/query/breakdown). Whereafter the two API's return a structured MongoDB query and natural language variant.

Figure 26: Drawing bounding box example for the $geoWithin operator

4.4 Challenges faced

Generation of geospatial queries for multiple collections

One of the unforeseen problems encountered during the implementation of this project was to include support for querying multiple collections.

Relational databases such as MySQL support the querying of multiple tables, however for NoSQL databases such as MongoDB they are poorly optimised to join collections, especially for those of which are sharded [2] (distributing data across multiple machines). Unlike relational DBs MongoDB is a NoSQL database and is not required to conform to any established relational schema.

To resolve this and create geospatial queries over multiple collection severals potential approaches could be used. Firstly, and the most performance optimal solution would be to combine the two collections into a single collection before querying. This can be achieved using aggregation, which will process the data from the two collections and return the computed results into one collection. Alternatively, at the cost of decreased performance and which defeats the purpose of using a BASE orientated database solution, would be to perform a join on the two collections using $lookup operator.

Change in underlying map visualisation library

The completed application uses Leaflet.js to provide web mapping, interactivity and layering of geoJSON, for the map. However, early development used the D3 Javascript library[16] which is a more versatile, lower level library I had had previous experience with. D3 is designed to produce "dynamic, interactive data visualisations in web browsers"[16], compared to Leaflet.js which is specifically designed to build interactive maps.

Development of core map visualisation was straightforward using D3 thanks to the popularity of the resource and previous knowledge of the library. However, as development progressed the implementation of more complicated features became increasingly difficult with support for specific mapping tools lacking, including interactivity tools such as pan and zoom. As a result, the decision to transition to Leaflet.js was decided because of the significant time investment being spent in sourcing documentation and developing complicated mapping features, which were present in other resources such as Leaflet.js.

[2] Sharding is a method used to distribute data across multiple machines and is used to address system growth through horizontal scaling.

(a) Initial Implementation: D3.js

(b) Final Implementation: Leaflet.js

Figure 27: D3 versus Leaflet.js Comparison

Filtering on MongoDB Document Attributes

One of the features of the application is to allow users to filter the generated query and visualised MongoDB map. An unforeseen challenge during the implementation was to predetermine the field type which is required when producing the MongoDB query and for typeahead functionality. Because of this filtering is limited to only attributes of type "string" with further opportunity to support other data types. The reasoning for this is the used MongoDB Node.js $regex operator is limited and doesn't offer support for the other data types including Integer, Date and Object.

To resolve this, an API has been added to determine the field type whereafter further development needs to take place to modify the generated query to filter on other data types.

```
router.get('/attribute/type', async (req, res) => {
        let collection = req.query.collection;

        const client = await mongoDBConnection.establishConn(req.query);
        const db = client.db(req.query.database);

        const attr = req.query.attribute;
        let attrType = '';

        db.collection(collection).findOne({}, function(err, result) {
                if (err) throw err;

                for (let key in result) {
                        if (result.hasOwnProperty(key)) {
                                console.log(key + ":" + attr);
                                if (key === attr) {
                                        attrType = typeof result[key];
                        }
                }
}

        res.status(200).send({
                'attributeType': attrType
                        });
                });
});
```

```
/home/danielh112/Documents/University/Year 3/FinalYearProject/Project/node_modu
les/mongodb/lib/utils.js:132
      throw err;
      ^

MongoError: $regex has to be a string
    at queryCallback (/home/danielh112/Documents/University/Year 3/FinalYearPro
ject/Project/node_modules/mongodb-core/lib/cursor.js:248:25)
    at /home/danielh112/Documents/University/Year 3/FinalYearProject/Project/no
de_modules/mongodb-core/lib/connection/pool.js:532:18
    at _combinedTickCallback (internal/process/next_tick.js:132:7)
    at process._tickCallback (internal/process/next_tick.js:181:9)
[nodemon] app crashed - waiting for file changes before starting...
```

Figure 28: Node.js MongoDB Regex Error

5 Testing, results and evaluation

The following section evaluates the success of the application in meeting functional and non-functional requirements and to what extent the system works as intended. User testing, test cases and a heuristic evaluation have been completed to test the application as well as considerations into more tests which would help provide added confidence.

5.1 Requirements Evaluation

Functional Requirements

Requirement The user must be able to connect there MongoDB instance using the applications user-interface.
Accepted: Yes Users can establish a connection to their MongoDB instance using the web-based user interface. The connection wizard provides this functionality and is displayed upon launch of the application.

Requirement The user must be able to visualise multiple collections of Geospatial data on the map from their connected MongoDB instance.
Accepted: No After connection of the users MongoDB instance, the map is able to visualise a subset of 1000 geospatial objects from a single collection. Visualisation is limited to a single collection as a result of unforeseen limitations which have been described in detail in section 4.4.

Requirement
The user must be able to generate MongoDB Geospatial queries through interaction of the map and user interface. Including query selectors, • $geoIntersects (Returns geometries that intersect other GeoJson objects) • $geoWithin (Returns geometries entirely within a specified shape) • $near (Return Geospatial Geometrics objects in proximity to a point) • $nearSphere (Return Geospatial Geometrics objects from nearest to farthest using spherical geometry) And Geometry Specifiers, • $maxDistance Specifies a maximum distance to limit the results. • $minDistance Specifies a minimum distance to limit the results.
Accepted: Yes After creation of a MongoDB geospatial query or input of a custom query, the "Query Output" tab displays a subset of five results from the executed query. For invalid query input, the system is tolerable and continues to function as intended, with an appropriate error message being displayed in the user interface.

Requirement
The user must be able to test the generated MongoDB query against the connected collection.
Accepted: Yes After input or generation of a MongoDB geospatial query, the "Query Output" tab displays the first five results from the executed query. For invalid query input, the system is tolerable and displays an appropriate error message in the user-interface and continues to function as intended.

Requirement
The user must be able to visualise the query together with an explanation of each component of the generated query.
Accepted: Yes In conjunction with generating a MongoDB geospatial query, a natural language breakdown is produced explaining each component of the generated query in the "Query Breakdown" tab.

Non-functional Requirements

Requirement
The applications user-interface must be responsive when working with large datasets.
Accepted: Yes The program is responsive for datasets containing over 10000 geospatial objects and the map is visualised in less than 5 seconds for the following test data sets, • MongoDB restaurants, • Yelp Open Business dataset and • Ordnance survey Points of interest and open Roads.

Requirement
The application should be intuitive and easy to use.
Accepted: Yes To assess this a heuristic evaluation, user-testing and SUS questionnaire have been completed, helping to identify whether the application's usability.

Requirement
The application must be tested and evaluated with multiple MongoDB datasets of varying size.
Accepted: Yes The application has been tested with multiple datasets of varying size including, • MongoDB restaurants, • Yelp Open Business dataset and • Ordnance survey Points of interest and open Roads.

Requirement
The applications web interface should be compatible with varying browsers including Chrome (v73+), Firefox (v65+) and Microsoft Edge (v44+).
Accepted: Yes The system has been tested with all listed browsers and corresponding versions to validate the application's functionality works and behaves as intended for all platforms.

Requirement
The application's user interface should adhere to usability standards.
Accepted: Yes To assess whether the user interface adheres to usability standards a heuristic evaluation using Nielsen's ten usability principles has been conducted helping to identify usability problems in the user interface.

Requirement
The application must be tested to ensure all paths of the application function correctly.
Accepted: No Manual test cases and user testing have been completed evaluating the different workflows. However, due to time limitations, no implementation of automated testing has been completed. The result of this is that any changes to the existing code could lead to unknown bugs, as no validation is in place checking modifications to existing functionality. Nevertheless, it would be advantageous to verify that the system continues to function as intended after the implementation of new features or changes.

5.2 Test Cases

Ten test cases have been produced to help validate the success of the project and expose issues in the application. The following section summarises the results of these tests with a full breakdown of the steps and criteria defined in Appendix 1.

The test cases have been created based on the requirements and goals of the project and have been successful in validating functionality and workflows in the program. The overall result of these tests has been positive with a 100% pass rate, for all of the tests.

Test cases summary:

Test No	**Test Case Name**	**Result**
1	Connect a MongoDB instance and select a collection to visualise	Passed
2	Connect a MongoDB instance with invalid connection details	Passed
3	See details of a map data point (object)	Passed
4	Filter map data to visualise only objects with a specific value (attribute)	Passed
5	Create a query to find all objects near a specified point	Passed
6	Create a query to find all geospatial objects intersecting a specified region	Passed
7	Create a query to find all geospatial objects within a specified region	Passed
8	Pan and zoom dense regions of map data	Passed
9	Test custom MongoDB geospatial queries	Passed
10	Test custom invalid MongoDB geospatial queries	Passed

Tests cases are made up of a combination of valid and invalid use cases aiming to simulate different workflows. One thing to note is that with future development, continual testing of these test cases and expansion should take place, to ensure the test cases remain valid and test all system functionality.

5.3 User-Testing

User testing has been carried out to evaluate the success of this project in identifying whether the application "supports new developers create and understand MongoDB geospatial queries". Individuals have completed five test cases followed by a usability questionnaire measuring the usability of the program. The questionnaire utilised SUS (System Usability Scale[26]) as a model for the questions and is an industry standard. The assessment comprises of ten questions with five possible responses for each.

The following section outlines a summary of the results for both tests, with a full breakdown included in appendix 2. As evident in table 1 usability testing was a success and highlighted some valuable improvements to the program's workflow; with only two tests failing in total. Because of the limited number of participants of three, results may not accurately depict the consensus of the program's usability. Although have still been beneficial in highlighting improvement areas and used as a determining factor to assess whether the project was successful in meeting its aims and objectives. For the continuation of this project, completion of further user testing would be beneficial to assess current findings and potentially uncovered new knowledge.

Prior to conducting user testing, testers were made aware of the use of data in this project and how they would remain anonymous. In addition to this, an overview of the programs and tests purpose were clearly explained to the participant.

Summary of User interface Testing

Test case 1: Pass rate: 3/3 **Summary of results** All users were successfully able to connect a MongoDB instance using the application's interface. On the occasion where the user inputted incorrect connection details, they were able to correct the mistake after being prompted with an error message. One thing to note for the completion of this test case, participants were provided with an existing pre-setup MongoDB instance and were provided with connection details before the test.
Test case 2: Filter the map to display only T.G.I. Friday's restaurants Pass rate: 3/3 **Summary of results** All users were successfully able to filter the map to display T.G.I. Friday's restaurants. One thing to note during the completion of this test case was different variations in the naming of T.G.I. Fridays confused a participant when attempting to filter collection on this value - the multiple variations in naming returned a different subset of results and as a result confused the user. • Expected: "T.G.I. Fridays" • Alternative: "Tgi Fridays"
Test case 3: Create a query to find T.G.I. Fridays Restaurants nearest to Chicago Pass rate: 2/3 **Summary of results** For this use case, two out of three testers were able to create the required query. However, a participant expressed confusion when attempting to find "Chicago" by inputting the city name in the filters panel. Two possible improvements could be made to resolve this, firstly, introducing the functionality to allow users to search the map for specific locations such as "Chicago" rather than having to locate it manually. Secondly, an improvement to the description on the filters panel making its use clearer.
Test case 4: Create a query to find all Mcdonald's Restaurants within New York City Pass rate: 3/3 **Summary of results** All users were successfully able to filter the collection to view Mcdonald's restaurants and draw a bounding box around the restaurants in New York City. Different variations in the tool selection for drawing the bounding box was used, e.g. Rectangle, Pencil tool etc. but all achieved the correct result.
Test case 5: View results and an explanation of the created query Pass rate: 2/3 **Summary of results** All users viewed the query results intuitively before being prompted. However, one participant was unsure how to see the "explanation of the created query" (Query Breakdown). To resolve this prompting the user when the explanation of the query has been updated, and providing an interface tour, demonstrating the different tools would both enhance the user's experience.

Table 1: User Testing Summary

System Usability Scale Questionnaire

The results from the SUS questionnaire correlated to user testing results with an agreement that testers would use the system frequently (question 1) and found it consistent and easy use (questions 3,6).

However, for the two participants who were unable to complete one of the two failed test cases; both felt they needed prior knowledge before using the system (question 10). This is understandable as the application is targeted at new developers and requires a basic foundation in knowledge of databases. Based on this data, enhancements in explaining the user interface before interaction would be beneficial such as in the form of a user interface tour to provide the user with a more in-depth understanding.

Based on research, a SUS score above 68 is considered above average and is calculated "through "normalising" the scores to produce a percentile ranking" [26]. The combined mean of the tests conducted averaged at 77.5 which is considerably above the threshold. What's more each of the individual averages was higher, providing valuable support that the application is indeed usable and effective.

Figure 29: SUS Results Summary

SUS Questions
Listed below are the ten questions participants answered with a breakdown of candidates scores viewable in Figure 29 and Appendix 2.

SUS Questions
Q1: I think that I would like to use this system frequently.
Q2: I found the system unnecessarily complex.
Q3: I thought the system was easy to use.
Q4: I think that I would need the support of a technical person to be able to use this system.
Q5: I found the various functions in this system were well integrated.
Q6: I thought there was too much inconsistency in this system.
Q7: I would imagine that most people would learn to use this system very quickly.
Q8: I found the system very cumbersome to use.
Q9: I felt very confident using the system.
Q10: I needed to learn a lot of things before I could get going with this system.

6 Future Work

The goal of this project is to support "developers in the creation and understanding of MongoDB geospatial queries", throughout this project several ideas have developed, as well as shortcomings in the implementation. The following section outlines these and there reasoning.

Visualisation of multiple collections

One of the unforeseen problems during the development of this project was to support users in "visualising multiple collections of Geospatial data" (Functional Requirement 2). MongoDB is not designed or well optimised to support the joining of collections and consequently increases the challenge of implementing this requirement.

However, the functionality to visualise and query multiple collections is a beneficial use case. To overcome this and performance implications discussed in section 4.4, an advantageous enhancement would be to provide the ability for users to aggregate and merge the two collections as a way of overcoming this shortcoming.

Upload or paste MongoDB collection data

A beneficial enhancement would be to support the ability for user's to upload or paste MongoDB JSON files directly into the user interface. Doing so provides a fast alternative to having to connect their MongoDB instance which may not always be possible, particularly when hosted on a different server and a remote connection is required to the database.

Enhanced User testing

To evaluate the success of this application, real user testing has been conducted, helping to highlight potential issues and areas for improvements in the user interface. However, as a result of time constraints, user testing has been completed with a limited sample size of three participants. It would be hugely beneficial to continue the testing with a larger sample size to ensure the results obtained are valid as well as the discovering of new knowledge.

Filtering on MongoDB Documents Attributes

One of the unforeseen challenges during the implementation was to filter on attributes for types other than string. This was a result of limited support for the different data types in the MongoDB Node.js $regex operator (Section 4.4). Added support would allow users to filter the map and produce a generated query for all collection attributes instead of a subset.

An API has been developed to determine the field type of the selected attribute but requires further development to the query generation functionality. This would require research and implementation of alternative Node.js MongoDB operators, which could be used to filter the collection using the alternative data types.

Support for the creation of complex geospatial queries

An enhancement to the systems current support for MongoDB query creation would be to facilitate the creation of complex geospatial queries including MultiPolygon, MultilineString and MultiPoint queries; to better support experienced developers. Outlined beneath are several sample use cases which are achievable from adding support for these GeoJSON shapes.

- Querying MultiPolygons allows you to define multiple distinct regions such as two cities from which to find document matches, such as, finding restaurants in Cardiff and Swansea. Furthermore, you can set interior rings if you need to exclude a particular region.
- And finally, for MultiPoint queries it allows you to specify multiple locations from which to retrieve results from nearest to farthest.

7 Conclusion

The main aim of the project is to produce a program which aids developers in the creation and understanding of MongoDB geospatial queries. Based on the evaluation of user testing and requirement completion, the project has been successful in producing an intuitive and easy to use application for MongoDB geospatial query creation and learning.

Creating a successful solution has required extensive research into likely stakeholders, tools and software which could aid development and an assessment of existing solutions to determine the focus of the application. From which detailed requirements have been constructed setting the goals and foundations for the project.

A major focus for the application has been on usability with a significant investment of time and effort invested in producing a usable and intuitive user interface. Multiple iterations of mockups have been created based upon an evaluation of the user interface through heuristic evaluation, and assessment of the applications use.

Constructed from the design mockups, the implementation of the program was successful using the tools and technologies sourced from the in-depth research. However, several unexpected obstacles were incurred, increasing the complexity of supporting specific requirements such as the visualisation and querying of multiple MongoDB geospatial collections and visualising large datasets. As explained MongoDB is ill-suited towards the querying of two or more collections particular in sharded environment. The result of this is visualisation and querying is limited to a single collection and would be an advantageous enhancement for development.

In summary, the application supports developers in their understanding and creation of MongoDB geospatial queries using an intuitive map interface; with additional project scope for more advanced query generation and visualisation tools.

8 Reflection on Learning

Completing this final year project and bringing it to fruition has been extremely rewarding in supporting my personal development of valuable skills such as prioritisation, adaptability and the advancing of multiple technical skills. This section outlines significant experiences during this project which have enabled me to develop and improve.

The initial plan outlining the aims and objectives for this project was in hindsight ambitious for the timescale. Despite this to guarantee the success of this project, it was critical that I was able to adapt, prioritise and reassess goals to ensure that the primary requirements were met.

For the design of the application's workflow, a larger than predicted time was spent on the design. As a result, adjustments to the project plan (Gantt chart) were made to accommodate this, including modifications to the testing of the application. Having this experience increased my awareness of the projects progress and time management, and was able to utilise these skills for the rest of the project.

Development of this program has been challenging and required me to work with new technologies. Preceding this project the vast majority of learning material has been provided in the form of trustworthy lecture material. However, working independently with unfamiliar technologies has required me to engage my problem-solving skills in understanding and sourcing reliable and accurate material.

The decision to use Node.js for the backend of the application has required me to understand a new programming language as well as implementation specifics such as best practices. Evaluating learning resources reliability and accuracy to ensure they were trustworthy; overall developed my ability to research, assess and source new information through the development of my problem-solving skills.

Throughout this project, I have met weekly with my supervisor to discuss how the project is progressing in terms of development and challenges. Having regular interaction with my supervisor has been beneficial to review progress, however, having a limited timeframe to discuss ideas and alterations has required me to clearly and concisely explain development to ensure there is a clear understanding of the undertaken work and challenges. This is a skill which has continually developed throughout the project and has enabled me to showcase work and problems in a clearer manner.

For the implementation of the user interface, I was eager to begin and as a result, did not invest enough time into its design. The outcome of this was a considerable amount of rework was required for the implementation of the user interface.

The design underwent several iterations as a result of a heuristic evaluation assessing the design's usability and meetings with my supervisor. Having to spend time reimplementing major changes to the mockup resulted in a large amount of rework and lost time. Having this experience has helped me to adapt and reassess the approach of the project; where consequently for future tasks more time was allocated were needed before progressing in the project.

Bibliography

[1] The hitchhacker's guide to tiled maps 2019. [Accessed 23 April 2018]. [Online]. Available: http://www.liedman.net/tiled-maps/

[2] D. G. Chandra. Base analysis of nosql database 2018. [Accessed 12 February 2019]. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X15001788

[3] Mongodb geojson objects 2018. Accessed 17 February 2019]. [Online]. Available: https://docs.mongodb.com/manual/reference/geojson/

[4] Geojson 2018. [Accessed 17 February 2019]. [Online]. Available: http://geojson.org/

[5] H. Butler. The geojson format 2018. [Accessed 18 February 2019]. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc7946.txt.pdf

[6] Mongodb geospatial queries 2019. [Accessed 5 May 2019]. [Online]. Available: https://docs.mongodb.com/manual/geospatial-queries/

[7] Mongodb nearsphere operator 2019. [Accessed 5 May 2019]. [Online]. Available: https://docs.mongodb.com/manual/reference/operator/query/nearSphere/

[8] Compass 2018. [Accessed 18 February 2019]. [Online]. Available: https://www.mongodb.com/products/compass

[9] geojson.io 2018. [Accessed 18 February 2019]. [Online]. Available: http://geojson.io/

[10] Robo 3t 2018. [Accessed 5 March 2019]. [Online]. Available: http://geojson.io/

[11] (2018) Mapbox 2018. [Accessed 5 March 2019]. [Online]. Available: https://www.mapbox.com/

[12] Leafletjs 2018. [Accessed 11 March 2019]. [Online]. Available: https://leafletjs.com/

[13] Mango. [Accessed 12 April 2018]. [Online]. Available: https://mangomap.com/

[14] maptiler. [Accessed 12 April 2018]. [Online]. Available: https://www.maptiler.com/

[15] Deck.gl. [Accessed 12 April 2018]. [Online]. Available: http://deck.gl/

[16] D3 2019. [Accessed 26 April 2018]. [Online]. Available: https://d3js.org/

[17] J. Nielsen. 10 usability heuristics for user interface design. [Accessed 16 April 2018]. [Online]. Available: http://courses.ischool.utexas.edu/rbias/2014/Spring/INF385P/files/10%20Usability%20Heuristics%20for%20User%20Interface%20Design.docx

[18] J. aTidwell. Designing interfaces. [Accessed 6 May 2019].

[19] Presentation design: Principles and techniques 2008. [Accessed 6 May 2019]. [Online]. Available: https://www.presentationzen.com/chapter6_spread.pdf

[20] J. Nielsen. Severity ratings for usability problems. [Accessed 16 April 2018]. [Online]. Available: https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/

[21] A. Kaley. Match between the system and the real world 2018. [Accessed 16 April 2018]. [Online]. Available: https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/

[22] D. Kumar. Best practices for building restful web services. [Accessed 13 April 2018]. [Online]. Available: https://www.infosys.com/digital/insights/Documents/restful-web-services.pdf

[23] C. P. Alessandro Bozzon, Philippe Cudré-Mauroux. Web engineering. [Accessed 13 April 2018]. [Online]. Available: https://link-springer-com.abc.cardiff.ac.uk/content/pdf/10.1007%2F978-3-319-38791-8.pdf

[24] K. Lei, Y. Ma, and Z. T. 2014, "Performance comparison and evaluation of web development technologies in php, python, and node.js," 2014, [Accessed 16 April 2018].

[25] Leaflet draw 2019. [Accessed 23 April 2018]. [Online]. Available: https://leaflet.github.io/Leaflet.draw/docs/leaflet-draw-latest.html

[26] J. B. 1996. System usability scale (sus). [Accessed 28 April 2019]. [Online]. Available: https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html

9 Appendices

9.1 Test Cases

Test Case ID: 1 **Test Purpose:** Connect MongoDB instance and select collection to visualise			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions:			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	Launch application's website	The connection page is loaded	Pass
2	Enter MongoDB instance hostname and port **Test Data:** Hostname: ds135786.mlab.com Port: 35786		Pass
3	Click Next	The system will take the user to the next page in the wizard, and the step indicator is updated accordingly to display the current step	Pass
4	Select the authentication type from the dropdown **Test Data:** Username/Password	The user interface is updated to display the authentication form input for username/password	Pass
5	Enter the username and password for the user you wish to connect from your MongoDB instance **Test Data:** admin password123!		Pass
6	Click Next	The popup is dismissed, and the user is taken to the next page in the wizard, with the step indicator being updated accordingly to display the current step	Pass

7	Enter MongoDB instance Database name **Test Data:** heroku_j7g5sqhn		Pass
8	Click Test Connection	The system informs the user with a popup that the connection was successful	Pass
9	Click OK	The popup is dismissed and the user is taken to the next page in the wizard with the step indicator also being updated	Pass
10	Select the collection you wish to visualise **Test Data:** GeoDB	The selected collection is highlighted, and the "Connect" navigational button is enabled.	Pass
11	Click Connect	The page is updated to display the map interface.	Pass
Comments:			

Test Case ID: 2 **Test Purpose:** Connect MongoDB instance with invalid connection details			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions:			
Step No	Procedure	Response	Pass/Fail
1	Launch application's website	The connection page is loaded	Pass
2	Enter MongoDB instance hostname and port **Test Data:** Hostname: nohost Port: 8000		Pass
3	Click Next	The system will take the user to the next page in the wizard, and the step indicator is updated accordingly to display the current step	Pass
4	Select the authentication type from the dropdown **Test Data:** Username/Password	The user interface is updated to display authentication form input for username/password	Pass
5	Enter the username and password for the user you wish to connect from your MongoDB instance **Test Data:** admin password123!		Pass
6	Click Next	The popup is dismissed, and the user is taken to the next page in the wizard, with the step indicator being updated accordingly to display the current step	Pass
7	Enter MongoDB instance Database name		Pass
8	Click Test Connection	The system will inform the user with a popup stating a connection error of "failed to connect to server"	Pass
9	Click "OK" Button	The popup is dismissed and kept on the same page where they can retest the connection or amend the inputted connection details	Pass
Comments:			

Test Case ID: 3 **Test Purpose:** See GeoJSON objects details (attributes)			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	Hover over a point (GeoJSON object) on the map **Sample Point:** Carlyle Green Pool Stand	A popup is displayed listing the objects name	Pass
2	Click on the hovered object	A full list of MongoDB attributes for the selected geospatial object are displayed	Pass
Comments:			

Test Case ID: 4 **Test Purpose:** Filter map data to visualise only objects with a specific value (attribute)			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	Click "Filters" tab	The filters tab is displayed listing all of the attributes found in the collection	Pass
2	Click attribute element **Test Data:** Click "Name" attribute	The selected attribute is expanded displaying an input field	Pass
3	Enter the name to filter the collection on **Test Data:** T.G.I. Friday's		Pass
4	Click "Apply Filters"	The map is recentered and refined to display only objects with the attribute name, equal to T.G.I. Friday's The filters indicator is updated to show 1 (indicating the number of filters applied)	Pass
Comments:			

Test Case ID: 5 **Test Purpose:** Create a query to find all objects near a specified point			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	Click "Tools" tab	The tools tab is displayed listing all of the collections attributes	Pass
2	Click Near information Panel	Near Panel is expanded	Pass
3	Click "Create Query" Button	Near panel is expanded to display query creation tools	Pass
4	Click "Draw Selection" Button	"Draw selection" button colour changes to purple Hovering over the map will display a marker beneath the cursor	Pass
5	Click on map where you wish to draw the marker **Test Data:** Drawn point at Cardiff	Marker is drawn on map "Generated Query" input box is updated to display the generated MongoDB query "Query Breakdown" displays a natural language version of the query Query output displays results	Pass
6	Click "Next"	Next page of Near query Creation panel is displayed	Pass
7	Enter a value in the max distance input field **Test Data:** Max Distance: 50000	A ring is displayed around the draw marker The "Generated Query" input window is updated to display the generated MongoDB query The "Query Breakdown" tab is updated displaying a natural language version of the updated query Query output displays updated results	Pass
Comments:			

Test Case ID: 6 **Test Purpose:** Create a query to find all geospatial objects intersecting a specified region			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	Click "Tools" tab	The tools tab is displayed listing all of the attributes found in the collection	Pass
2	Click "Intersects" information Panel	Intersects Panel is expanded	Pass
3	Click "Create Query" Button	Intersects panel is expanded to display query creation tools	Pass
4	Under "Draw Bounding Box" click the rectangle drawing tool	Button is selected and colour is changed to purple Hovering over the map will display an edit cursor	Pass
5	Continually click and move to draw the bounding box on the map **Test Data:** Bounding box drawn on map selecting the UK	Bounding box is drawn on the map "Generated Query" input box is updated to display the generated MongoDB query "Query Breakdown" displays a natural language version of the query Query output displays the results	Pass
Comments:			

Test Case ID: 7 **Test Purpose:** Create a query to find all geospatial objects within a specified region			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	Click "Tools" tab	The tools tab is displayed listing all of the attributes found in the collection	Pass
2	Click "Within" information Panel	Within Panel is expanded	Pass
3	Click "Create Query" Button	Within panel is expanded to display query creation tools	Pass
4	Under "Draw Bounding Box" click the rectangle region drawer	Button is selected and colour is changed to purple Hovering over the map will display an edit cursor	Pass
5	Click and drag on the map to draw the bounding box **Test Data:** Rectangle drawn selecting the UK **Test Data:** Bounding box drawn on map selecting the UK	Bounding box is drawn on the map "Generated Query" input box is updated to display the generated MongoDB query "Query Breakdown" displays a natural language version of the query Query output displays the results	Pass
Comments:			

Test Case ID: 8 **Test Purpose:** Pan and zoom dense regions of map data			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	Iteratively double click on area of map **Sample Point:** Zoom on New York City	On map zoom the counter displaying the quantity of geospatial objects continues to display 1000 as the map is zoomed in retrieving new data for the updated view window unless at a high level of zoom	Pass
2	Pan away from dense area of map **Test Data:** Pan South to Highlands	Zoom counter updates after every map movement	Pass
Comments:			

Test Case ID: 9 **Test Purpose:** Testing custom MongoDB geospatial query			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	User inputs valid query **Test Data:** Sample below	Notification ring displayed around the "Query Output" tab	Pass
2	Click "Query Output" tab	Results for the query are displayed	Pass
Comments:			

Test Case 9: Sample MongoDB Query

```
db.GeoDB.find(
        {
                "location":
                { "$geoWithin" :
                        {
                        "$geometry": { "type": "Polygon", "coordinates":
                            [[[-3.693857,50.590103],[-3.693857,53.182893],
                                [1.524779,53.182893],[1.524779,50.590103],
                                [-3.693857,50.590103]]] }
                }
        }}
)
```

Test Case ID: 10 **Test Purpose:** Testing custom MongoDB geospatial query			
Environment: Ubuntu 18.04 - Google Chrome			
Preconditions: The user has connected their MongoDB instance to the application and is on the map interface			
Step No	**Procedure**	**Response**	**Pass/Fail**
1	User inputs invalid query **Test Data:** Sample below	Notification ring displayed around the "Query Output" tab	Pass
2	Click "Query Output" tab	Appropriate error message is displayed in Query Output and application continues to function	Pass
Comments:			

Test Case 10: Sample invalid MongoDB Query

```
db.GeoDB.find(
{
        "location":
        { "$geoWithin" :
                {
                        "$geometry": { "type": "Polygon", "coordinates":
                            [[[-3.693857,50.590103],[-3.693857,53.182893],
                                [1.524779,53.182893],[1.524779,50.590103],]] }
                }
        }}
)
```

9.2 User interface Test Cases

User 1: Test Cases

Test Case 1: Connect a MongoDB instance to the application

Result: Pass
The user successfully connected the MongoDB instance to the application.

However, during input of the instances connection details the user mistypes the password but was able to correct it after being prompted with invalid authentication details.

Note: Database connection details and collection have been provided to the user

Test Case 2: Filter the map to display only T.G.I. Friday's restaurants

Result: Pass
The user was successfully able to filter the map to display T.G.I. Friday's restaurants.

Test Case 3: Create a query to find T.G.I. Fridays Restaurants nearest to Chicago

Result: Pass
The user successfully created a query to find T.G.I restaurants near Chicago, although intuitively the user right-clicked where they wanted to place the near marker instead of selecting the tool from the Near Panel.

Test Case 4: Create a query to find all Mcdonald's Restaurants within New York City

Result: Pass
The user successfully created a query to find T.G.I. Restaurants near Chicago.

When attempting to mark Chicago on the map, intuitively the user right-clicked where they wanted to place the marker instead of selecting the tool from the Near panel.

Test Case 5: Connect a MongoDB instance to the application

Result: Pass
The user viewed the results and breakdown query intuitively before being prompted.

User 1: System Usability Scale Results

		Strongly disagree				Strongly agree
1.	I think that I would like to use this system frequently	1	2	3	**4**	5
2.	I found the system unnecessarily complex	**1**	2	3	4	5
3.	I thought the system was easy to use	1	2	3	**4**	5
4.	I think that I would need the support of a technical person to be able to use this system	**1**	2	3	4	5
5.	I found the various functions in this system were well integrated	1	2	3	**4**	5
6.	I thought there was too much inconsistency in this system	**1**	2	3	4	5
7.	I would imagine that most people would learn to use this system very quickly	1	2	3	4	**5**
8.	I found the system very cumbersome to use	1	**2**	3	4	5
9.	I felt very confident using the system	1	2	3	**4**	5
10.	I needed to learn a lot of things before I could get going with this system	1	2	**3**	4	5

Table 2: User 1: SUS Results

User 2: Test Cases

Test Case 1: Connect a MongoDB instance to the application

Result: Pass The user successfully connected the MongoDB instance to the application.

Test Case 2: Filter the map to display only T.G.I. Friday's restaurants

Result: Pass The user was successfully able to filter the map to display T.G.I. Friday's restaurants. To be noted because of the multiple variants in the naming of T.G.I. Fridays the user filtered on "Tgi Fridays" which only displaying a single object on the map instead of the hundreds expected when filtering on "T.G.I. Fridays".

Test Case 3: Create a query to find T.G.I. Fridays Restaurants nearest to Chicago

Result: Fail The user was successful in filtering the collection but attempted to find Chicago on the map through input in the "Location" filter input and was not clear that this was achieved through the "Tools" panel.

Test Case 4: Create a query to find all Mcdonald's Restaurants within New York City

Result: Pass The user was successfully able to filter the collection to view Mcdonald's restaurants and draw a bounding box to find all Mcdonald's restaurants in New York City.

Test Case 5: Connect a MongoDB instance to the application

Result: Pass The user viewed the results and query breakdown intuitively before being prompted.

User 2: System Usability Scale Results

		Strongly disagree				**Strongly agree**
1.	I think that I would like to use this system frequently	1	2	3	**4**	5
2.	I found the system unnecessarily complex	1	**2**	3	4	5
3.	I thought the system was easy to use	1	2	3	**4**	5
4.	I think that I would need the support of a technical person to be able to use this system	1	**2**	3	4	5
5.	I found the various functions in this system were well integrated	1	2	3	**4**	5
6.	I thought there was too much inconsistency in this system	**1**	2	3	4	5
7.	I would imagine that most people would learn to use this system very quickly	1	2	3	**4**	5
8.	I found the system very cumbersome to use	1	**2**	3	4	5
9.	I felt very confident using the system	1	2	3	**4**	5
10.	I needed to learn a lot of things before I could get going with this system	1	2	3	4	**5**

Table 3: User 2: SUS Results

User 3: Test Cases

Test Case 1: Connect a MongoDB instance to the application

Result: Pass The user successfully connected the MongoDB instance to the application.

Test Case 2: Filter the map to display only T.G.I. Friday's restaurants

Result: Pass The user was successfully able to filter the map to display T.G.I. Friday's restaurants. To be noted the participant took longer than expected to complete the test case in comparison to other candidates. This was a result of the user exploring all of the application's functionality before proceeding to complete on the test case.

Test Case 3: Create a query to find T.G.I. Fridays Restaurants nearest to Chicago

Result: Pass The user successfully created a query to find T.G.I restaurants near Chicago, although intuitively the user right-clicked where they wanted to place the near marker instead of selecting the tool from the Near Panel.

Test Case 4: Create a query to find all Mcdonald's Restaurants within New York City

Result: Pass The user was successfully able to filter the collection to view Mcdonald's restaurants and draw a bounding box to find all Mcdonald's restaurants in New York City.

Test Case 5: Connect a MongoDB instance to the application

Result: Fail The user viewed the results intuitively before being prompted, however, was unsure how to see the "explanation of the created query" (Query Breakdown).

User 3: System Usability Scale Results

		Strongly disagree				Strongly agree
1.	I think that I would like to use this system frequently	1	2	3	**4**	5
2.	I found the system unnecessarily complex	**1**	2	3	4	5
3.	I thought the system was easy to use	1	2	3	**4**	5
4.	I think that I would need the support of a technical person to be able to use this system	1	2	**3**	4	5
5.	I found the various functions in this system were well integrated	1	2	3	4	**5**
6.	I thought there was too much inconsistency in this system	**1**	2	3	4	5
7.	I would imagine that most people would learn to use this system very quickly	1	2	3	**4**	5
8.	I found the system very cumbersome to use	1	**2**	3	4	5
9.	I felt very confident using the system	1	2	3	**4**	5
10.	I needed to learn a lot of things before I could get going with this system	1	**2**	3	4	5

Table 4: User 3: SUS Results