University of Bergen
Department of Informatics

Data Structure, Access and Presentation in Web–GIS for marine research

Author:
Torgeir Mossige Grønning

Long master thesis
June 2013

Acknowledgements

I primarily want to thank my supervisor Torill Hamre for great cooperation and advice during my work on the thesis. I would also like to thank the OpenGEO initiative [1] for providing software, documentation and guides without which my work would have been so much harder.

The tools used in the writing of this thesis were: LaTeX[2] for typesetting (with the IEEE citation style [3]), Texmaker [4] as a LaTeX–GUI, yEd [5] for general models, Dia [6] for UML class diagrams and ckwnc [7] for UML sequence diagrams. The base of the LaTeXtemplate in use is due to [8] and [9]. Thanks are extended to the creators of and contributors to these tools and all other free and/or open source software I have employed during my work.

Torgeir Mossige Grønning
Sunday 2nd June, 2013

Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

AJAX Asynchronous JavaScript And XML

GIS Geographic Information System

GML Geographic Markup Language

HTTP HyperText Transfer Protocol

ISO International Organization for Standardization

ODB Oceanographic DataBase

OGC Open Geospatial Consortium

OOP Object–Oriented Programming

OWS OGC Web Service

SFA Simple Feature Access

SLD Styled Layer Descriptor

SRS Spatial Reference System

WFS Web Feature Service

WKB Well–Known Binary

WKT Well–Known Text

WMS Web Map Service

XML eXtensible Markup Language

1 Introduction

1.1 Background

A *Geographic Information System* (GIS) is a software tool used for storing, viewing, manipulating and analysing geographic data. GIS software has been around almost as long as software itself, as they are useful in a wide variety of scientific, governmental and commercial fields [10].

GIS accessed over the *World Wide Web* (WWW) are called *Web–GIS*. Web–GIS are easily accessible for thin clients, reducing the client–side demands on storage size and computational power. However, additional development challenges for web applications, such as security and compatibility, also apply.

Many GIS software tools and suites exist, some customised for a data set or domain, others more generic. To aid integration and collaboration, standardisation of storage encodings, communication protocols, etc. is needed. This is perhaps especially important to Web–GIS as a part of the WWW, where a historical lack of standards and compliance has injured both developers and users [11]. The main standardisation body in the GIS domain today is the *Open Geospatial Consortium* (OGC) [12].

1.2 Motivation

A GIS named *Oceanographic DataBase* (ODB) is in use at NERSC to aid in research on marine climate in the Arctic ocean. ODB is built specifically for a fixed set of data, with a GUI providing querying, presentation and analysis tools. The data is structured around measurements of values such as temperature and salinity with both spatial and temporal variables.

This system has unfortunately aged to the point where it is cumbersome to use. Especially detrimental is the overall software architecture: ODB is a single–client application, meaning data and processing power must be provided locally. Another central issue is the custom–built approach of ODB. Its underlying geographic data is not accessible by other tools, due to non–standard encoding and access methods. There are also additional miscellaneous shortcomings, including cumbersome navigation and the lack of a data export functionality.

An alternative solution is desired, following modern standards and best practices for geographic data encoding and publishing software architecture. A new system should provide the same basic functionality on the same set of data, while remedying the identified issues with the ODB system. It should also be open to extensions, including the addition of data from new sources.

1.3 Goals

The overall goal of this thesis is the construction of a prototype Web–GIS system as outlined in the previous section: a replacement for ODB. The constructed software system should store and provide access to the data contained by ODB using standardised formats and methods. The system should also have a front–end client that offers the same basic functionality as ODB does for query and analysis of the data, along with some additional, requested functionality. This client should be designed for extensibility and maintainability, and be available to use through internet browsers.

The entire system should be constructed from *Open Source Software* (OSS) and be modular to the point that any component may be independently improved on or replaced. More specific requirements for the system have been set before and during work on the thesis, and are the topic of chapter 2.

1.3.1 Sub–goals

To achieve the overall goal, a set of sub–goals has been identified. There are interdependencies among the sub–goals, goal 1 is for example a prerequisite for completing goals 2, 3 and 4, but the goals were generally worked on in parallel through an iterative process. All sub–goals are presented in more detail in chapter 2. The sub–goals are as follows:

1. Research and determine appropriate standards and best practices
2. Determine criteria for, and make selections to construct software stack

3. Transform geographic data to standard–compliant encoding
4. Provide data through standardised access methods
5. Develop client application
6. Evaluate goal achievement

1.4 Research questions

Some research questions guide the thesis:

- *Is it possible to construct a modern and standard compliant replacement for ODB using open source tools and software?*
- *What are the current relevant standards in constructing marine web–GIS software?*
- *What challenges are associated with modernising GIS systems such as ODB to be web–based and standard compliant, and how may these challenges be approached?*

As with goals and requirements, we elaborate on these questions in chapter 2.

1.5 Thesis structure and outline

With our background and goals presented, we now introduce the remaining thesis text. The text is split into four parts:

- Part 1 (chapters 1 and 2) forms the introduction and problem statement.
- Part 2 (chapters 3 and 4) introduces and outlines relevant concepts and technologies. This part corresponds to sub–goal 1 and 2.
- Part 3 (chapters 5, 6, 7) covers software design and implementation – the solution to our defined problem. This part corresponds to sub–goals 3, 4 and 5.
- Part 4 (chapter 8, 9 and 10) concludes the text by giving an overview of the constructed software system and evaluating the result and work process. This part covers sub–goal 6.

In more detail, each chapter covers the following:

Chapter 1: **Introduction**

Presents the background and motivation for the thesis. The goals to achieve and research questions to answer are introduced. An outline for and overview of the thesis is provided.

Chapter 2: **Problem statement**

Refines and details the goals, challenges and scope of the thesis. The software development aspect is introduced with a conceptual model of the system to be constructed, along with a requirements analysis section.

Chapter 3: **Geospatial concepts and standards**

Gives an introduction to relevant concepts and standards. Simple Feature Access, Web Map Service and Web Feature Service are presented in detail, and other standards are introduced more briefly.

Chapter 4: **Software stack**

With a software stack and its communication protocols and roles outlined in the previous two chapters, a set of software technologies to fulfil these roles are selected. Alternatives are presented and evaluated from a set of criteria that are also determined in the chapter.

Chapter 5: **Data transformation**

The steps required to transform the data from the ODB application to a standard complaint structure are defined. An implementation of these steps is then presented.

Chapter 6: **Data accessor configuration**

The data accessor component of the software stack is configured to publish our data according to web standards.

Chapter 7: **Client design and implementation**

A description of the structure and implementation of the user client in our system. Begins with a description of used design pattern, and then presents each code component of the application.

Chapter 8: **System overview and demonstration**

A guided tour through a selection of use cases. This chapter serves both as a demonstration of the full system and a full explanation of how the components configured/designed in the three previous chapters interact.

Chapter 9: **Evaluation**

The achievement of goals, requirements and the work process is evaluated. The chapter also gives a basis for determining future work.

Chapter 10: **Conclusion**

This chapter concludes the thesis by summarising the state of the system, outlining possible future work and giving an account of the work an learning process.

Three appendices are also included:

Appendix A: **OGC standard examples**

Demonstration of SFA methods and example files for various OGC standards.

Appendix B: **Barnes Surface Interpolation SLD**

A complete GeoServer SLD rendering transformation file.

Appendix C: **WebFeatureService module**

Provides the full source code for a module from the client application along with detailed annotations.

2

Problem statement

This chapter provides more details on the goals, research questions and scope of the thesis. The requirements for the system to be constructed are also presented. A conceptual model of the system described in the overall goal is designed and presented. This model will be built on in Chapter 3 and 4.

2.1 Elaboration on goals

We go into more detail on the sub–goals listed in section 1.3.1.

1. **Research and determine appropriate standards and best practices**

   In replacing the ODB application, we must take special care in not constructing a system that will soon suffer the same fate as ODB did. All data encoding and communication should be performed according to the current standards and best practices in the field of web–GIS. Developing according to this goal requires research into what exactly these standards and practices are.

2. **Determine criteria for, and make selections to construct software stack**

   The new system should not be a monolithic piece of software, but rather a collection of modular components with well–defined roles. It is not our intention to develop all these components ourselves, but to employ available software tools and technologies. Many valid options exist, so we must determine a set of criteria on which to judge and select components. A selection of software must be made according to these criteria. The roles and requirements of each component are discussed in section 2.3.

   For roles where there is no available software to fulfil the requirements, new software must be developed. This is the case for the front–end user client of our system. The

development of this client is covered by sub–goal 5. Principles of modularity and extensibility are just as important for this component as for the others.

3. **Transform geographic data to standard–compliant encoding**

   With the desired data encoding established through achieving sub–goal 1, we must transform the data from ODB into this encoding. The ODB database may be referred to as the *legacy database* and its data may be referred to as *the legacy data*. The transformation process involves getting an understanding of the initial state of the data set and tools by which to transform it. We will refer to the resulting software component that holds these data as a *geographic data store*.

4. **Provide data through standardised access methods**

   We must avoid having the data "locked" to our system by utilising loose coupling between components; storage and presentation must be separated. The data will then be available not only to a specific presentation application, but any client that adheres to employed protocols. Achieving this separation requires that all access of data is handled through standardised methods. We will refer to the component that accesses the data store and publishes the data through standardised interfaces as the *geographic data accessor*.

5. **Develop client application**

   The two previous sub–goals outline a software stack, where the components discussed form the lower layers related to data and data access. In constructing an ODB replacement, an user client with a GUI must also be developed. The requirements for this component are covered in section 2.3, in more detail than for the components of sub–goal 2. This component is referred to as the *geographic presentation service*, *front–end application* or simply the *client application*.

6. **Evaluate goal achievement**

   When all preceding goals are complete, the achievement of each should be assessed. In this process, lessons learned and conclusions can be extracted.

2.2 Research questions

Along with goal statements, some research questions guide the thesis. These questions are to serve as a companion to the more pragmatic sub–goals identified in the previous section. While no part of the thesis text directly target these questions until the evaluation chapter, answers are expected to be produced by completing the sub–goals outlined in the previous section.

The central question is whether or not it is feasible to construct a system as outlined by the goals with our requirements and restrictions (more on this in the next section):

- *Is it possible to construct a modern and standard compliant replacement for ODB using open source tools and software?*

The requirement on standard compliance generates another question:

- *What are the current relevant standards in constructing web–GIS software?*

The work process should also produce some more general knowledge that may be useful in similar undertakings:

- *What challenges are associated with modernising GIS systems to be web–based and standard compliant, and how may these challenges be approached?*

2.3 System architecture and requirements

Sub–goals 3, 4 and 5 set the stage a software development process required to achieve the overall goal. In Figure 2.1, the goals are represented in a conceptual model of a software system. This is also a model of the system as imagined from the overall goal. In the remainder of the text, this model will be referred to as *the conceptual model*.

We now look at the three components in this conceptual model and identify some requirements for each component. The requirements identified here are broad and generic. During Chapter 3, we examine current geographic data encoding and communication standards to refine these requirements.

Standard compliance is the central requirement for all components. We also wish to employ open source software and tools when feasible. We will not list this a requirement here – it is introduced properly in Section 4.1.

The three components presented below are all found in Figure 2.1. As can be seen in this model, they form a hierarchy where the data presentation service depends on a data accessor, which in turn depends on a data store. These components, with these dependencies, form a layered system that will be referred to in the thesis as our *software stack*.

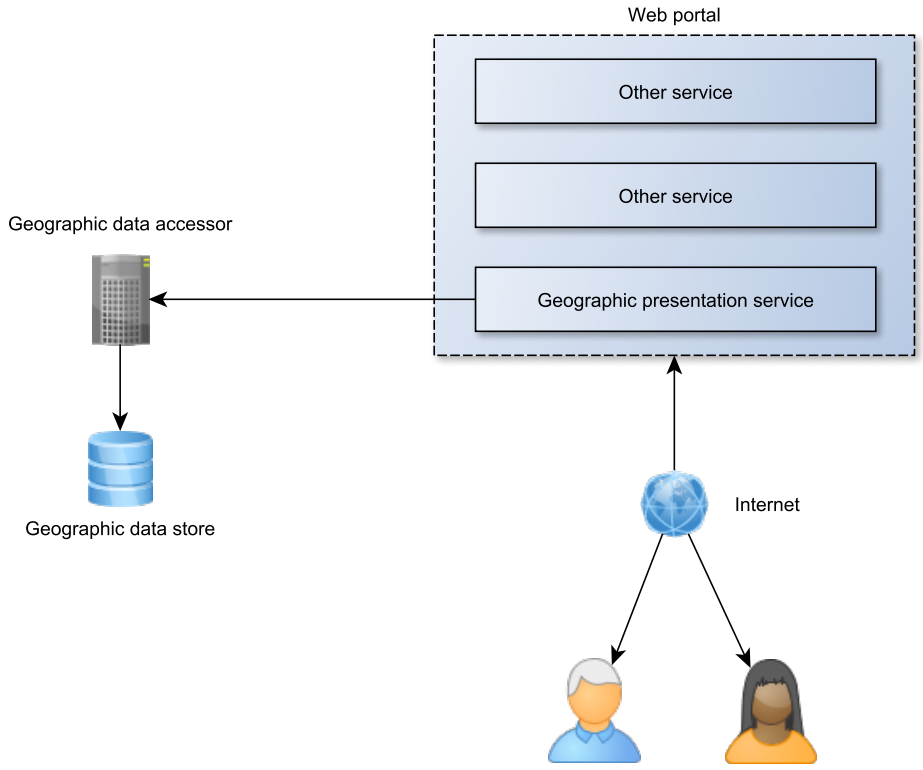


FIGURE 2.1: Conceptual model of the application

2.3.1 Geographic data store requirements

The geographic data store corresponds to sub–goal 3. Successfully transforming the legacy data should result in it being available for query in a standard compliant data store, for example a database. The general requirements are:

- Quality and efficiency in common data store operations
- Compliant with relevant standards in geographic data encoding
- Support for reading text files for loading the legacy data set

2.3.2 Geographic data accessor requirements

The geographic data accessor corresponds to sub–goal 4. It should allow access to the data store through web services. It is not necessarily a single "physical" component – several pieces of software may be required if we need support for several protocols. The requirements are:

- Communicates with our data store

ID	Description
NFR1	Accessible and responsive user interface
NFR2	Quick response time for simple operations
NFR3	Information on status during more complex operations

TABLE 2.1: Front–end: Non–functional requirements

- Provides standardised access methods for the underlying data store
- Configurable for adapting to our data set
- Adequate performance – tasks may be computationally demanding

Exactly what level of performance is *adequate* is hard to determine, as tasks performed on large amounts of data are expected to be computationally demanding.

2.3.3 Geographic presentation service requirements

This component corresponds to sub–goal 5. Its goals are to allow navigation in and analysis of our data set by communicating with the geographic data accessor. Architecturally, it will exist in a web application context, so it should be accessible by a regular web browser. Functional requirements are guided by the existing ODB application and especially noted shortcomings in that application.

In contrast to the other components, where we employ ready–made software that fulfil our requirements, this components was developed as part of the thesis. Development was based on a set of non–functional and functional requirements. The requirements evolved throughout the development period, with the current sets presented here. The initial set was smaller, with new functional requirements added as others were implemented. More on development methodology in Section 7.1.

Table 2.1 lists non–functional requirements by an identifier and a description. Table 2.2 lists the functional requirements. Some of the terms used here, such as *feature* will be clarified in Chapter 3. Read requirements as "The application must provide <*description*>".

The achievement of the requirements will be evaluated (as part of sub–goal 6) based on what functionality is present in the final system and the correctness and completeness of the implementations.

ID	Description
FR1	Query of data based on various parameters (bounding box, time, attribute values)
FR2	Presentation of features and parameters in tabular form
FR3	Presentation of parameter values as graph plots
FR4	Map widget with functionality (pan, zoom, layer selection)
FR5	Map widget/query interactivity: search based on selected area of map
FR6	Graph plot presentation of a parameter value for multiple features
FR7	Calculation and presentation (table, graph plot) of density based on parameters
FR8	Interpolation and display of contour plots based on parameter values
FR9	Export functionality for all tables and graph plots

TABLE 2.2: Front–end: Functional requirements

2.3.4 Scope of functionality

As stated in the overall goal, a *prototype* for a full replacement for the ODB system is to be constructed. The ODB system has a substantial set of functionality, so a full re–implementation is beyond the time frame designated for this thesis. The constructed software stack and client application will serve as a base on which to further develop the system, suited for further implementation of functionality from the ODB system or completely new functionality. Care must be taken not to make design or implementation decisions that would hinder such future developmental work.

Notable features present in ODB that are not implemented in this prototype include:

- Altering data

  The features identified in the functional requirements require read–only access to the data. Altering the data is especially challenging in the web domain where multiple users may access the same data at the same time. This challenge is not present in ODB, where data is stored locally. Data storage, access and client design must accommodate for the addition of read–write functionality.

- Misc. plots and query tools

  A basic, representative set of query tools and plots are to be implemented. The infrastructure developed for these features must accommodate for extensions. The functional requirements reflect this, with an attempt made to select diverse functionality rather that focusing on the detail of a single feature.

3

Geospatial concepts and standards

In this chapter, we introduce standards that are central to the construction of our system. Along with the standards, important concepts are also defined and introduced.

We begin with a brief introduction to the OGC and its role in GIS standardisation. Next, we present a selection of standards and concepts that are very relevant to the remaining thesis. *Simple Feature Access* (SFA) is introduced first, a specification for geographic data stores. We then discuss two important *OGC web services* (OWS); *Web Map Service* (WMS) and *Web Feature Service* (WFS). We also briefly present some other OGC standards that are less important, but still relevant to this thesis. With knowledge of these standards, we update the conceptual model of our system to reflect which standards will be employed.

3.1 The Open Geospatial Consortium

The *Open Geospatial Consortium* (OGC) maintains a large number of open standards that are central to geospatial software development and research. These standards define interfaces and encodings for the storage and exchange of geospatial data. The OGC was founded in 1994 with a board of 8 members from governmental bodies, academia and private corporations. It has since grown to include (as of writing) 482 members.

The OGC standards cover a large range of topics - some general in concept and others focused towards specific domains or situations. For example, the *Geographic Markup Language* (GML) defines a data encoding scheme for describing geographic objects in general. *CityGML*, on the other hand, is a data encoding scheme specifically for describing three dimensional models of cities (with buildings). Some OGC standards are also *International Organisation for Standardisation* (ISO) standards.

3.1.1 OGC standard compliance

The expression "conforming to standards" (frequently used in previous chapters) in the GIS domain largely means to conform to appropriate OGC standards. The OGC distinguishes between *implementing* and *certified compliant* software technologies. The first category uses an OGC standard by fully or partly implementing it. The second category is much stricter and requires that the software be put through a rigorous testing process [13] to qualify.

Our application does not aim to implement any specific standard, but use software that is compliant and conform to any relevant encoding schemes and communication protocols. When choosing software to fulfil technological needs, it should however be considered a major bonus if it is *certified compliant* with relevant OGC standards. This ensures both the correctness and completeness in implementing the standard, as well as being a general sign of quality for the software product.

3.2 Simple Feature Access

The *Simple Feature Access* (SFA) standard (full name: *Geographic information – Simple feature access*) is a specification for the storage and manipulation of geographic data. It consists of two parts: part 1, known as "Common architecture" [14] and part 2, "SQL Option" [15]. It is also an ISO standard [16, 17].

This section will largely be a synopsis of these standard parts in their current version, with an emphasis on the topics most relevant to this thesis. Technical discussions will be shortened and simplified (for the sake of brevity), and parts considered irrelevant or unimportant to this thesis will be omitted entirely.

3.2.1 Part 1 – Common Architecture

The *Common architecture* standard defines a set of data types, storage formats and operations for *simple features*. A *feature* is conceptually defined by the standard as "an abstraction of a real world phenomenon". In practical use, *simple* features can be understood as features with associated geometry (such as a measurement of temperature (a feature) on a specific longitude/latitude 2–D point, on a line of such points, or on a polygon constructed from such points (all examples of *simple geometry*)). The standard also gives a more rigorous definition. We will look at how simple features are constructed, what actions may be performed on them and how they are represented. The standard

uses technical terms (such as *class, abstract class, method*) from the Universal Modelling Language [18][19] that will be reused here.

3.2.1.1 `Geometry` and the class hierarchy

The abstract (non–instantiable) class `Geometry` serves as the base class for all simple feature geometry. All simple features are instances of subclasses of `Geometry`. All instances must have an associated *spatial reference system* (SRS). A SRS describes the coordinate system in which a geometric object is placed. By enforcing that every `Geometry` instance's coordinates are described in a SRS, all instances are commensurable by translation to a common SRS. The standard defines a set of methods for the `Geometry` class. These are split into three categories:

- Basic methods

  Basic methods describe the geometric object represented, giving its dimension, geometry type, SRS, etc. Here are also some "utility"–methods such as `AsText()` and `AsBinary()`, returning a representation of the geometric object in *Well Known Text* (WKT) or *Well Known Binary* (WKB), respectively. WKT and WKB are the topic of section 3.2.1.3. Common to all basic methods is that they take no parameters and are generally retrieval of data rather than computation–focused. Basic methods are essential building blocks for methods in the remaining categories.

- Query methods

  Query methods focus on the geometric object's relation to other geometry. These methods all take one parameter; an instance of the `Geometry` class. There are query methods for testing equality (spatial equality – not object equality), intersection, touching, containing, overlapping, etc. The behaviour of these methods varies with the type of the `Geometry` instances involved.

  We will require query methods when building our prototype, for example to filter out measurements made within a specific *bounding box*. Such functionality requires being able to determine the set of `point` that are within a box–polygon. The use of some query methods is demonstrated in Section A.1.1.

- Analysis methods

  Analysis methods depend upon the accuracy of the SRSs involved. As with query methods, these method operate on two instances of geometry. Methods include measuring the least distance between any two points in two geometric objects, generating the convex hull of an object, generating the union/difference between two objects, etc. The use of an analysis method is demonstrated in Section A.1.2.

It is the responsibility of the implementer to represent all subclasses of the `Geometry` class and provide these methods for them. `Geometry` also has a "companion" abstract subclass in `GeometryCollection`. This is an abstract class for representing a collection of `Geometry` (subclass) instances. There are no constraints on this abstract collection except that all instances it contains must have the same SRS.

3.2.1.2 `Geometry` subclasses

Four classes directly subclass the `Geometry` class (including the previously discussed `GeometryCollection` class). Several of the subclasses have subclasses of their own. The `GeometryCollection` class notably has a subclass for almost all other instantiable classes – representing a collection of objects from these classes (named `Multi-...`, for example `MultiPoint`).

We will now list and present briefly the other subclasses of `Geometry`. In addition to their inherited methods (these will not be repeated in the list), subclasses may provide additional methods. Some of these are found only in certain classes. The `Line` class, for example, has the basic method `length()`, which would not make sense for a `Point` object which by definition has no length.

- `Point`

  The `Point` class represents the simplest geometric object in the Simple Feature Access standard. It consists of an x and y coordinate. It may have up to two additional coordinates if that is required by its associated SRS. It has no methods beyond those inherited from the `Geometry` class. A `Point` is well suited to represent an instance of two dimensional coordinates such as latitude/longitude pairs.

- `Curve`

  The `Curve` class is an abstract class representing curves by being a collection of points with an interpolation algorithm. The `Curve` class provides methods for measuring length, retrieving start– and end–points and checking whether the curve is closed or if it is a ring.

- Lines (`LineString, Line`)

  The `LineString` class subclasses `Curve`, and specifies linear interpolation for its points. `Line` is a special case of `LineString`: it has only two points. There is one additional line geometric object – the `LinearRing`, which is a `LineString` with special properties. We will not discuss it further. Lines are suited to represent relationships between points.

Description	WKT representation
A `Point` at *(5, 10)*	`Point ( 5 10 )`
A `3D Point` at *(5, 10, 7)*	`Point Z ( 5 10 7 )`
A `LineString` of three points	`LineString ( 5 10, 7 14, 2 10 )`
A simple `Polygon`	`Polygon (( 5 5, 5 10, 10 10, 10 7, 5 5 ))`

TABLE 3.1: Well Known Text – example productions

- Surfaces and polygons

  There are four additional classes that represent geometric objects that are not immediately relevant to the work in this thesis, so they will only be mentioned. These are the surface classes `Surface` and `PolyhedralSurface` and the polygon classes `Polygon` and `Triangle`. Polygons are well suited to represent geographic areas such as a seas or national borders.

3.2.1.3 Well Known Text and Well Known Binary

The SFA standard defines two data storage formats: *Well Known Text* (WKT) and *Well Known Binary* (WKB). As the names imply, the formats store and present Geometry instances in a textual and a binary representation, respectively.

WKT is defined rigorously with a grammar in Backus–Naur Form that will not be reproduced here. Some example productions are given in Table 3.1. Note again that the given coordinates hold no meaning relative to anything but themselves without an associated spatial reference system: the geographic features in the table can not be compared meaningfully to each other without knowledge of their SRS.

WKB is not human–readable, and its details are beyond the scope of this text. Since the `Geometry` class has the basic method `AsText()`, a human–readable representation will always be available to the user of a SFA compliant system.

The standard also defines a WKT representation of SRSs with a BNF grammar. In this thesis text, the terms Well Known Text and WKT will always refer to the encoding of geometry that has been the topic of this section, not that of SRSs.

3.2.2 Part 2 – SQL Option

The SQL Option part of the Simple Feature Access standard [15] is a standard for and guide to implementing the Common Architecture in SQL. There exists equivalent

documents for OLE/COM [20] and CORBA [21] technologies as well. However, these standards and/or technologies have fallen into disuse (the latest versions of these documents are over 10 years old), so they will not be covered in this thesis.

The standard provides implementation restrictions and advice for representing all the geometry types found in the Common Architecture (the classes described in Section 3.2.1.1 and 3.2.1.2) using only standard SQL data types. This comprises a significant part of the standard text. However, it also acknowledges the usage of SQL User Defined Types (UDT) as a valid alternative. We will not repeat the at times complex operations described for using only built–in SQL data types.

The SQL Option standard defines two meta–data tables called `SPATIAL_REF_SYS` and `GEOMETRY_COLUMNS` that must be present in a compliant database. The `SPATIAL_REF_SYS` table should contain all SRS used by the system, encoded in SRS Well Known Text format (discussed in the last paragraph of 3.2.1.3). `GEOMETRY_COLUMNS` is a catalogue of all columns of a `Geometry` data type in the database.

3.3 OGC web services (OWS)

The OGC maintains several *web service* standards, commonly named with a *Web* prefix and categorized as *OGC Web Services* (OWS). They define communication protocols for GIS operations, such as requesting information on a specific feature or requesting an image of a map with certain parameters. All these protocols are constructed as standard HTTP request/responses using various HTTP methods. These protocols allow easy construction and interconnection of GIS web applications.

As HTTP communication protocols, a clear *client/server* divide is present in all services. A server that implements a service accepts and responds to valid requests from a compliant client. Two–ways communication is of course also possible if both parts can act as both server and client. The role of both server and client will be discussed when we look closer at the specific services.

3.4 Web Map Service

The *Web Map Service* (WMS) standard is defined in [22]. The standard is supplemented with a number of best practices documents [23, 24] and discussion papers [25–27].

A server that implements the WMS standard will, upon request, serve images generated from some underlying geographic data. We will refer to such a server simply as offering

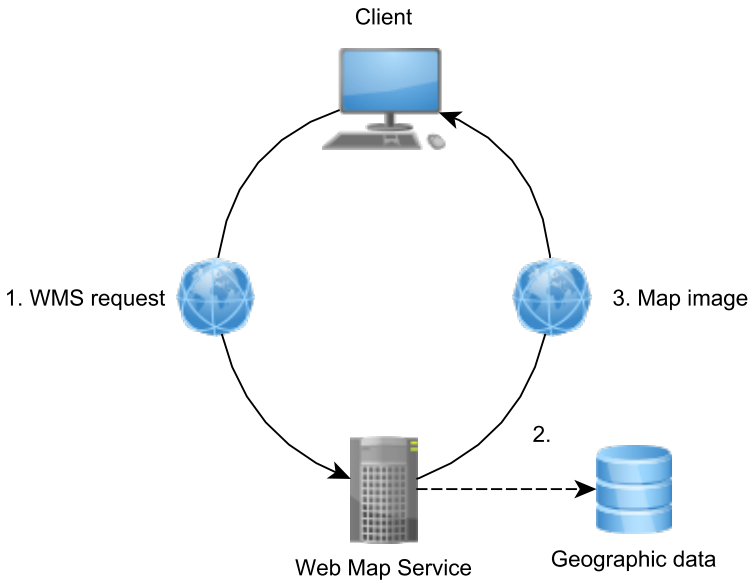


FIGURE 3.1: Web Map Service – GetMap operation

and *being* a *Web Map Service* (WMS). Figure 3.1 is a conceptual model of a successful WMS `GetMap` operation – a practical example and description is provided in section 3.4.2.

Standard compliance will guarantee the possibility of a client combining multiple generated images from several WMSs (either from different physical servers, or from different WMS' hosted on a single server) onto a single surface. This enables, for example, building a web application that displays temperature measurements on top of a variety of underlying map images such as satellite photos, coastline bathymetry, etc.. Since the images from a WMS server are generated dynamically, rather than simply selected from a predetermined set, the application could also support multiple zoom levels with varying amounts of geographic details in the map.

3.4.1 The WMS operations

The WMS standard defines three operations: `GetCapabilities`, `GetMap` and `GetFeatureInfo` (which is optional to implement) . Implementers may also define additional non–standard operations. We will look briefly at each one (from the standard) in this section, while section 3.4.2 re–examines the central `GetMap` operation in more detail.

The operations are carried out as HTTP requests towards the WMS. Both GET and POST–methods are described in the standard, but supporting POST is optional. Using GET, parameters are set in the URL, while using POST the parameters are set and sent to the server in a simple XML–file. There is a fixed set of required parameters for each

Parameter	Description
`service`	Service the request is for
`version`	Version of WMS the request is for
`request`	Which operation the request contains

TABLE 3.2: Web Map Service – common parameters

operation, and additional optional parameters for some. A parameter being mandatory means both that the client must supply it in a request, and that a server must be able to handle any functionality it implies.

We will now look at the operations; outlining their purpose and selected parameters. Example code for performing a complete operation (as illustrated in figure 3.1) is found in the next section. Operations marked *optional* are not required to be supported by implementations.

Note that we will not discuss technical complexities present in the standard such as the valid representation of numerical values in requests, coordinate systems (except briefly for relevant operations), etc.

- Common parameters for all operations

  Table 3.2 presents the parameters that should be in included in all operation–requests (all but `version` are mandatory). The `service` parameter must be set to "`wms`", indicating to the receiving server that the HTTP request is a WMS operation. `version` should reflect that of the WMS the request is targeted to (or, if the WMS supports several, the version the request itself should be identified as). `request` is set to reflect the type of operation that follows, being the name of the operation (as they are named in this list).

- `GetCapabilities`

  GetCapabilities is the simplest WMS operation; it has no additional mandatory parameters beyond the standard set. The WMS server will respond with valuable metadata – a list of its "capabilities". Capabilities include information on the WMS *service* itself (service name, contact information, etc.), what WMS operations it provides and information on available geographic data.

  Optional parameters include `format`, which specifies which data format (in MIME–type) the capability–response should be served as. The standard defines quite extensively the contents and phrasing of the response.

- `GetMap`

Parameter	Description
`layers`	List (comma–separated) of map layers
`styles`	List (comma–separated) of styles
`SRS`	Spatial reference system
`BBOX`	Bounding box coordinates
`width`	Pixel width of the map image
`height`	Pixel height of the map image
`format`	Output format of the map image

TABLE 3.3: Web Map Service – GetMap parameters

Parameter	Description
`layers`	List (comma–separated) of map layers
`info_format`	Format of feature info
`i`	i pixel coordinate of the feature
`j`	j pixel coordinate of the feature

TABLE 3.4: Web Map Service – GetFeatureInfo parameters

Table 3.3 lists the mandatory parameters of the `GetMap` operation. This is the operation that actually produces a map image, as visualized in figure 3.1, and is as such the most important operation of a WMS. We will look at a practical example of this operation in section 3.4.2.

`layers` is a list of layers from which the image should be produced. The names of the available layers in a WMS is retrieved through `GetCapabilities`. The `styles` parameter is a list of styles to be applied on geometry when rendering the layers of the map image. It must correspond 1–to–1 with the `layers` list. `SRS` (map projection) and `BBOX` define a bounding box on the layers in which the geometry is rendered out to the map image. `BBOX` syntax is `min_x,min_y,max_x,max_y`. The `SRS` applies to the bounding box itself, and is necessary to give meaning to the coordinates specified for the bounding box. `width`, `height` and `format` apply to the produced map image itself, giving its desired width, height and output format. Supported output formats for each available layer is given in a `GetCapabilities` response.

Optional parameters include the option of producing images with transparent backgrounds (necessary for overlaying several map images) or setting the background to a specific colour.

- `GetFeatureInfo` (optional)

  The optional `GetFeatureInfo` operation allows clients to query the WMS on more information regarding features that it has discovered through a `GetMap` operation. The use case outlined in the standard is a user using a WMS client to click on an area

of a retrieved map image, whereupon a `GetFeatureInfo` operation is performed with the appropriate `i` and `j` parameters. The client may then present the user with more information on the specific feature. The content and formatting of this information is left to implementers.

A WMS announces its support for the `GetFeatureInfo` operation by setting the `queryable` option for a layer when listing them in a `GetCapabilities` operation. A successful `GetFeatureInfo` request requires support for the operation both from the WMS itself and for the layer that is queried. Optional parameters include setting the (maximum) amount of features to be listed for the chosen pixel coordinates (the graphical representation of features are likely to overlap at some zoom levels).

`GetFeatureInfo` is interesting because it shares usage scenarios with Web Feature Service, which is the topic of section 3.5. WFS offers wider and more complex functionality, so using `GetFeatureInfo` is not necessary if there is also a WFS serving the same geographic data. It is, however, a neat alternative if WFS is not available.

3.4.2 The GetMap operation in practice

In this demonstration, we assume a WMS is hosted on `http://demonstration.web.com/wms`. Through a `GetCapabilities` request, we already know of the layer `measurements` and that the WMS supports outputting in the PNG image format. We now want to retrieve an image of the layer with the following parameters:

- Bound by the box given by coordinates *-120.0, 18.2, -37.0, 42.3* (box defined by lower left corner point at (-120.0, 18.2) and upper right corner point (-37.0, 42.3)) in the spatial reference system ESPG:4326 (the details of this SRS is not important at the time, but required to give meaning to the box coordinates)
- Default styling
- Image width of 700 pixels
- Image height of 300 pixels
- Image format PNG (MIME–type *image/png*)

A WMS must support operations over HTTP GET. In the GET–method, parameters are added to the URL as key–value pairs with "`=`" indicating assignment and using "`&`" as a delimiter. The parameter list is appended (with a leading "`?`") to the URL of the host which will receive the request.

```
http://demonstration.web/wms?
request=GetMap
&service=wms
&version=1.1.1
&layers=measurements
&styles=
&srs=EPSG%3A4326
&bbox=-120.0,18.2,-37.0,42.3
&width=700
&height=300
&format=image%2Fpng
```

Code Listing 3.1: URL for GetMap operation

The URL for HTTP GET–request with our specified parameters is given in code listing 3.1. Note the percent–encoding [28] of some characters and the splitting of the URL into several lines for readability. The `styles` parameter is empty, meaning it will be filled with the WMS server's default value. Styling is discussed further in section 3.6.2.

The request can be made with any client supporting HTTP GET request (for example by navigating to the URL in an internet browser). Making the request corresponds to step 1 of figure 3.1. The response will include a map image constructed to our parameters. Generating this image corresponds to step 2 of the figure, while step 3 would be the client receiving the image as a response. How step 2 is performed is not described by the standard, leaving this process to the server implementers.

If the WMS supports performing operations through HTTP POST, the same result could be achieved by setting the same parameters in an XML file included as the message of the POST–request. A schema for such XML files are defined in the WMS standard.

3.5 Web Feature Service

The Web Feature Service (WFS) standard defines a protocol for exchanging geographic information over HTTP. While WMS generates and transmits images, WFS transfers information on *features* encoded in the *Geographic Markup Language* (GML). GML is an XML–based markup language that is its own OGC standard and is presented briefly in section 3.6.1.

The WFS standard is defined in [29] and [30], along with a discussion paper [31] and a best practices document [32]. The latest version of the standard is [29], which is also an ISO standard as [33].

Parameter	Description
`typeName`	The WFS equivalent of WMS' `layers`; the container for which the request is intended
`featureID`	Supply an ID as this parameter to retrieve this specific feature
`maxFeatures`	The maximal number of desired features in the response
`propertyName`	Retrieve only the desired property from a feature. May also retrieve several by providing a comma–separated list. Equivalent to retrieving a subset of a row instead of the entire row from a database.
`filter`	Provide a Filter Encoding (see 3.6.3) encoded filter that restricts the returned set of features

TABLE 3.5: Web Feature Service – GetFeature parameters

WFS is built to support transactions to enable users simultaneously accessing the service (then sometimes called *WFS transactional* (WFS–T)). This adds a general layer of complexity, along with several operations. Editing the data is not currently a functional requirement of our system, so these capabilities will not be discussed here.

3.5.1 WFS Operations

The WFS *protocol* itself is structured similarly to WMS. For this reason, we will not repeat the *web protocol* aspects shared between WMS and WFS. Instead we go directly to the operations relevant in constructing our system.

- `GetCapabilities`

  This operation is equivalent to the `GetCapabilities` operation of WMS. It serves the same purpose and is structured similarly.

- `GetFeature`

  The `GetFeature` operation is WFS' central operation: it produces a feature or set of features according to the input parameters. An interesting set of parameters is presented in Table 3.5, note that familiar, mandatory parameters (`service, version, request` etc.) are omitted. The default response format for the `GetFeature` operation is GML; see Section 3.6.1 for an introduction to the format and Section A.2 for an example.

3.6 Other relevant standards

The standards presented below all see use in the constructed software, but are not essential aside from when discussing implementation details (the topic of Chapter 7). Their presentation here will be very brief, and does not reflect their general size or importance (GML, especially, is a very substantial and fundamental standard).

3.6.1 Geography Markup Language and GeoJSON

Geography Markup Language (GML) is an XML–based encoding for geographic features. It is the default encoding used for OWSs such as WFS. GML documents may be validated against the official GML schemas hosted by the OGC. The GML standard is defined in several documents corresponding to versions, the latest being [34], along with several discussion papers and best practice documents.

GML is in use in our prototype application. We have however opted to use GeoJSON [35] as the standard internal encoding in the client application. GeoJSON is equivalent to GML in purpose, but is encoded in the JSON format [36]. An example GML file is provided in Section A.2, and a GeoJSON encoding of the same feature is provided in Section A.3.

3.6.2 Styled Layer Descriptor

Styled Layer Descriptor is an OGC standard that enhances WMS with additional options for customising the style (presentation) of layers served from a WMS. Styles are defined in XML markup that follow a schema set by the standard. The standard is defined in [37, 38].

Styles are used to alter the look of features as they appear from a WMS request. For example, the feature representation could change colour based on the value of some attribute it contains. The WMS must provide a default style, requested by supplying an empty `STYLE` parameter in a WMS URL, but can also be provided by the client using this parameter. An example SLD file is provided in Section A.5.

3.6.3 Filter encoding

The OGC filter encoding is an application–neutral XML encoding for *filtering* of features. A valid OGC filter can be supplied in a WFS request as the `FILTER` parameter, replacing

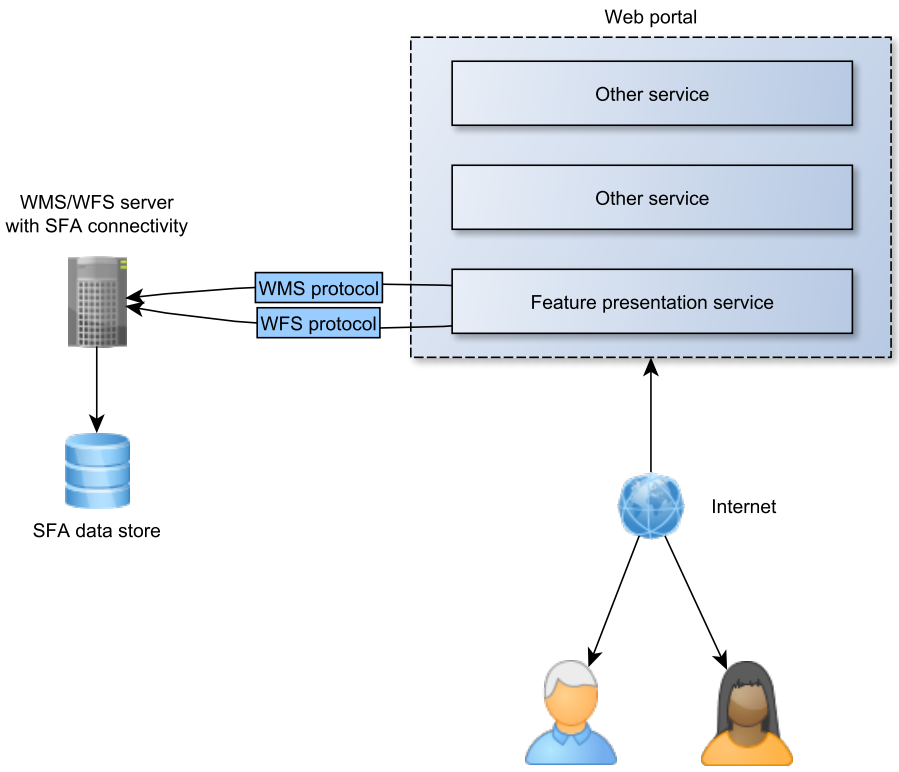


FIGURE 3.2: Conceptual model – OGC standards aware

the `BBOX` parameter (the two are mutually exclusive due to functional overlap). Filtering results on WFS–server side is beneficial as it lessens the network load and client workload.

A filter may contain various rules, such as "attribute `temperature` must be between 10 and 20", that can be combined or altered by the logical operators `AND`, `OR` and `NOT`. Filtering may also be performed on geographic attributes, so filtering can fully replace the simpler `BBOX` parameter in WFS.

An example Filter Encoding file is provided in Section A.4.

3.7 Conceptual overview model with appropriate standards

Figure 3.2 is the conceptual model of the system to be constructed, now updated to indicate which standards will be used.

It is clear that the geographic data store must be SFA compliant, and will probably be a technology implementing the SQL option. The geographic data accessor must function as an SFA–aware client towards the data store, using the methods the SFA standard defines to present the geographic data. It will present this data to clients (thus functioning also as

a server) by being a Web Map– and Web Feature Service (they may be hosted on separate servers, but are represented as one entity in the model). In short, this component will be – as we imagined – an intermediate "translator" of client requests to spatial database queries; its languages will be SFA methods and WMS/WFS operations.

The presentation service, accessible to regular users over the internet, will use the WFS and WMS protocols to communicate with clients. The primary data encoding of WFS traffic will be GeoJSON, but other formats (including GML) are still relevant due to FR9 (data export). In this model we have renamed the service itself to "*feature* presentation service" to align ourselves with the terms of SFA. We will still refer to this component as the (front–end) client application or geographic data presentation service elsewhere.

4 Software stack

The software components and relationships between them used in constructing a layer–structured application is referred to as a *software stack*. In the previous chapters, such a stack was conceptually outlined in a series of models. In this chapter, we again update this model by selecting implementing technologies for all entities in the conceptual model. The result is a complete, real software stack on which the application will be developed.

We begin by discussing and determining a set of criteria on which to evaluate software technologies. With these criteria in mind, the selection processes are presented in a bottom–to–top order (beginning with the data store). The chosen technology is introduced in some detail, and relevant concepts may also be discussed.

4.1 Selection criteria

The problem statement guides the criteria on which potential software technologies are evaluated. Four main criteria are presented below, in a partially prioritised list (the first criterion being the most important).

- Potential in fulfilling requirements

  Requirements for each component was detailed in chapter 2. Any selected software must be able to fulfil their appropriate requirements.

- Standard compliance

  As described in Chapter 2, we aim to achieve maintainability and extensibility by utilising standardised data encoding and communication. This makes the standard compliance of involved software technologies a very important criterion. The standards to implement or support were introduced in Chapter 3 and summarised

in Section 3.7. Cerified compliance (see Section 3.1.1) is valued over claimed or incomplete support.

- Open source

  It is part of the overall goal of this thesis that all software involved should be open source. The most important benefits of choosing open source software over proprietary solutions are reported to be avoiding vendor lock–in, cost and quality [39]. Avoiding lock–in is especially important to this software system to maintain modularity of components beyond just the initial deployment.

  An open source software alternative will be valued over any proprietary software unless it lacks in other criteria. We will not pay much attention to the complexities of *open source* versus *free* licences [40], concluding simply that "open source" status is sufficient for our needs.

- Perceived quality

  Software quality is hard to define or measure. Our selection will be based on the *perceived* quality of a software technology – developing with all alternatives and evaluating the result would be too time–consuming. Quality indicators are widespread use, quality and availability of documentation, test coverage, standard compliance, etc.

- Future support

  As with quality, evaluating how well a software technology will be supported after initial deployment is hard. Wide adoption and active development are indicators that support is likely to persist.

Each following section corresponds to a layer in the software stack/entity in the conceptual model. For each layer, a selection of candidate software technologies are briefly examined. This will not be an exhaustive list; only technologies that were seriously considered are included. Candidate technologies were discovered through the OGC's *implementing products* web catalogue [41] and web search engines.

Our selection is then presented along with an argument for its selection. More detail on any software technology is also provided in upcoming chapters when discussing its role in the implementation the system.

4.2 Geographic data store

The requirements for the geographic data store, given in Section 2.3.1 were as follows:

- Quality and efficiency in regular data store operations
- Compliant with relevant standards in geographic data encoding
- Support reading text files for loading the legacy data set

The first two overlap with our selection criteria, while the third is a functional requirement that is necessary due to the format in which our geographic data is initially available in. The standard to support was determined to be SFA, SQL Option.

4.2.1 Alternatives

- **PostGIS**

  PostGIS [42] is an extension to the PostgreSQL SQL database management system [43]. It *spatially enables* PostgreSQL by providing additional data types and functions. PostGIS implements support for *all* objects and functions of SFA, along with own additions [44]. Both PostgreSQL and PostGIS are open source projects, and have long and proven track records for quality [45][46]. PostgreSQL notably includes native support to read data from text files [47].

  PostGIS is open source, fully standard compliant, builds on a widely adopted database system and is still under active development.

- **MySQL**

  MySQL is, like PostgreSQL, a powerful and widely used open source database management system [48]. It has native support for spatial operations based on SFA, but it does *not* have full standard compliance as it implements only a subset of the SFA classes and methods [49].

  MySQL is open source, partly compliant to standards, very widely used, but extended standard compliance in the future can not be guaranteed.

- **Others**

  Many of the larger, proprietary database management system implement some level of geospatial support guided by the SFA standard. Examples include Microsoft's SQL Server [50] and Oracle's Oracle Spatial and Graph [51]. These would be valid options if it were not for our open source–criterion.

4.2.2 Selection: PostGIS

Both MySQL and PostGIS are likely to be solid choices, but we decided on PostGIS due to the fact that it fully implements SFA. We may not initially require all the classes

or methods of SFA, but using PostGIS guarantees that we can do anything within the limitations of the standard. Since PostGIS is a dedicated extension, it is also likely to be built specifically to accommodate typical geospatial database operations, while the spatial support of MySQL is just one among a vast number of functionalities.

4.3 Geographic data accessor

The requirements for the geographic data accessor, given in Section 2.3.2 are:

- Communicates with our data store
- Provides standardised access methods for the underlying data store
- Configurable for adapting to our data set
- Adequate performance – tasks may be computationally demanding

The geographic data accessor must be able to connect to our selected SFA compliant geographic data store. It must also be able to publish the data store tables as layers through the employed OWSs. Performance is a non–functional requirement that may be considered under our quality–criterion.

4.3.1 Alternatives

- **MapServer**

  MapServer [52] is a data rendering engine that can publish data from SFA sources including PostGIS and MySQL through WMS and WFS. However, WFS is notably only partly implemented [53] and all employed OGC standards are on the *implementing* level of support.

  MapServer is open source (custom, but very permissive licence) licence, partly standard compliant and under active development.

- **GeoServer**

  GeoServer [54] is a Java–based server application that implements several OGC web standards. The application is open source and developed as part of the OpenGEO initiative. GeoServer supports providing WFS and WMS access to data from a selection of SFA–compliant data stores (including PostGIS natively, and MySQL through an unsupported extension [55]). GeoServer is *certified compliant* (see Section 3.1.1) with both the WMS and WFS standards [56].

GeoServer is open source (GPL2 licence), fully standard compliant and in active development (with several new versions being released during work on this thesis).

4.3.2 Selection: GeoServer

GeoServer perfectly matches our requirements and fully supports our desired standards – MapServer can not compete for our purposes. GeoServer appeared so strong that it also influenced our choice of data store, since it does not support MySQL natively.

4.4 Client application

A client application is to be developed during the thesis according to requirements defined in Chapter 2. A central functional requirement to provide a map widget that displays features on a world map and allows interactivity such as zooming/panning and interactivity with the rest of the application. The data must be retrieved from the geographic data accessor through OWSs.

We examined several software technologies that provide such functionality and considered them under the same criteria as for other components of the software stack.

4.4.1 Map widget alternatives

- **Leaflet**

  Leaflet [57] is a JavaScript library for displaying interactive maps in a web browser. It can read from WMS layers, but this is its current extent of OGC standard–relevant features. It has a very tidy and well documented API and is under active development. Leaflet is open source (custom, permissive licence).

- **OpenLayers**

  OpenLayers [58] is a JavaScript library for displaying interactive maps in a web browser. It can display maps from both WMS and WFS sources. OpenLayers' functionality is wide in scope, going beyond functionality that is directly related to map widget operations. Some of these are interesting to us, for example functionality for constructing Filter Encoding filter strings.

  OpenLayers is open source (FreeBSD licence) and under development, with a new major release focusing on API/documentation clean-up and HTML5 capabilities on the horizon.

- **Others**

  Both OpenLayers and Leaflet are built on JavaScript and HTML/CSS. Other technologies also exist for building rich web applications, such as Adobe's Flash and Microsoft's Silverlight. Due to the availability of JavaScript solutions, no other technologies were seriously considered. This decision follows from the standard compliance criterion: HTML and CSS are W3C managed standards, as are the interfaces for communication between JavaScript and the browser. Not relying on browser plugins also increases the general security and availability of web applications [59].

4.4.2 Selection: OpenLayers

OpenLayers is the heavier an more mature option of the two alternatives. Leaflet has a more initially appealing design and documentation, but the standard compliance and wide GIS–oriented feature set of OpenLayers trumps it for our purposes. OpenLayers is also proven as a Web–GIS presentation component on top of the PostGIS–GeoServer stack [60].

4.4.3 Additional client technologies and libraries

OpenLayers is the most significant library used in the presentation service. However, some additional libraries are also employed and are presented in the list below. These libraries were added to the system during development, as demand became apparent. The technical details of their usage is covered in Chapter 7.

- **jQuery**

  jQuery [61] is a JavaScript library offering a range of utility functionality. It is used throughout the client source code for e.g. DOM manipulation, event handling, looping constructs. jQuery is released with open source under the MIT licence.

- **jQuery UI**

  jQuery UI [62] is a JavaScript and CSS extension of jQuery. It provides GUI elements like buttons, lists and more complex layouts (all called jQuery UI *widgets*) with graphical design and interactivity. The overall design of the client application is structured around jQuery UI widgets. jQuery UI is licensed like the jQuery library.

- **DataTables**

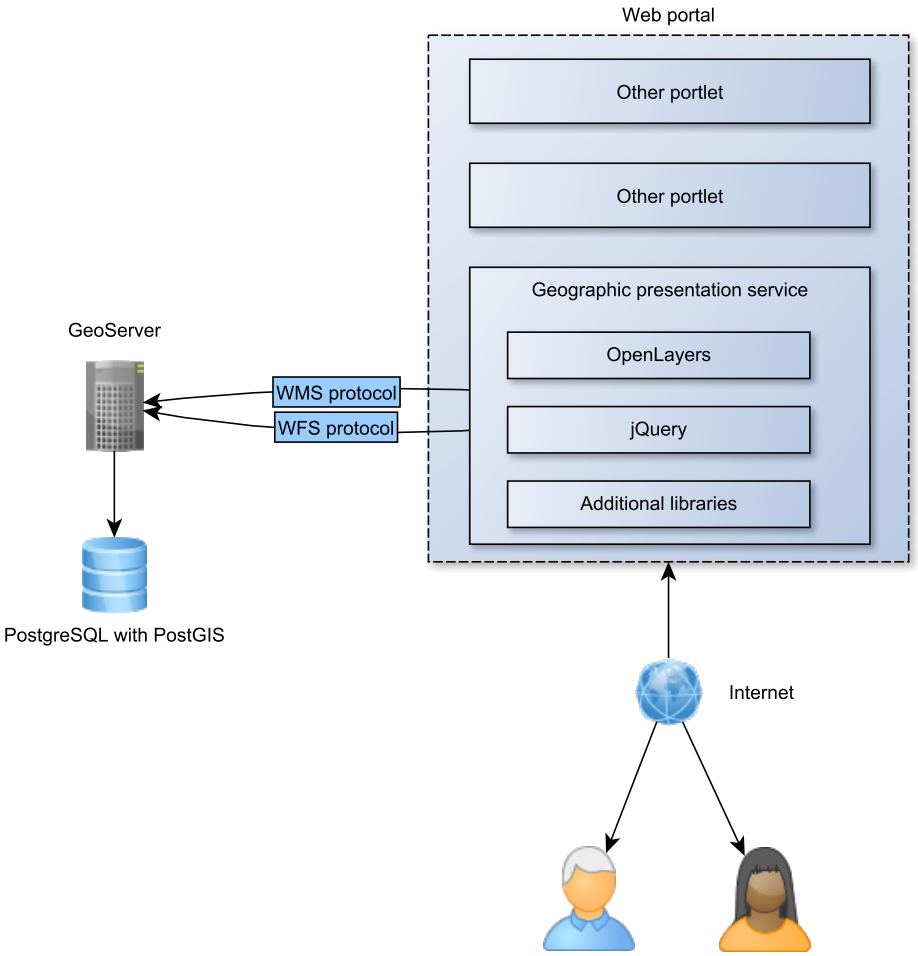


FIGURE 4.1: Conceptual model – technologies

DataTables [63] is not an independent JavaScript library, but a plugin to jQuery. It enhances the presentation and functionality of standard HTML tables. It is used in the application on all tabular data to enable sorting on columns, automatic pagination of large tables and other minor table–related functionality. DataTables is provided under either the GPL2 licence or the BSD licence.

- **HighCharts JS**

  HighCharts JS [64] is JavaScript library for drawing graphs and diagrams. It is used in the client application to render graph plots and provide interactivity for these plots. HighCharts JS is available with open source under the Creative Commons Attribution–NonCommercial 3.0 licence.

4.5 The complete software stack

We already have a model of our conceptual software stack, and it was updated in the previous chapter to reflect our new knowledge on OGC standards. We now update it again by replacing the OGC standards to support with our selected implementing technologies. Figure 4.1 is our updated model. PostgreSQL with PostGIS is the SFA compliant geographic data store. GeoServer is the geographic data accessor, exposing the data from PostGIS through the OWSs WMS and WFS. The geographic presentation client is implemented in HTML/CSS and JavaScript, with OpenLayers providing the map widget and additional JavaScript libraries enhancing functionality and presentation.

5 Data transformation

With an overview of relevant standards and the set of software technologies that make up our stack, we can now describe the implementation of the system constructed to reach the overall thesis goal. The first challenge to handle is the transformation of the data from the ODB system into a standard compliant structure, sub–goal 3. We operate at the lowest level in the software stack; at the *geographic data store* layer.

We begin by describing the original structure and layout of the ODB database. We then outline necessary high-level steps required to make it conform to the SFA data structure. Next we describe how the described steps were implemented using PostGIS. We end the chapter with a presentation of the resulting database structure.

5.1 The ODB database

The ODB database is centred around the `station` table. This table has approximately 750 000 rows. A station row consists of data for latitude/longitude, the name of the vessel, time and date of the measurement, etc. The `station` table has a number of associated parameter tables. Most `station` rows have parameter measurements for temperature and salinity, some for many more. Other recorded parameter values include sulphide and nitrate. A parameter table row is a level–value pair describing the measured value of that parameter at a depth. Each row is associated to a `station` by a foreign key. A `meteo` table is also associated with each station row, giving weather conditions recorded during the measurements. Interesting sets of `station` rows are collected in the `catalog` table.

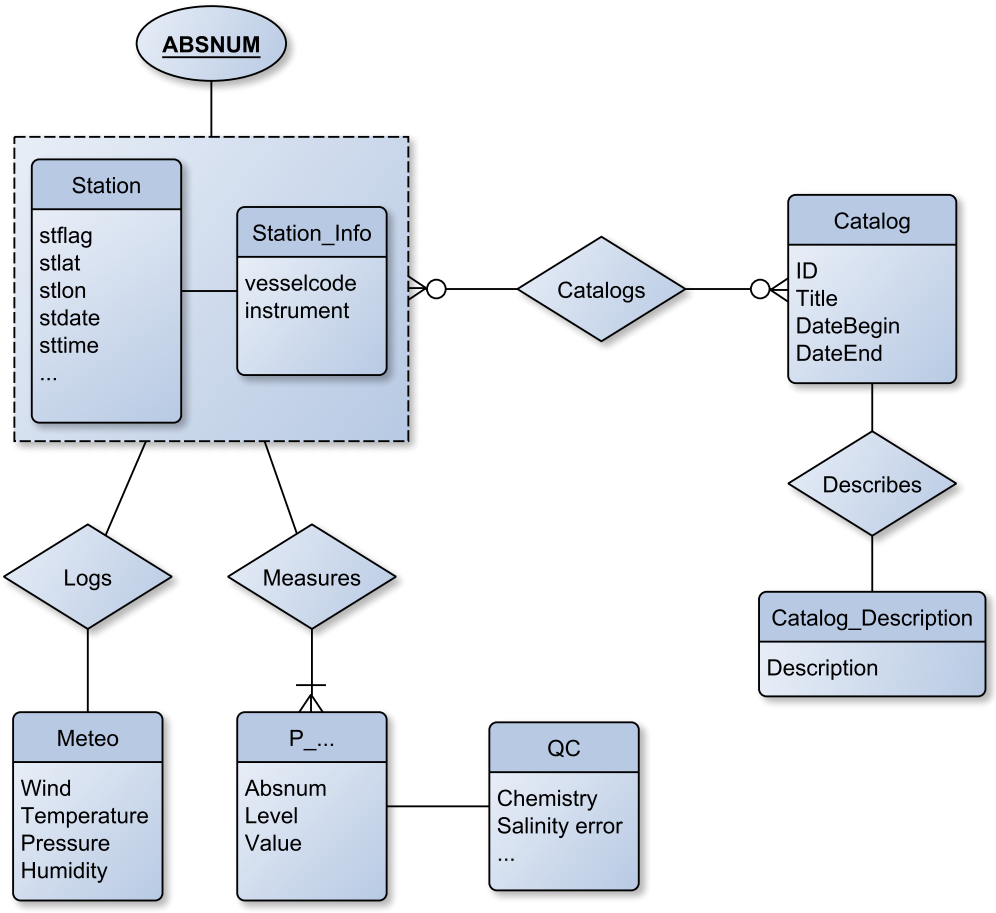


FIGURE 5.1: ER–diagram: the legacy database

5.1.1 ODB database diagram

The ODB database is modelled in Figure 5.1 as a conceptual Entity–Relationship diagram using crow's foot notation. This model is reverse–engineered from the actual database; it was not used in the original design process. Regarding the diagram, please note:

- Some non-essential attributes are omitted for visual clarity.
- Bilateral 1–to–1 relationships are represented as simple lines.
- The non–standard dotted line entity surrounding the `Station` and `Station_Info` entities represents how these two entities are for all intents and purposes the same entity, they have simply been split into two tables in the database.
- The "`P_...`" entity represents parameter tables. These vary in type and amount depending on the chosen database. Examples are `P_Temperature` (a temperature measurement entity) and `P_Salinity` (a salinity measurement).

5.1.2 Development subset

For development purposes, a subset of the complete data set was selected. A set of around 2500 station rows with their associated data were extracted. The `meteo` and `catalog, catalog_description` were not included since they have no relation to the current functional requirements of the system. A representative set for parameter tables (`p_...` tables) was decided to be the `p_temperature` and `p_salinity` tables. Note that while development was done with this limited subset, the client and procedures for other layers in the software stack should also accommodate for the complete data set.

5.1.3 Towards Simple Feature Access-compliance

We wish to transfer and transform this data into a new database and have its structure be compliant with SFA. We performed this task in a sequence of three high–level steps that are outlined below. Note that these steps are not specific to PostGIS, but rely only on SFA functionality.

1. All data inserted into geospatially enabled database system

   The extracted data from the legacy database must be inserted into a database management system that supports spatial data types and operations. The data types of the new columns must be equal to or wider than what the data was originally stored in to avoid loss of precision.

2. Longitude and latitude transformed to geometry fields

   Any geometric data must be stored in a column of a recognised geometry data type and encoded as WKT or WKB. In our case, this applies to the longitude and latitude columns (`stlon` and `stlat`, respectively). The appropriate data type for this pair is the SFA `Point`. This is the only true *transformation* step required on the data itself – the original columns should be removed to avoid the possibility of data corruption caused by redundant storage.

   The SFA SQL Option metadata tables must also be created and updated to reflect the addition of a geometry column.

3. Choose and set appropriate SRS, update geometry columns accordingly

   To be able to meaningfully use the data, a spatial reference system must be chosen and applied in the database management system.

```
CREATE TABLE station
(
  absnum integer,
  stflag smallint,
  stlat numeric(8,5),
  stlon numeric(9,5),
  stdate date,
  sttime time without time zone,
  stsource character varying(12),
  stversion smallint,
  stcountryname character varying(40),
  stvesselname character varying(40),
  stdepthsource integer,
  stlastlevel smallint,
  stdepthgrid smallint,
  stdepthgridmin smallint,
  stdepthgridmax smallint,
)
```

Code Listing 5.1: Creating the Station table

5.2 Implementing the new database structure

The following three sections correspond to the steps described in Section 5.1.3. As real implementations of steps initially described at a higher level of abstraction, they will be specific to our selected database technology (PostGIS). Equivalent operations should be available in all SFA compliant geospatial database management systems.

5.2.1 *Step 1* – Creating the database and importing data in PostGIS

5.2.1.1 Creating the database and tables

A new database is created using the template provided by the PostGIS system. This will automatically configure a new PostgreSQL database with access to PostGIS functions and data types. The mandatory metadata tables of the SFA SQL Option standard (`SPATIAL_REF_SYS` and `GEOMETRY_COLUMNS`) are also automatically constructed and configured.

Next, we replicate the tables of the legacy database in our new database. Our recommendation is to make an exact (or as close as possible) copy of the original table structures – same column names, ordering and data types. Any alteration at this stage will increase the complexity of all following steps in the procedure. Further restructuring of the final database is the topic of Section 10.2.1.

```
1 COPY station FROM '/home/auser/station.dat';
```

Code Listing 5.2: PostgreSQL COPY function

Code Listing 5.1 is an example code listing for (re)creating the `station` table in PostGIS/PostgreSQL. This operation is standard SQL, no spatial extension are employed yet. Data types used are based on documentation from the ODB system. The procedure is repeated for all tables of our working set.

5.2.1.2 Import with `COPY`

With the database and tables set up, we are ready populate it with data from the legacy database. The exact procedures of this step depend on the format of the original data and the capabilities target database.

PostgreSQL provides the `COPY` function for reading text files into tables. Some minor manual adjustments on the text files were necessary: a single-line header giving column names was removed and a superfluous level of tab-indentation present on all data rows was deleted. With these adjustment, the import operation itself is simple. Code Listing 5.2 is an example PostgreSQL code snippet that utilises `COPY` to import the file `/home/auser/station.dat` to the table `station`.

Similar procedures are performed from the other table text dumps. Note that the table to be imported into must already exist and have a matching amount and type of columns (with regards to the text file) for the `COPY` function to work as it does here (without any additional configuration). Any error means no data will be read, so the input file must be consistent.

5.2.2 *Step 2* – Creating and populating geometry columns

We now have a functional, populated database that responds to standard SQL calls. To utilise SFA, however, we must introduce the concept of geometry to the appropriate tables.

As discussed in step 2 of Section 5.1.3, the geometry of our data is contained entirely in the `stlon` and `stlat` columns. They should be replaced by a geometric column that represents the longitude/latitude of a row with the SFA `Point` data type.

The replacement is done in three steps. We utilise PostGIS functions in combination with traditional SQL. The steps are given below with example code, acting on the same `station` table used in previous examples.

```
SELECT ST_AddGeometryColumn(
  'public', 'station', 'myPoint', 4326, 'POINT', 2
);
```

Code Listing 5.3: Adding a geometry column

```
UPDATE station
SET myPoint =
  ST_SetSRID( ST_Point(stlon, stlat), 4326)
```

Code Listing 5.4: Transforming *lat/lon* pairs to geometry encoding

1. Create new geometry column

   We use PostGIS' `ST_AddGeometryColumn` function to add a geometry column to the table. This will also automatically add a record in the `GEOMTRY_COLUMNS` metadata table. Code Listing 5.3 is an example usage, where we name the new column `myPoint`.

   Data in this column will be stored in the `Point` SFA data type, represented as WKB. The arguments of the `ST_AddGeometryColumn` function are, in order:

   - Schema name
   - Table name
   - New geometry column name
   - New geometry column SRID (Id of the SRS in use)
   - New geometry column data type
   - New geometry column dimensionality

   PostGIS provides the function `ST_AsText()` for easy conversion to WKT (and vice–versa for WKT to WKB conversion).

2. Insert longitude/latitude data for all rows

   Through an SQL `UPDATE` call, we insert data into our new column calculated from the longitude/latitude data contained in each row. Note that we also need to set the SRID at this time, which originally was presented as step 3 of the overall process. This is discussed in the next subsection.

3. Remove the old longitude/latitude columns

   No PostGIS functions are required here, as there is no geometry involved. A simple `ALTER TABLE` SQL query is sufficient.

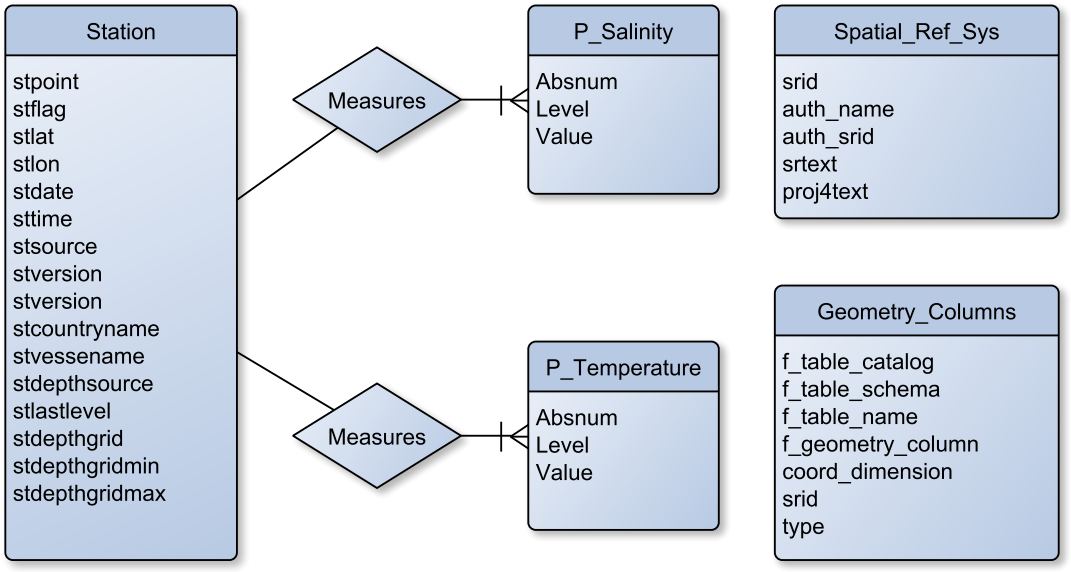


FIGURE 5.2: ER–diagram: The new database structure

5.2.3 *Step 3* – Spatial reference system

We do not know the precise SRS of the legacy data, but it is provided in latitude and longitude. The SRS World Geodetic System 1984 (WGS84) [65] is a global latitude/longitude grid which is widely used (for example by the *Global Positioning System* (GPS)). We assumed this SRS for our data, and this assumption appears correct when the data is overlayed on WGS84 map images (aligned with coastlines, no features on land, etc.).

The WGS84 SRS has SRID 4326 [66]. We declared WGS84 to be the SRS of our system by providing it in the `UPDATE` calls of the previous subsection, where 4326 is given as the SRID parameter. In doing so we have already fulfilled step 3.

5.3 Resulting database structure

Our data is now available in a Simple Feature Access–compliant database management system. This means we can access it with client software that supports querying Simple Feature Access–data. Some example SFA SQL option methods implemented as PostGIS functions are demonstrated in Section A.1. Such operations may now be performed on this database.

Figure 5.2 is an ER diagram of the new database structure. The SFA SQL Option metadata tables `SPATIAL_REF_SYS` and `GEOMETRY_COLUMNS` are included in the diagram, these are automatically managed by PostGIS. The addition of more tables from the legacy

database is trivial, but we decided on this set since it suits our functional requirements (see Section 5.1.2).

Further restructuring is possible and recommended. Topics include naming and database indices (both regular and spatial indices) to improve query speed. Since this is not currently necessary to fulfil the functional requirements, it is deferred to future work and discussed in Section 10.2.1.

6

Data accessor configuration

In the previous chapter, we transformed the data from ODB to a standard compliant structure. In this chapter, we connect the *geographic data accessor* component of our software stack to this data store, and make the geographic data available through the OWS protocols introduced in Chapter 3. Communication between the data store and the data accessor is based on SFA, which was also a topic in Chapter 3.

We begin by describing in more detail the selected data accessor software – GeoServer – and how it connects to PostGIS. We then present how both geographic and non–geographic data is presented as *layers* in WMS and WFS services. We also present how interpolated *contour plots* are implemented in GeoServer, as part of fulfilling `FR8` of Table 2.2. We end with a representation of the web–accessible layers as tables of an ER diagram.

6.1 GeoServer overview and organisation

The central construct in GeoServer is the *layer*. GeoServer's layers are based on and correspond to the layer terminology of WMS/WFS. Once data is published as a layer, it is available for access to WFS and WMS, and also in a number of formats directly from the GeoServer web administration interface. Metadata, access permissions and more can be configured per–layer.

A layer is added from a *store*. A number of vector and raster sources may serve as stores. Supported vector stores include PostGIS databases, shapefiles [67], or another WFS service. Raster sources may be added from a number of file formats.

GeoServer supports SLD for styling of WMS data and refer to them by the term *styles*. A style can be added directly in the web administration interface or uploaded from an XML file.

Styles and stores (and under them, layers) are organised into workspaces. This allows several projects to be hosted from a single GeoServer instance. Some settings may also be configured per–workspace.

6.1.1 GeoServer and PostGIS

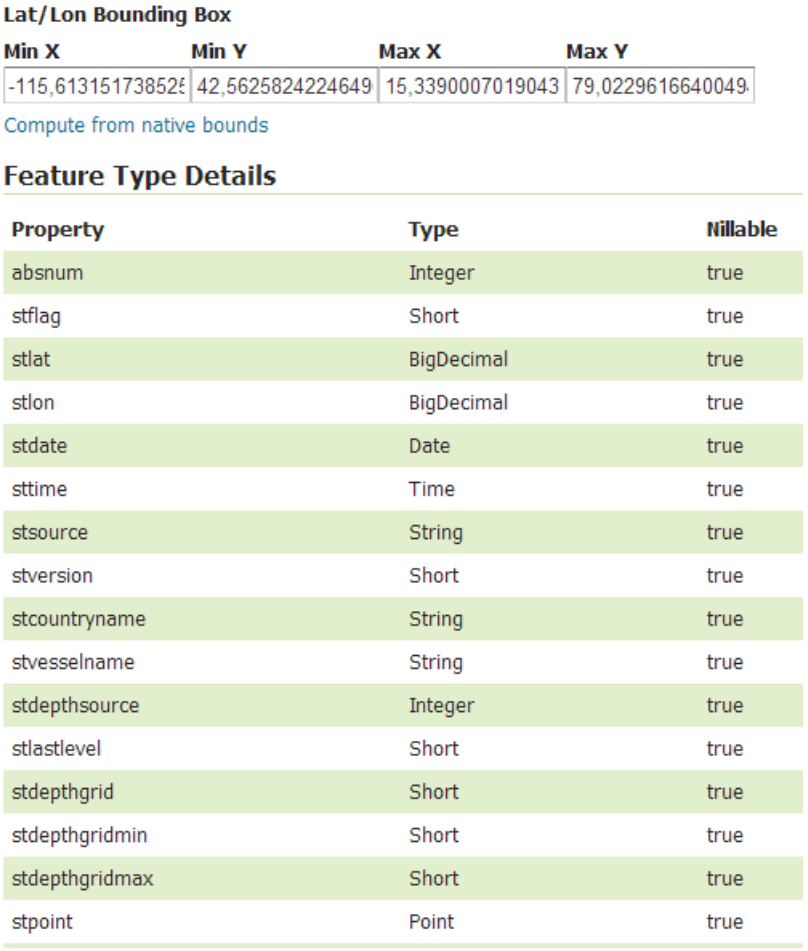
GeoServer natively supports using PostGIS as a vector store. Adding the database constructed in Chapter 5 is done by providing the database URL and credentials in the GeoServer web administration interface. Other kinds of databases would require extensions to GeoServer, and possibly a more complex configuration process.

6.2 Offering geographic data through WMS and WFS

A core task for the geographic data accessor component is to expose the `station` table from the database through WFS and WMS. This table was identified in chapter 5 to be the only truly *geographic* table of the ODB database, and was transformed to contain a SFA `POINT` column.

A database table with a geographic column can be directly published as layer in GeoServer. After connection to a database has been established, such tables will be detected and presented for publishing in the GeoServer web administration interface. The data types of all columns (including the geographic column), SRS, bounding box and more are automatically recognized. Figure 6.1 is a screenshot of the `station` table being configured in the administration interface. Note that `stpoint` has been recognized as being of the SFA `POINT` data type. The administration interface also allows for customization such as setting a default SLD representation for WMS requests (the SLD used for the `station` layer is included in Section A.5), layer metadata, access restrictions etc.

Once a layer is configured and published, it is available for access through WFS and WMS. The GeoServer process of going from a SFA database table to a WFS/WMS accessible layer was in our experience remarkably simple and robust.



Lat/Lon Bounding Box

Min X	Min Y	Max X	Max Y
-115,613151738525	42,5625824224649	15,3390007019043	79,0229616640049

Compute from native bounds

Feature Type Details

Property	Type	Nillable
absnum	Integer	true
stflag	Short	true
stlat	BigDecimal	true
stlon	BigDecimal	true
stdate	Date	true
sttime	Time	true
stsource	String	true
stversion	Short	true
stcountryname	String	true
stvesselname	String	true
stdepthsource	Integer	true
stlastlevel	Short	true
stdepthgrid	Short	true
stdepthgridmin	Short	true
stdepthgridmax	Short	true
stpoint	Point	true

FIGURE 6.1: Configuring a layer in GeoServer

6.3 Offering non–geographic data

Creating a layer in GeoServer based on a database table requires that the table contains a column of a `Geometry` data type. What then, if you wish to access some non–geographic table from the same database? Such use cases appear in our system, for example if we wish to plot temperature data from the temperature parameter table. Recall that these tables contain no geographic information in themselves, only traditional data type columns, and are associated to the geographic data by the `absnum` foreign key.

A solution to this problem would be to implement a separate server application that accesses the database and then exposes its functionality as non–OWS web services. This, however, greatly increases the overall complexity of the system and adds a non–standard component to an otherwise standard compliant system. In our context, it would move us towards some of the shortcomings of the ODB system. Fortunately, other solutions exist that leverage the WFS protocol and special GeoServer capabilities. We now present our solution.

```
1 SELECT station.absnum, station.stpoint, station_p_salinity.
    level, station_p_salinity.value, station_p_salinity.flag
2 FROM floats
3 NATURAL JOIN station_p_salinity
4 WHERE station.absnum = %n%
```

CODE LISTING 6.1: GeoServer SQL View with parameter substitution

6.3.1 GeoServer SQL views

GeoServer allows the creation of layers from *SQL Views*. Views are a common feature in SQL databases that essentially create new tables from some query to existing tables. These view tables are temporary; they are created when a query is made towards it.

The views of GeoServer has an additional feature: variable substitution. Parts of the view query can be set to be replaced by a user–supplied value. This value is supplied as the WFS vendor parameter `VIEWPARAMETERS`.

We use these features in combination to provide a layer where parameter values (non–geographic) are joined with their geographic data. The specific feature whose parameters should be retrieved is identified by variable substitution on the `absnum` value. Code Listing 6.1 is our implementation. Note the "`%n%`" on line 4. This is GeoServer's syntax indicating that `n` is provided in the `VIEWPARAMETERS` parameter of a WFS request towards this layer.

The result is that we move one step away from the standard in client–data accessor communication, utilising a WFS vendor parameter that may only be implemented by GeoServer. Information on how to access data in such layers should be included in layer metadata. This solution is still much closer to standard compliance (recall that WFS allows vendor parameters; they usage may just not be well known) than implementing a separate server component to handle non–geographic data. The response to a request which does include the `VIEWPARAMS` parameter is also a completely valid feature representation; it can be treated by a system just like responses that originate from regular layers.

An alternative solution would be to make these views into permanent tables in the database. It is a poor solution since it requires redundant storage (copying the `stpoint` column to parameter tables), which can result in data corruption and increases required storage size. For this reason it also does not scale well, since the `stpoint` column must be added to all parameter tables that are added in the database, with every table significantly increasing redundancy.

```
1 ^[\d]+$
```

Code Listing 6.2: GeoServer SQL View Regular Expression validation pattern

A note must be made on the security aspect of out selected solution. Directly involving user–supplied data in an SQL query opens the application to SQL injection attacks [68]. This type of attack is consistently considered *the* most dangerous and common web application vulnerability [69, 70]. Fortunately, GeoServer offers protection by allowing validation of the supplied data via regular expressions. In our case, `n` should be an integer: the pattern in Code Listing 6.2 allows only integer–representing strings and should prevent an injection vulnerability.

6.4 Contour plots

FR8 states that the system should provide functionality for computing and displaying *contour plots*. Contour plots are interpolated from a data set, and may be presented as raster images. This may involve heavy calculations, so it is an advantage if the operation can be performed on the server rather than on clients. GeoServer implements a concept called *rendering transformations* that can is used to achieve this functionality in our system. We now describe rendering transformations in general, and then our use of it to implement contour plots.

6.4.1 Rendering transformations

GeoServer rendering transformations add the possibilities of server–side transformation of data. A set of *raster–to–vector* and *vector–to-raster* rendering transformations are available, and their usage is documented in [71].

The transformations operate on the layers of a GeoServer instance through SLD files. This means that a rendering transformation can be requested as the `style` parameter of a WMS request – no extra work is required on the requester's side. Rendering transformations are a powerful tool because they allow the transformations to take place in GeoServer's rendering pipeline, rather than on the client's side after the original data has been transmitted.

Rendering transformations are internally implemented as a *Web Processing Service* (WPS) [72]. This is another OWS like WFS and WMS also are (see Section 3.3) However, while WFS and WMS provide access to stored data, a WPS *produces* data through calculations. Due to being implemented as a WPS, a GeoServer instance may also add

Format	Description	Support
`GML2`	Geographic Markup Language, version 2	Native
`GML3`	Geographic Markup Language, version 3	Native
`CSV`	Comma Separated Values text file	Native
`GeoJSON`	JavaScript Object Notation, GML vocabulary	Native
`Shapefile`	Shapefile file format	Native
`XLS`	Microsoft Excel 2003	Plugin
`XLSX`	Microsoft Excel 2007	Plugin

TABLE 6.1: GeoServer WFS response formats

WPS as one of the public services it offers. This allows access to transformations without using SLDs. This is not a relevant use case for our application, so we will not discuss this functionality, or WPS in general, any further.

Code Listing B.1, included as Appendix B, is an example vector–to–raster SLD. It takes a vector layer, extracts a `value` parameter of each vector and generates a raster image. The image is a Barnes surface interpolation [73] (one of the GeoServer's rendering transformations) based on the parameter `value` (lines 19–22). General parameters for the interpolation, such as the number of passes, are set in the `Transformation` tag (lines 14– 66). The `RasterSymbolizer` tag contains key–value pairs for defining the colours used in the raster image, encoded in standard HTML hexadecimal fashion (lines 68–79) .

Applying this rendering transformation to our layer – the final step of producing a contour plot – is simple: this SLD style is set as the default for the layer produces by the SQL view. When requested over WMS by a client, a surface interpolation is calculated and the resulting image is sent as a response. Images are produced for each WMS request (though a cached version may be returned if an identical request has been handled before), meaning the layer allows for multiple levels of detail for various levels of zoom and pan.

6.5 Exporting data

By default, GeoServer may respond to to WFS requests in a number of formats. This functionality is used in the geographic data presentation service to implement FR9. Exporting to the Microsoft Excel spreadsheet format is added as a feature by installing the officially supported (from GeoServer's side) Excel export plugin [74].

GeoServer's supported WFS response formats are listen in table 6.1. This table corresponds to the available data export formats in the front end application.

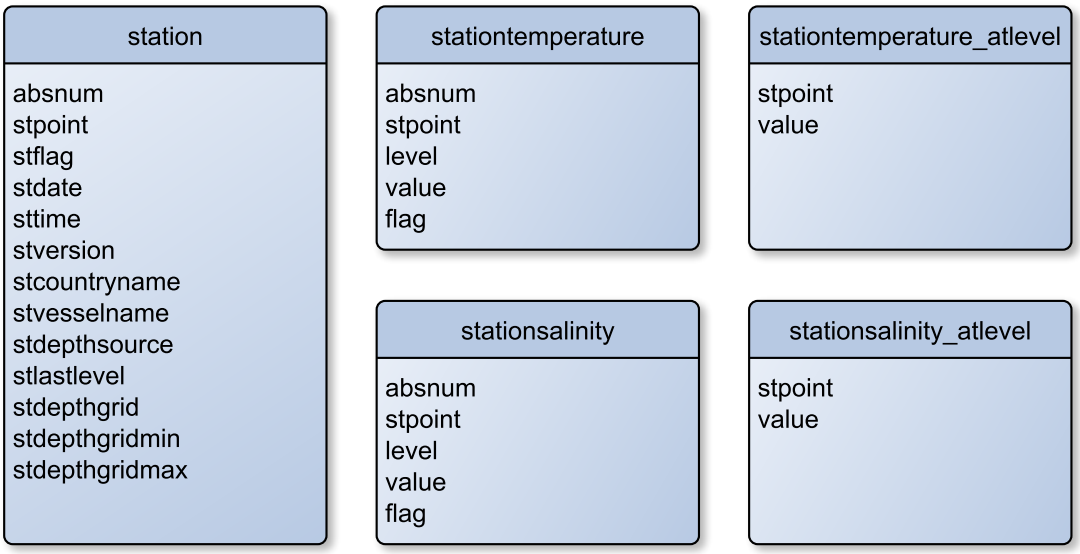


FIGURE 6.2: ER diagram: OWS exposed layers

6.6 The resulting "database"

Figure 6.2 is an ER diagram of the database as it now appears exposed to web services. The `station` entity is similar to that of the original database, except for the addition of the `stpoint` column that holds the SFA geographic information. The parameter entities (`stationtemperature, stationsalinity`) are also similar to the original parameter entities, but now being exposed together with their geographic data through GeoServer SQL views. The WFS standard offers the `propertyName` parameter for the `GetFeature` operation to filter out which properties (here corresponding to columns) of a layer that should be transmitted. This should be used by a client to reduce bandwidth usage if the geography of a parameter is already known. The `..._atlevel` entities have no equivalent in the original database, but are essentially views of parameter tables specialised for GeoServer rendering transformations.

Notice the lack of relationships: all entities are now independent, and their relationships that were explicit in the database through the `absnum` key are now implicit. While some relationship information is lost because of this, it also means that the entities may be independently included as layers through WFS/WMS. This makes them more suited for use outside of applications designed specifically for them. The (now implicit) connection between `station` and parameter layers should be declared in the layer metadata; a request to a parameter layer with the `ViewParams` parameter set to `n:`x will retrieve parameter entries for the station identified by `absnum` x.

7 Client design and implementation

This chapter is concerned with the development of a front–end client that allows a user to view and analyse the data that has been prepared and published in Chapters 5 and 6. The application was developed to fulfil the requirements described in 2.3.3.

We begin by describing the development methodology and environment. The application design and layout is then presented in a non–technical manner. Before looking closer at implementational details, we introduce a number of JavaScript design patterns and practices. On this background, we then present the components that make up the application. We finish with an overview of the employed testing framework and current test set.

The developed application is demonstrated in interaction with the other components of the software stack in Chapter 8.

7.1 Development methodology and environment

Development was carried out following agile development principles inspired especially by [75], albeit adapted to single–person team. The process of requirements analysis and implementation was highly iterative and exploratory with a focus on early delivery of working functionality. The choice of methodology was motivated by the feasibility research question; the existence of suitable tools was not guaranteed at the start of the thesis work. When an initial functional requirement was implemented to a working state, new ones were determined based on the original ODB functionality. We found this development methodology well suited to the task.

Development was largely done in the Eclipse IDE [76] (with the Liferay IDE [77] plugins for portlet management, Aptana Studio [78] plugin for additional JavaScript development

functionality) and with the Google Chrome Developer Tools [79]. The application is hosted as a portlet[1] on a Liferay Portal [82] server. The Liferay Portal server automatically deploys files on change, so no additional infrastructure was required for continuous integration–style development. Version control for source code was performed using Git [83].

The full system software stack (including generic hosting servers that will not be discussed further), with version numbers, is as follows:

- PostGIS 2.0 on PostgreSQL 9.2
- GeoServer 2.2
- Liferay Portal 6.1 Server on Apache Tomcat 7
- OpenLayers 2.13, jQuery 1.9.0, jQuery UI 1.10.3, DataTables 1.9.0, HighCharts 3.0, qUnit 1.11.0

7.2 Front–end development challenges

Before introducing the client application, we address some challenges that occur in all web front–end development.

7.2.1 Browser compatibility

JavaScript has historically been, and is still, plagued with compatibility issues across even the most popular browsers [84]. We will not retell the language and implementations' troubled history here, but we will give a simple statement on the ambitions of browser support for the client application.

JavaScript development and compatibility is not the main topic of the thesis, so we have limited ourselves to targeting the more recent iterations of the most popular desktop browsers. We do, however, attempt not to use any features that would lock out some browsers unless there are good reasons for doing so. Our selection of libraries, listed in Section 4.4.3 also reflects this. jQuery especially offers functionality that handles common compatibility issues such as event handling.

[1] The client was initially imagined to take advantage of the portlet specification [80, 81] for additional server–side capabilities, but this was gradually scrapped as we discovered that our requirements could be met client–side with JavaScript. The application was still developed in a portlet context since this is likely to be how it will be hosted at NERSC.

7.2.2 Same Origin Policy

The *Same Origin Policy* (SOP) is a concept enforced by web browsers to increase the security of cross–site JavaScript DOM manipulation [85]. It is also known to be a challenge for legitimate cross–site operations in JavaScript web applications [86]; as it did for our system.

The geographic data store and the client are independent components, so they may be hosted on different locations that do not pass the SOP check. During development, they were mapped to the URL `http://localhost:8080` and `http://localhost:8081`, respectively, meaning they are not considered *same origin* by most web browsers (Internet Explorer being the exception). This was an issue for WFS operations performed through the OpenLayers library. This is a known challenge with OpenLayers/WFS [87], and we applied their suggested proxy solution during development. In production, these SOP challenges must be addressed by the network administrators responsible for hosting the system.

7.3 JavaScript concepts, techniques and design patterns

In this section, we describe JavaScript concepts, techniques and design patterns are essential to the structuring of the client application. The first topic is the handling of namespaces, and we next introduce and discuss a selection of design patterns.

7.3.1 Namespace management

The entire JavaScript client application is contained within a *namespace object*. The purpose of this object is to minimize the namespace "footprint" of the application, thus avoiding pollution of the global namespace. This is recommended practice [88], and especially important in the portlet context as it means we have little knowledge of the environment in which the application may be run (increasing the likelihood of an identifier collision in the parent window).

The name of the namespace object is not important, but should reflect the name of the application and be highly unlikely to collide with other identifiers in the global namespace. In this text, we will refer to it simply as "myNamespace". The object may be omitted from models and descriptions when not relevant: it is an implementational detail that is not very relevant to the design on a conceptual level.

```
1 var myNamespace = myNamespace || {};
2
3 myNamespace.module = (function() {
4   // module definition
```

Code Listing 7.1: Namespace management

Code listing 7.1 is an example of how the namespace object is used in the application. The components attached to the namespace object in our application are modules. Modules are the topic of Section 7.3.2.

Line 1 is the namespace initialization. The check for definition on is in place to avoid reliance on a specific execution order when an application consist of several files. If the namespace object is already defined, no changes happen. If not, due to `undefined` being a falsy value (evaluates to `false` in a boolean expression) in JavaScript, the namespace object is created using the object literal notation. A module is then attached in line 3.

7.3.2 The *Module* and *Revealing Module* design patterns

JavaScript does not have an explicit concept of access modifiers. This is in contrast with programming languages like Java or C♯ that have e.g. `private` and `public` keywords available for class and variable declarations. JavaScript also has untraditional scoping rules [89] that can make it challenging to manage scope and variable access. These normally are useful concepts for encapsulating functionality by allowing a strict declaration of the publicly accessible interface of a code component.

Fortunately, despite these (arguably) shortcomings, JavaScript is also very flexible. Constructs that allow access modifiers and predictable variable scope are expressible. We will look at a particular set of constructs: named solutions to well–known challenges that can as such be regarded as design patterns [90].

These design patterns are built on the concept of *function closure*. A function defining a closure means that the environment in which the function was invoked is stored [91]. Closures appear in JavaScript through the `scope` property of functions as defined in the ECMAScript standard [92] (the language specification of which JavaScript is an implementation).

7.3.3 Module

The *Module* design pattern [93] emulates the OOP class concept by defining objects called *modules* that can have both private and public members. Code listing 7.2 defines

```
var module = (function() {

  // private variable
  var i = 0;

  // public interface
  return {
    increment : function() { i++; },
    decrement : function() { i--; },
    getCounter : function() { return i; }
  };

}());

// usage examples
// prints 0
console.log(module.getCounter());

module.increment();
module.increment();
// prints 2
console.log(module.getCounter());

// prints error; undefined
console.log(module.i);

var module2 = module;
// prints 2 (not 0) - module is a singleton
console.log(module2.getCounter());
```

Code Listing 7.2: Module design pattern

a simple module according to the Module design pattern and shows some examples of its usage.

Because of function scope, functions and variables defined inside the module function are generally not accessible outside of the function definition. They may, however, be accessed by other inner functions or objects inside the function, such as the one that is returned in the code listing. This object itself is accessible to outside code, as it is bound to the `module` variable. The inner functions and variables of the module remain available to use by the returned object due to closure, but are never directly exposed to the rest of the world.

The result is an object which resembles a traditional OOP class, and the instance functions like an instantiation of the traditional OOP Singleton design pattern [94]. With some modifications to the pattern, a module may also support multiple instantiations. We do not describe this modification here, since it is not used in the client application.

```
var module = (function() {

  // private variable
  var i = 0;

  // private functions
  function inc() { i++; }
  function dec() { i--; }
  function getCounter() { return i; }

  // public interface
  return {
    increment : inc,
    getCounter : getCounter
  };

}());
```

Code Listing 7.3: Revealing Module design pattern

7.3.4 Revealing Module

It's cumbersome to describe the entire public interface of a module using the Object Literal Notation. This shortcoming is addressed in a modification to the module design pattern called the *Revealing Module* [95, 96]. Here, all variables and functions of a module are initially declared as private. The returned object then contains only references to the functions intended to be public. The result is a more friendly syntax and a clearer boundary between private/public members. The public part of a module no longer grows increasingly cumbersome to write as the amount and/or size of public functions increases.

Code Listing 7.3 is an example where the module from Code Listing 7.2 is structured according to the Revealing Module pattern. All functions are now initially declared private, and then some are exposed in the public interface. In the Code Listing, the `dec()` function will not be accessible outside the module. Notice also that this design patterns allows using aliases for functions (the function `inc()` is exposed with the alias `increment()`).

7.3.5 Module dependency management

JavaScript does not provide or enforce any specific technique for declaring the dependencies of a component (unlike, for example, the `import` statement of Java). Any piece of code may instead access any object or function from the global namespace (a concept touched on in Section 7.3.1). This is typically how external libraries are included and

```
var module = (function(someMathLibrary) {

  // private variable
  var i = 0;

  // private functions
  function inc() { i++; }
  function dec() { i--; }
  function getSqrt() {
    return someMathLibrary.sqrt(i);
  }

  // public interface
  return {
    increment : inc,
    decrement : dec,
    getSqrt : getSqrt
  };

}(myMathLibrary));
```

Code Listing 7.4: Module dependency management

utilised: a script adds the library object(s) to the global namespace, and it is from then on accessible by variable reference.

While quick and simple to use, this technique means it is potentially very hard to manage the dependencies of each component as they are never explicitly declared. The Module (and Revealing Module) pattern enables a very simple and effective technique for declaring and utilising dependencies. Dependencies are passed as parameters to the module function upon declaration/invocation, and then only referenced by their local identifier. Code Listing 7.4 is a demonstration with the dependency `myMathLibrary`.

The obvious disadvantage of this technique is that it isn't strict. Code inside a module *may* still access the global namespace, and by that circumvent this "import system". Using the example of Code Listing 7.4, the `getSqrt()` function could access the math library with the `myMathLibrary` reference just as well as the `someMathLibrary` reference. Access to dependencies should when employing this pattern be limited by the programmer to be performed only via the input parameters of the module function; any violation is not in line good practice.

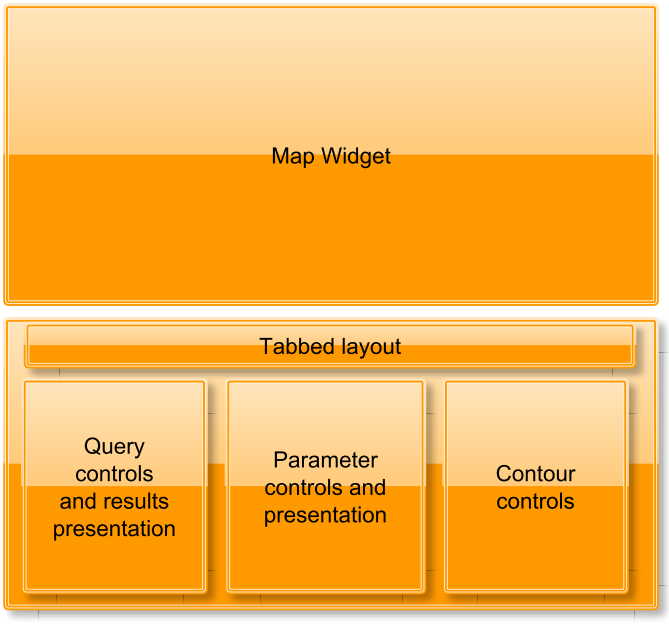


FIGURE 7.1: Front–end application layout conceptual model

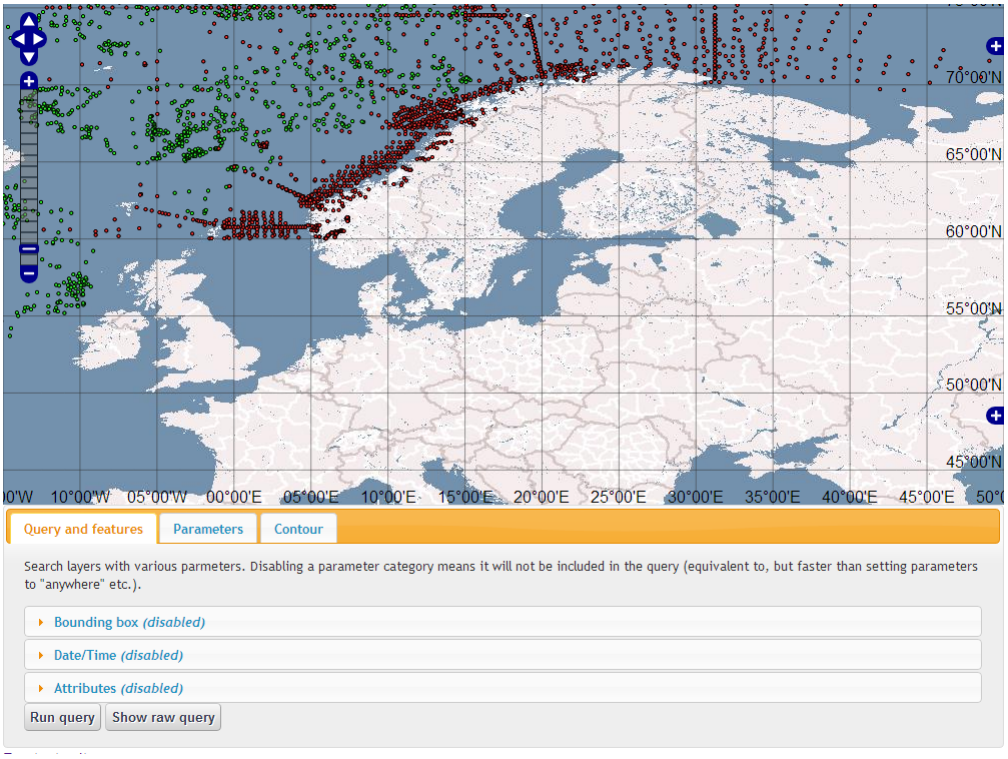


FIGURE 7.2: Front–end application layout screenshot

7.4 Client application design

Figure 7.1 is a conceptual model of the application GUI. Figure 7.2 is a screenshot of our implementation of this design.

The layout is implemented in standard–compliant HTML, CSS and JavaScript. Effort is taken to avoid the usage of tags or techniques that have been deprecated in HTML standard 4 [97], and to align the implementation with current draft of HTML standard 5 [98].

The Map Widget is implemented with OpenLayers, while the Tabbed layout–section of the design is implemented using the `tab` widget from jQuery UI. Additional widgets from jQuery UI, such as the `accordion`, `modal popup` and `progressbar`, are also used throughout the application.

7.5 Client application architecture

The client application is constructed from several components that fulfil roles and responsibilities in line with the *Single Responsibility Principle* (SRP) [99]. Components are constructed according to the techniques and design patterns described in section 7.3. Effort has been made to construct a modular system where components are loosely coupled.

The modules of the application fall into two categories: *control* and *utility* modules. Control modules are responsible for user interaction, while the utility modules are smaller modules that independently perform specific tasks. Control modules are somewhat tightly bound both to each other and to the presentation of the application (through reading and manipulating the current HTML via the *Document Object Model* (DOM). Utility modules, on the other hand, are not aware of or dependent on each other, or the control modules, and never directly affect presentation as they do not utilise the DOM. A new feature has typically been implemented by constructing a new utility module to handle tasks that the feature require, such as performing a calculation, and the functionality of the module is then exposed to the user through the control modules.

The application is presented as a UML class diagram in the next section. An overview of the complete design, with a brief introduction to each component is then given. After this, we take a closer look at the components and interesting code segments.

Generally, source code presented will be snippets and may be somewhat altered from the actual source due for the sake of presentation. Appendix C provides a complete source

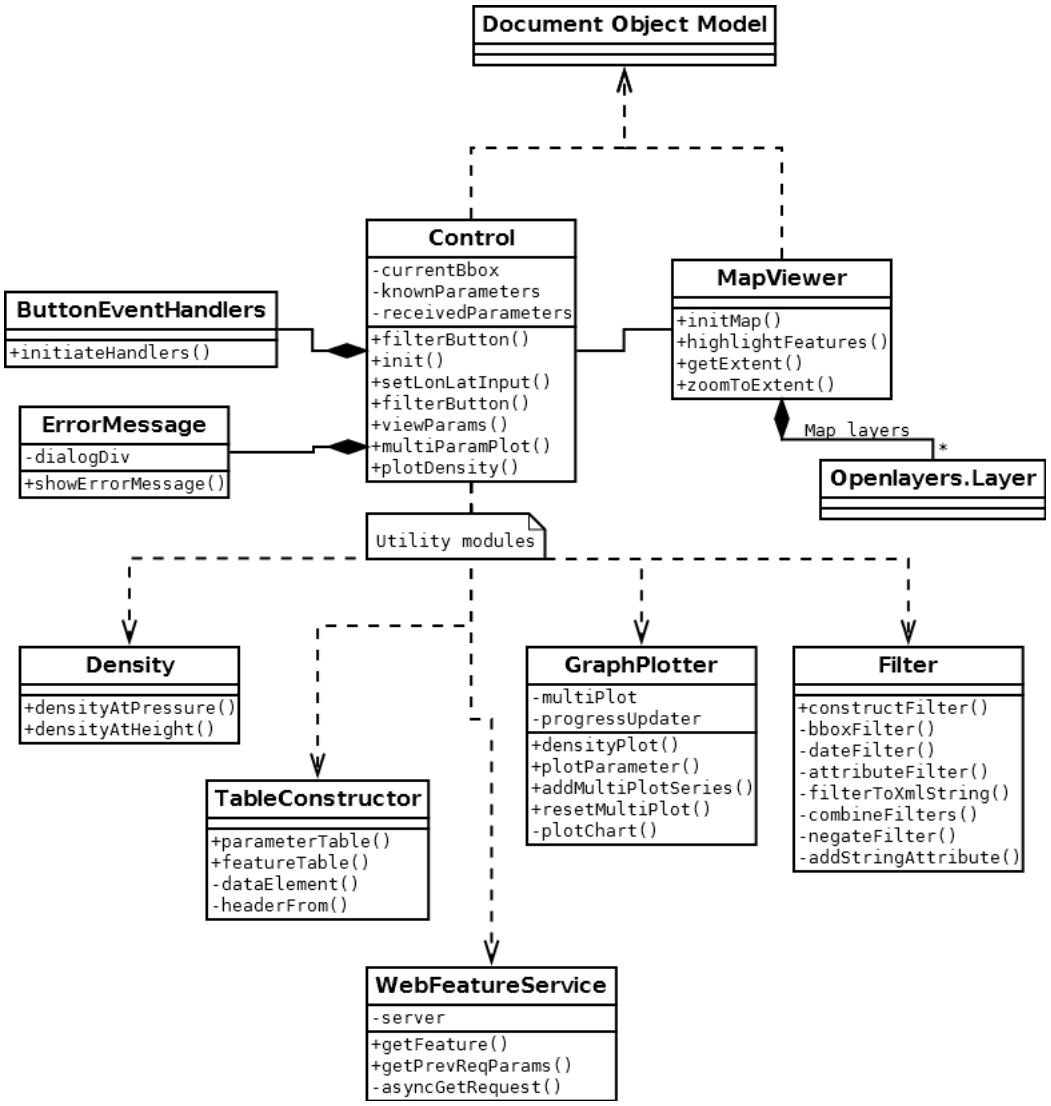


FIGURE 7.3: Front–end application UML class diagram

code listing for the `WebFeatureService` module exactly as it appears in the application, along with detailed annotation.

7.5.1 UML model

Figure 7.3 is a UML class diagram of the client application architecture. The components of the diagram and the relationships between them are introduced in Section 7.5.2.

A Revealing Module is represented in the model as a class. The inner functions of the module are represented as methods, with their access modifier based on accessibility outside the function scope of the module function itself. Due to space restrictions, some method names are shortened and parameters/data types are not represented.

Due to the lightweight nature of JavaScript objects, many are instantiated anonymously throughout the application as simple data containers or temporary functions. Such objects are not included in the model. The DOM is also present in the model, represented as a single class. Its methods and fields are not listed due to its significant size. Selected objects from libraries in use, that may or may not follow the Revealing Module pattern, are also represented as classes in the diagram.

7.5.2 Modules and relationships

The diagram has been organised to emphasise the division between control- and utility modules. The upper part of the diagram, dominated by `Control` and `MapViewer` are the control modules. The utility modules have been placed in the lower half of the diagram, all associated with the rest of the application with single, unidirectional dependencies from `Control`.

`Control` and `MapViewer` are the central modules of the application, and handle all interaction with the user. These modules also interact with each other to offer interactive functionality between the map widget and the rest of the application. In the diagram, this is represented as a 1–to–1, bidirectional association.

The modules `ButtonEventHandlers` and `ErrorMessage` encapsulate some functionality that conceptually belongs in the `Control` module. They are not represented as utility modules in the diagram, as they directly effect presentation and are only suited to be used by the `Control` module. `ButtonEventHandlers`, for example, is only intended to be invoked *once* as the application is invoked, and this invocation is performed by the `Control` module.

The current set of utility modules, and their roles are listed below.

- The `TableConstructor` module creates various kinds of HTML table representations of objects. The result is returned in a string format.
- The `WebFeatureService` module provides abstractions of operations from the WFS standard. It returns results by invoking a provided callback function.
- The `GraphPlotter` module creates graph plots from objects.
- The `Filter` module creates a Filter Encoding filter XML file based on parameters.
- The `Density` module calculates seawater density based on salinity and temperature.

```
1 $.each(knownParameters, function(i, val) {
2   myNamespace.WebFeatureService.getFeature({
3     TYPENAME : parameterPrefix + val,
4     VIEWPARAMS : 'n:' + id
5   }, function(response) {
6     displayParameter(response, val);
7   });
8 });
```

CODE LISTING 7.5: `Control` module: leveraging functional programming

We now look at each module in detail, further explaining its role and implementation. Source code excerpts that are especially relevant to functional requirements or interesting in their own right are presented in code listings.

7.6 Control modules

7.6.1 Control

The `Control` module connects all other modules and is responsible for all user interaction that is not done through the map widget. It is involved in fulfilling all functional requirements, and has a relatively large public interface with many functions. `Control` is also the entry point of the application. Its `init()` function is invoked once the document that contains the application has fully loaded. This function then initialises the module itself and the other control modules.

Most of the functions in the public interface of `Contol` are directly connected to user actions on the GUI, such as button presses. An example of this is `viewParams()`, which uses utility modules to display parameter values (FR2). Code listing 7.5 is the main content of this function. We include it and discuss it now as an example of how we utilise JavaScript's support for the functional programming paradigm to produce concise and expressive code.

The main construct of the code listing is a jQuery *for each*–like looping construct. It is a function that takes two parameters: a list to iterate over (in this case `knownParameters`), and a function to invoke for each element. We will refer to the latter function as the `iterator` function. For each invocation, `iterator` receives the index `i` and value `val` (in a traditional for–loop, this would be `array[i]`) of that element.

We supply an anonymous function as `iterator`. It invokes the `getFeature` function of the `WebFeatureService` module. This function takes two arguments: an object describing

the options of the WFS `GetFeature` operation, and a function that will act as a *callback*. We will refer to this last function simply as `callback`. In our invocation, the object is given as a literal via JavaScript's object literal notation. It will only exist as a parameter in some function invocation – for this purpose it is very convenient to not have to declare any class construct or do some constructor method operations. JavaScript's lightweight object model is highly effective in such situations.

The `callback` is declared equally tersely: as an anonymous function. This function will be invoked once the resulting AJAX call is completed – that is, when the results for the `GetFeature` operation arrive from the server. By passing this function as a parameter, in the functional programming–style, the `WebFeatureService` module needs not be concerned with the presentation of the results. The `callback` function is in this case merely a new invocation of a function in `control`, but invoked with both the response to the request and an additional parameter required by this function.

The other public functions of `Control` are similar to `viewParams()`, if not in content then in style and purpose. They are small and often utilise several utility modules to perform their task. Since it is involved in most common use cases of the client, `Control` will be covered more in Chapter 8.

7.6.2 MapViewer

The `MapViewer` module creates and manages the map widget. It fulfils FR4 (interactive map widget) and FR5 (interactivity with the query GUI), and is also responsible for the presentation aspect of FR8. `MapViewer` heavily depends upon the OpenLayers library and the DOM to draw graphics and handle mouse input (used in panning/zooming), but it only uses this directly on the map widget. Other interactions with the GUI are handled through `Control`.

The main functionality of the map widget is an automation of the task demonstrated in Section 3.4.2: requesting, composing and displaying images from our WMS service. Manually making these requests is cumbersome, but OpenLayers makes the task easy both for the developer and the user: all these operations are performed automatically and rendered to a `div` tag once the library is correctly set up.

The majority of the code in the module, then, is declaration, initialisation and configuration of OpenLayers layer and control objects. Code Listing 7.6 is an excerpt from the widget setup, where a OpenLayers `WMS` object is constructed. The constructor takes four arguments: a name, the WMS service URL, a layer description object and an options

```
stations : new OpenLayers.Layer.WMS("Stations",
  myNamespace.WMSserver, {
    layers : 'station',
    format : WMSformat,
    transparent : true
  }, {
    isBaseLayer : false
  }),
```

Code Listing 7.6: `MapViewer` module: declaring a layer

```
OpenLayers.Util.extend(control, {
  draw : function() {
    // intercept shift+mouse click
    this.box = new OpenLayers.Handler.Box(control, {
      "done" : this.notice
    }, {
      keyMask : OpenLayers.Handler.MOD_SHIFT
    });
    this.box.activate();
  },
  // notice function invoked after shift+mouse click action
   on map
  notice : function(genbounds) {
    // get pixel values of box: lower left, upper right
    var ll = map.getLonLatFromPixel(new OpenLayers.Pixel(
   genbounds.left, genbounds.bottom)), ur = map.
   getLonLatFromPixel(new OpenLayers.Pixel(genbounds.right,
   genbounds.top));

    // convert to lon/lat and set as current lon/lat query
   input
    myNamespace.control.setLonLatInput(ll.lon.toFixed(4), ll
   .lat.toFixed(4), ur.lat.toFixed(4), ur.lon.toFixed(4));
  }
});
```

Code Listing 7.7: `MapViewer` module: map–control interaction

object. This object will later be added to a OpenLayers `Map` object along with similarly initialised background layers.

As mentioned earlier and displayed in the UML class diagram, `MapViewer` and `Control` are associated bidirectionally. This association is used to fulfil FR5. An example is given in Code Listing 7.7. An OpenLayers *control* is added to the map widget that provide a functionality that lets the user drag a box on the map widget; this box is then set as the current latitude/longitude input in the query GUI. The functionality is added to the map widget by using the OpenLayers utility function `extend`, which copies the

properties of a new object (the second parameter) onto another (the first parameter). The interaction with `Control` occurs on line 16–17.

7.6.3 ButtonEventHandlers

The `ButtonEventHandlers` module is a simple module with a single public function that initiates event handlers for all the input elements of the GUI. Most event handlers are click events on buttons that map to public functions in `Control`.

Adding event handlers is performed differently depending on the browser vendor and version. The module exclusively uses jQuery event handling functions are used to achieve compatibility. These functions are designed with compatibility in mind, meaning the event handling should work on all standard browsers. This practice also means the application complies with the W3C DOM level 3 event handling–standard on browsers that support it [100]. This is the current/future standard for event handling.

7.6.4 ErrorMessage

`ErrorMessage` is another simple module that is essentially a part of `Control`. It provides public functions for displaying error messages to the user. The messages are displayed in jQuery UI's *modal windows*, ensuring that the user is made aware of errors before being proceeding.

7.7 Utility Modules

The utility modules are independent, singleton components with narrow and well–defined roles that typically correspond to one or more functional requirements (see Table 2.2). They handle their own initialization as they are loaded, and may be used by any other component of the client application. In our application, only the `Control` module invokes functions of the utility modules.

Their independent design mean each may be replaced by another module as long as it provides the same public interface. They could also easily be reused in other systems, even in a non–web setting as they have no reliance on the DOM.

```
1  // iterate through all features, generate table row for each
2  $.each(parameters, function(i, val) {
3    var property = val.properties;
4
5    row = "<tr>";
6
7    row += data(property.level);
8    row += data(property.value);
9    row += data(property.flag);
10
11   rows += row + "</tr>\n";
12 });
```

CODE LISTING 7.8: `TableConstructor` module: creating HTML table rows for features

7.7.1 TableConstructor

The `TableConstructor` module is responsible for FR2 (tabular presentation); presenting features and parameters in tabular form. More specifically, it takes parameter or feature objects, and construct strings of HTML table tags with information extracted from the objects. This constructed string is returned, and may then be injected into the HTML of the client web page by the invoker.

The public interface of `TableConstructor` is very simple, with one function for parameter tables and one for feature tables. It also has private helper functions for creating HTML table–row and element strings from data. The produced tables have some advanced functionality, such as sorting by column, added with the HighCharts jQuery plugin. This enhancement is performed by `Control` after `TableConstructor` has created a table, since it involves DOM manipulation.

Code listing 7.8 is an excerpt from the `parameterTable()` function. Each parameter that has been supplied as the `parameters` list is converted to a table row, and the resulting row is added to the total of rows. The data elements for each row is generated by wrapping the properties of the parameter in `td` tags through a call to `data`. The rows string is concatenated with a header string and returned. Feature tables are constructed similarly, but with an additional inner loop to add the larger amount of properties to a row.

7.7.2 WebFeatureService

The `WebFeatureService` module encapsulates and abstracts WFS operations (recall Section 3.5). Currently, only `GetFeature` is implemented, as this is the only operation

needed in the application. WFS is essential to fulfilling FR1 and FR9, and all other functional requirements are built on its retrieval of data.

Despite its important role, `WebFeatureService` is quite compact. It also demonstrates the use of the revealing module pattern, code style and other details of the source code that we will not touch upon in the main text. Because of this, we have chosen to include the entire module along with detailed code annotations in Appendix C.

7.7.3 GraphPlotter

The `GraphPlotter` module draws graph plots to represent parameter data at different depths. It is responsible for FR3 and FR6, and also involved in FR7. It is built around the HighCharts library (See Section 4.4.3),its public interface consists of functions that set up HighCharts objects with different settings depending on the desired graph plot layout.

Code Listing 7.9 is an excerpt from the `multiPlot()` function that draws several data series to a single plot. This function is invoked for each series, but they are retrieved asynchronously: no ordering can be assumed. Lines 2–4 occur for all but the first invocation, since `multiPlot` will then be defined and evaluate to `true`. Lines 5–18 occurs on the first invocation of the function (that is, for the first retrieved series). In addition to creating the graph plot itself by invoking the module's private function `plotChart`, it also initiates a progress bar to display how many series has been retrieved to the user. At the end of each invocation, a check is performed to see if the added series was the final one. If so, the graph plot is drawn and the progress bar destroyed.

7.7.4 Filter

The `Filter` module creates Filter Encoding (recall Section 3.6.3) XML strings from parameter objects that describe a spatial, temporal and attribute–based filter. It is essential in fulfilling FR1, as this XML string is passed as a parameter in the WFS `GetFeature` operation.

`Filter` relies heavily on the `Filter` and `Format` objects of OpenLayers. These are storage objects with utility methods for creating Filter Encoding filters. Code listing 7.10 is the private `combineFilters()` function of the module, displaying how OpenLayers (here aliased as `OL`) objects are used. The `combineFilters()` function takes an array of OpenLayers `Filter` objects and returns a single object that has the array concatenated with the logical `AND` operation. The module has similar private functions for negating filters, and more.

```
// already created?
if (multiPlot) {
  // don't redraw, don't animate
  multiPlot.addSeries(series, false, false);
} else { // not created, so we create it now
  multiPlot = plotChart(container, [ series ], "Level", "
   Temperature", "Temperatures by level", "line");

  // initiate a progress bar
  $("#multiPlotProgress").progressbar({
    max : finalSize,
    value : 1
  });

  // update prog.bar every second
  progressUpdater = setInterval(function() {
    $("#multiPlotProgress").progressbar("value", multiPlot.
   series.length);
  }, 500);
}

// draw plot if all series are now loaded
if (multiPlot.series.length === finalSize) {
  multiPlot.redraw();
  clearInterval(progressUpdater);
  $("#multiPlotProgress").progressbar("destroy");
}
```

Code Listing 7.9: `GraphPlotter` module: generating graph plot for with multiple series

```
// combine array of filters to single filter
function combineFilters(filtersToCombine) {
  return new OL.Filter.Logical({
    type : OL.Filter.Logical.AND,
    filters : filtersToCombine
  });
}
```

Code Listing 7.10: `Filter` module: combining filter objects

```
1 // write OpenLayers filter object to OGC XML filter encoding
2 function filterToXmlString(filter) {
3   var formatter = new OL.Format.Filter(), xmlFormat = new OL
     .Format.XML();
4   return xmlFormat.write(formatter.write(filter));
5 }
```

Code Listing 7.11: `Filter` module: object to XML representation

A complete filter object is constructed by first creating individual `Filter` objects for the input parameters, and then combining them by using the `combineFilters()` function. The OpenLayers `Filter` object is then converted to an XML string by the function shown in Code listing 7.11.

7.7.5 Density

The `density` module calculates seawater density according to the International equation for state of seawater [101], the backbone to fulfilling FR7. Calculations are based on temperature, salinity and depth. The option of calculation density becomes available in the GUI when a feature is selected that has data for the temperature and salinity parameters.

The module exposes only a single function through its public interface. This is the function that calculates density based on the described parameters. Many private functions are used as helper functions in the module, most corresponding to the formulae of [101]. One of these is displayed in Code Listing 7.12. Note that the somewhat cryptic variable names used here are based on the names used in (article).

Presentation and readability has been greatly prioritised over speed of execution in the implementation. The rationale here is to allow for easy optimisation later if it should be desired. Possible optimisations include pre–calculating constant values and eliminating repeat calculations (as are done for powers of the temperature and powers of ten – for example in lines 14–18 of Code Listing 7.12).

Another interesting optimisation would be to convert the `density` module into a *Web Worker* [102]. Since the task of calculating densities for multiple *(temperature, salinity, depth)*–tuples is *embarrassingly parallel* (each calculation being completely independent of the others) it would benefit greatly from parallel execution [103]. Web workers lend themselves to parallel execution. Since all code for calculating density is already contained in a separate module, extracting it to a web worker would be a fairly simple task. It was not prioritised in the thesis work due to execution speed being sufficiently fast in the current implementation.

```
1  // (10) of article
2  function pureWaterDensity(temp) {
3    var t = [], a = [];
4
5    // t to the power of n
6    t[1] = temp;
7    t[2] = Math.pow(temp, 2);
8    t[3] = Math.pow(temp, 3);
9    t[4] = Math.pow(temp, 4);
10   t[5] = Math.pow(temp, 5);
11
12   // a
13   a[0] = 999.842594;
14   a[1] = 6.793952 * Math.pow(10, -2);
15   a[2] = -9.095290 * Math.pow(10, -3);
16   a[3] = 1.001685 * Math.pow(10, -4);
17   a[4] = -1.120083 * Math.pow(10, -6);
18   a[5] = 6.536332 * Math.pow(10, -9);
19
20   return a[0] + (a[1] * t[1]) + (a[2] * t[2]) + (a[3] * t
      [3]) + (a[4] * t[4]) + (a[5] * t[5]);
21 }
```

Code Listing 7.12: `Density` module: equation example

Currently, IEEE 754 floating point numbers are used in the calculation (the standard floating point representation in JavaScript). If increased precision is desired, optimisations may become necessary due to the reduced execution speed of arithmetic.

The calculated density values are presented in a table and in a graph. The table presentation reuses the function for rendering parameter regular tables. The graph presents the density calculation along with temperature and salinity graphs. This is handled by a custom function in the `GraphPlotter` module that allows for several value ranges on the same axis. All presentation interaction is handled by the `Control` module after the `Density` module has performed the calculation. This means that `Density` as a module is highly reusable, and could be used by any project requiring its or similar kinds of calculations.

7.8 Testing and quality assurance

7.8.1 W3C standard compliance

As stated in 7.4, the front end application conforms to current W3C standards in markup and layout. For the purpose of testing compliance to the HTML standard version 5 (the

upcoming standard, currently in draft form), the World Wide Web Consortium's markup valication service [104] was used. The markup generated by the front–end application validates as HTML5 without errors. All widgets and common interactions were tested by validating markup generated as common use cases were carried out.

It should be noted that while the markup generated by the application itself successfully validates, this is not necessarily the case when hosted as a portlet in Liferay, as Liferay includes some markup of its own. This is an issue that must be resolved on the portal side (in this case, by Liferay), as a portlet has no control over the global markup.

The CSS of the application was tested using the W3C CSS validation service [105]. It successfully validates as CSS level 3 without errors.

7.8.2 JavaScript best practices

JavaScript as a language has some pitfalls that should be handled in implementation. To aid in discovery of potential error sources, and to align the source code with a consistent style, the JSLint Code Quality Tool [106] has been used in the development.

Some of the rules JSLint enforces are uncontroversial, such as forbidding the use of `eval` and unreachable code segments. Other, such as the rule that loop counters in for loops shouldn't use the increment operators (that is, a standard loop counter should be incremented as "`i += 1;`" rather than "`i++;`"), are arguably more a matter of preference by the author. Since our purpose in using this tool is mainly to achieve a consistent style, we have not excluded any rules when validating the source code with JSLint.

Code style formatting was handled by the automatic formatter of the Eclipse IDE. Identifiers for variables and function names are descriptive, and the source code is adequately commented.

7.8.3 Unit testing

Unit testing (also known as component testing) verifies that a component behaves according to its specifications [107]. The utility modules of the application are well–suited for unit testing due to their independent nature and concise public interface.

QUnit [108] was used as a testing framework. The framework was simple and lightweight in integration, and the tests are run through the browser by accessing a URL that is linked to from the application web page (this link should be omitted when the application is in production). QUnit presents tests and tests results neatly, with functionality for

```
1  test("Filter object to Filter Encoding string conversion",
       function() {
2
3    var expected = "<ogc:Filter xmlns:ogc=\"http://www.opengis
       .net/ogc\"><ogc:And><ogc:BBOX><ogc:PropertyName>stpoint</
       ogc:PropertyName><gml:Box xmlns:gml=\"http://www.opengis.
       net/gml\"><gml:coordinates decimal=\".\" cs=\",\" ts=\"
       \">0,0 10,10</gml:coordinates></gml:Box></ogc:BBOX></ogc:
       And></ogc:Filter>";
4
5    // construct filter for bounding box
6    var bbox = new OpenLayers.Bounds(0, 0, 10, 10);
7    var resultString = filter.constructFilterString(bbox, null
       , null);
8
9    equal(resultString, expected, "BBOX should be represented
       correctly in the constructed Filter Encoding string");
10
11   // similar tests follow ..
```

Code Listing 7.13: Unit test: Filter Encoding generation

rerunning any test and options for selecting which tests are to be run (categorized by module).

Code Listing 7.13 is an excerpt from the unit test for the `Filter` module. In the particular test case that is displayed, the correctness of the constructed Filter Encoding XML from `Filter` is verified for bounding boxes. Similar tests are run for other filtering options and combinations.

Figure 7.4 is a screenshot of a test run in QUnit, displaying the GUI. Some tests have been altered to fail for demonstration purposes. Notice the module selection menu in the upper right corner of the figure.

FIGURE 7.4: QUnit test framework demonstration

8 System overview and demonstration

In this chapter, we look at the constructed system in use through a set of use cases that are based on the functional requirements for the client application. Our purpose is both a demonstration of the application as well as a look at how the front–end application and the remaining software stack interact. The individual actions and components have been described in the previous chapters, but we now aim to give a full overview of the complete system in action.

A use case is covered in a section, with variations presented in subsections within. Complex use cases may be represented in a UML sequence diagram. The objects of these diagrams will be involved modules of the front–end application, the data accessor (GeoServer) and the data store (PostGIS). Messages between the objects will be function invocations, WMS/WFS AJAX requests and SFA data store access.

For the sake of presentation, some low level details are left out in the diagrams:

- Parameters of messages are simplified. For instance, names may be shortened and unimportant parameters may be omitted.
- The user actor that initiates the process is omitted from the model. The function call that starts the procedures will be mentioned in the accompanying text.
- If procedures are similar to those displayed in previous sequence diagrams, they may be omitted or simplified to avoid repetition.
- Parentheses are omitted for function invocations with no known parameters (for example callback functions)

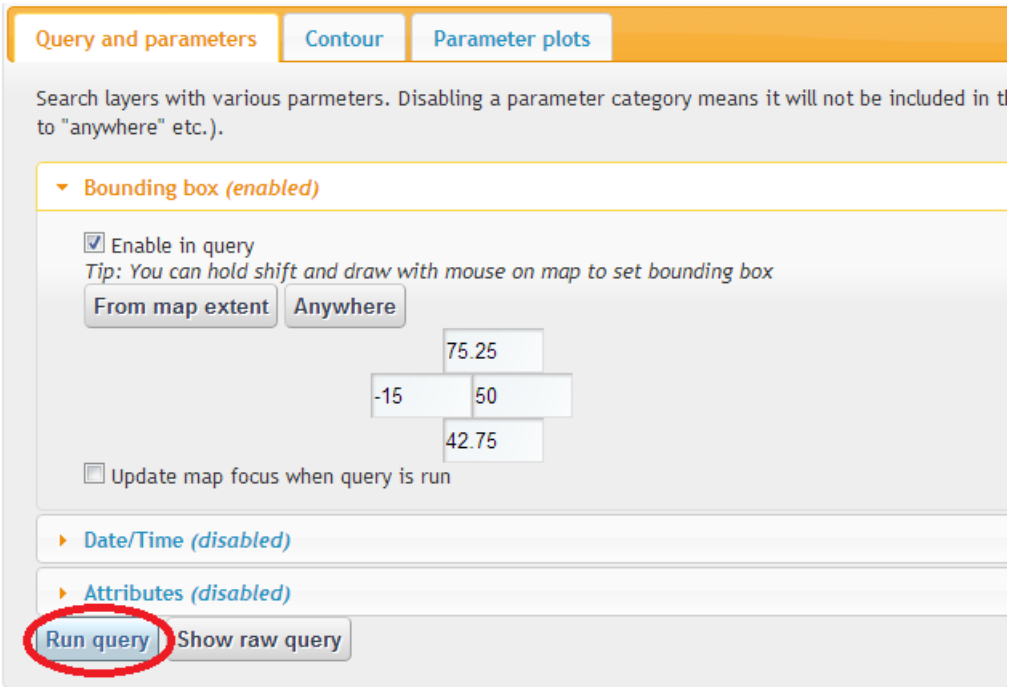


FIGURE 8.1: Initiating a search with the query GUI

8.1 Searching in data

A search is initiated when a user clicks a button that invokes the `filterButton()` function of the `control` module. The current values in the query GUI, as input by the user, is now to be processed and a search is to be initiated. Figure 8.1 is a screenshot of the query GUI. The highlighted "run query" button has been linked to the `filterButton()` function during initiation.

The following process is presented in two sequence diagrams due to the number of involved components and messages. The first diagram, Figure 8.2, presents the procedure up until the request for data is completed – we'll call this the request procedure (corresponding to FR1). The second diagram, Figure 8.3 presents how the received data is handled – the presentation procedure (corresponding to FR2 for features).

The exact moment of division is when the callback function `displayFeatures()` of the module `control` is invoked: this function is the callback handler for successful data retrieval. This invocation is included in both diagrams, being the last message of Figure 8.2 and the first message of Figure 8.3. In the first diagram, it is referred to simply as `callback` to have it stand out among regular parameters. The second diagram uses the full name.

Notice in both diagrams that no utility modules interact with each other directly, only through the `control` module. This is a visual representation of the modular design of

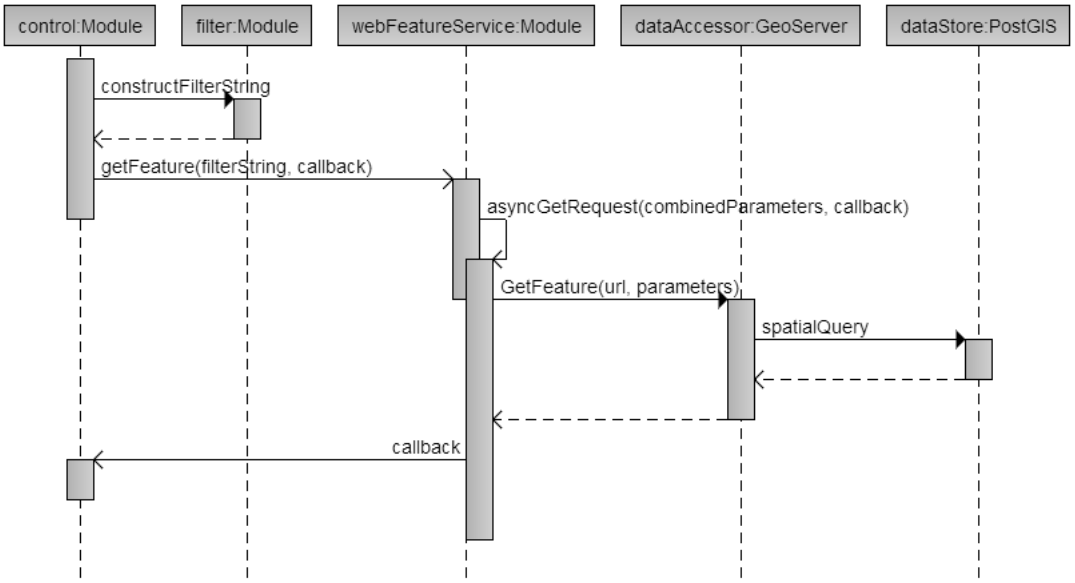


FIGURE 8.2: Sequence diagram: searching in data – request procedure

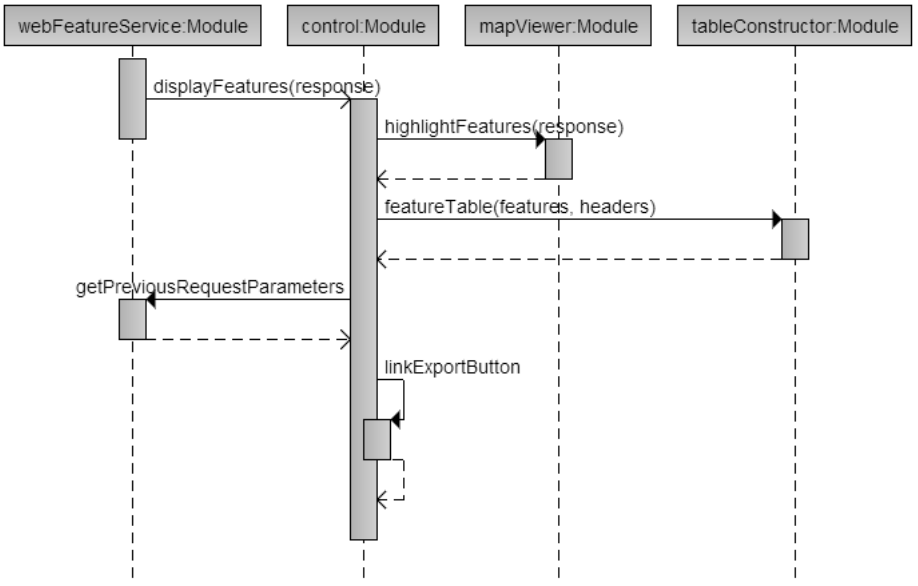


FIGURE 8.3: Sequence diagram: searching in data – presentation procedure

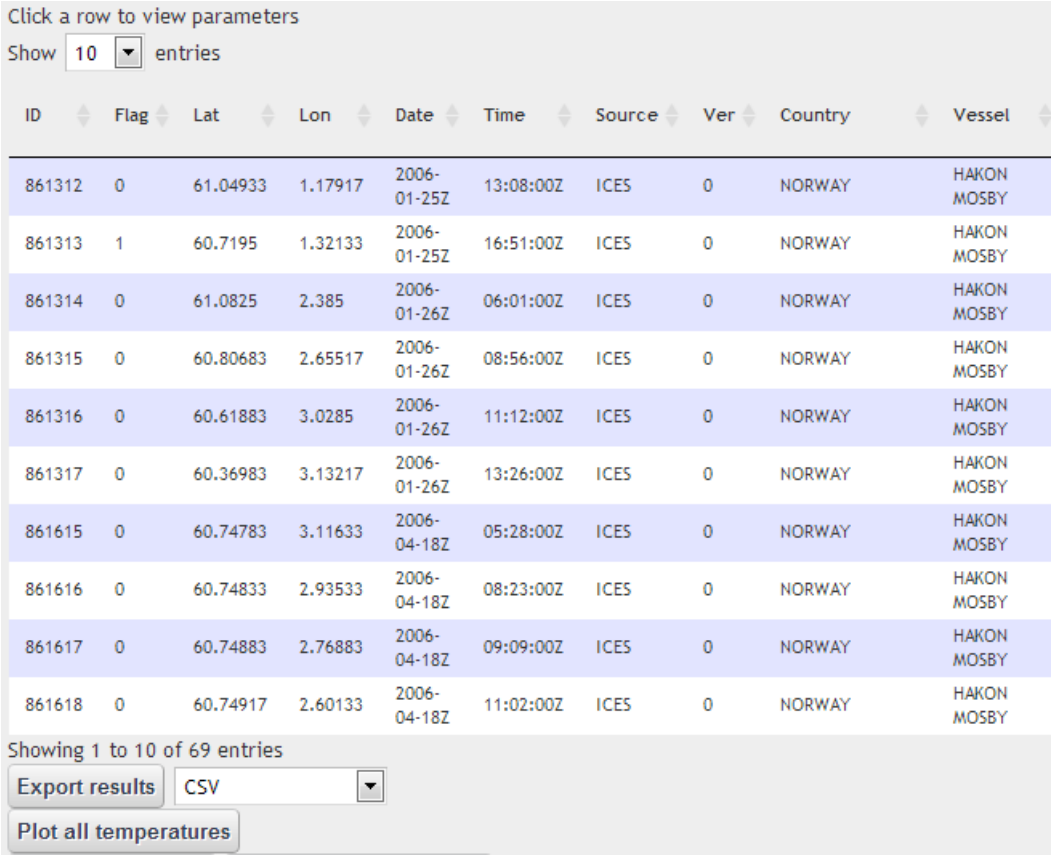


FIGURE 8.4: Search results in the GUI

these components, showing clearly that they are indeed independent. Notice also that they only invoke functions of the `control` module as callbacks that are passed to them by the `control` module itself (the example here being the `displayFeatures()`) function, fulfilling the unidirectional dependencies in figure 7.3.

The results of the search are presented in the GUI, see figure 8.4. More options are now available to the user. The invocation of the `linkButton()` function as seen in figure 8.3, for example, mean that the "export results" button will produce a representation of the search result in the selected format. A user could also view parameter values for a feature, the use case covered in section 8.2.

8.2 Viewing parameter data

This process is started when a user clicks an element that invokes the `viewParams()` function of `control`. In the current front–end application, this occurs when a user clicks a table row after searching for data. The `viewParams()` function is then invoked with the `id` of the clicked row as a parameter.

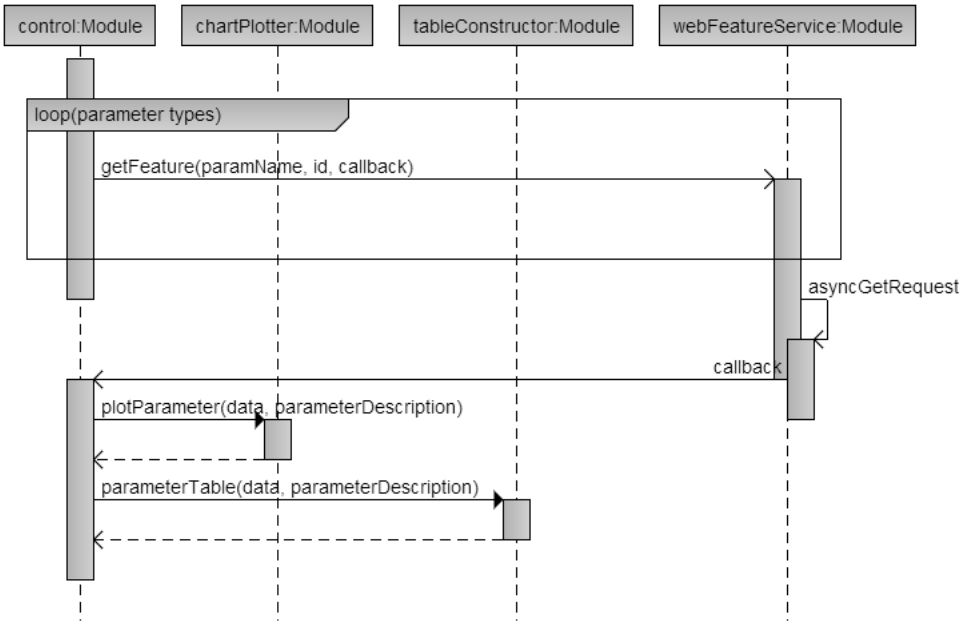


FIGURE 8.5: Sequence diagram: viewing parameter data

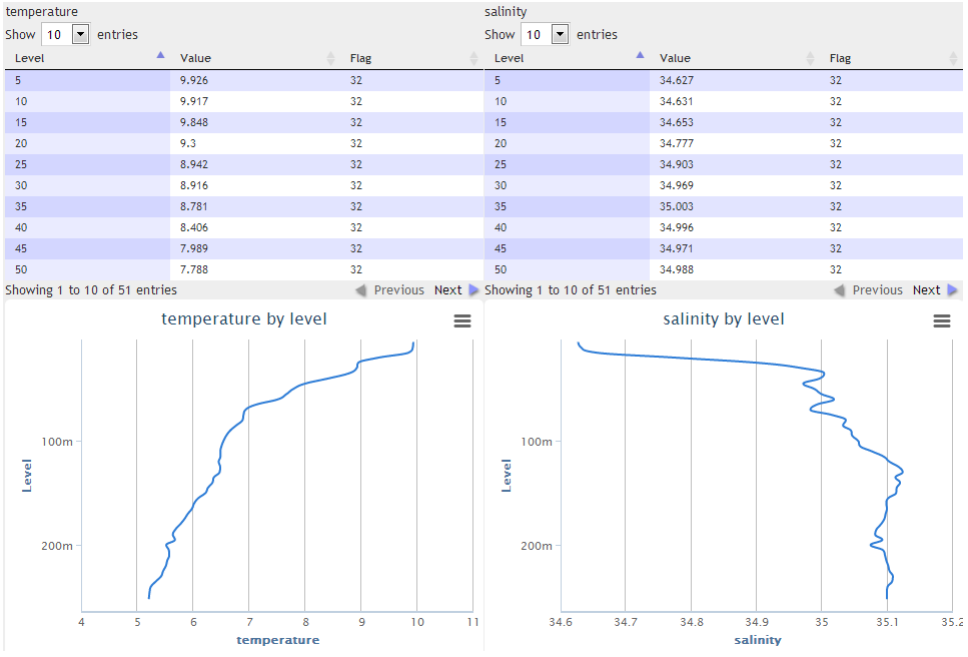


FIGURE 8.6: Viewing parameter data

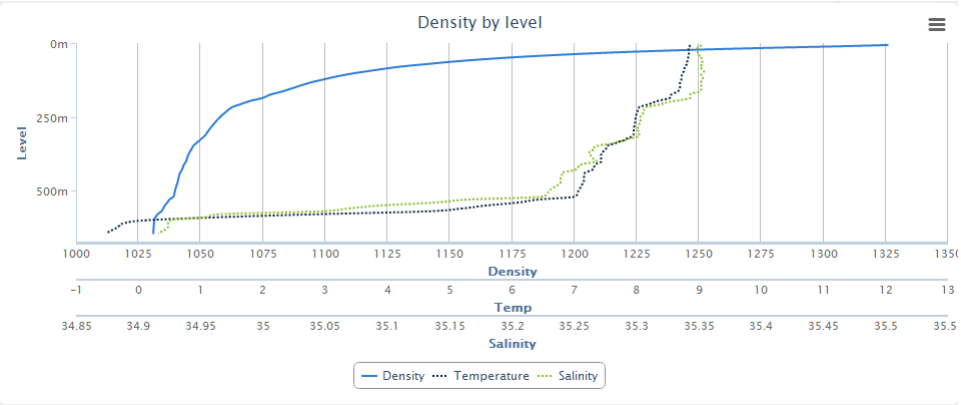


FIGURE 8.7: Viewing calculated density data in a graph plot

The procedure is shown as a sequence diagram in Figure 8.5. The code behind the loop construct from this diagram can be seen in Code Listing 7.5. Note that GeoServer and PostGIS have been omitted from the diagram. They would appear as in figure 8.2 and 8.3, as these `GetFeature` operations appear similar to the client application, except that an `id` is included as a parameter (see Section 6.3).

Figure 8.6 is a screenshot of the GUI presentation of this operation. In the screenshot, tabular (fulfilling FR2 for parameters) and graph plot (fulfilling FR3) presentations of temperature and salinity values for an arbitrary feature can be seen.

8.2.1 Calculated data: calculation and presentation

Calculating and presenting density values for a feature is required by FR7. The procedure for viewing calculated parameter data is much the same as with regular parameter data, apart from the inclusion of the calculation itself. The `density` utility module calculates density values for the currently selected feature, and stores the results in a structure similar to that of regular parameter series. This allows the presentation in table and graph plot to be handled by the regular modules, with some specialisation on the graph plot to allow for multiple value ranges on the x–axis. A screenshot of a produced graph plot is provided in figure 8.7.

8.2.2 Viewing parameter data from several features

While the regular parameter presentation procedure loops over the known parameter types, FR6 is based on the data from *one* parameter type retrieved for several features. Figure 8.8 is a screenshot of a graph plot for the temperature parameter values of over 100 features. The rendered lines have various colours applied to make them easier to distinguish from one another. These types of graph plots can be useful to get an idea

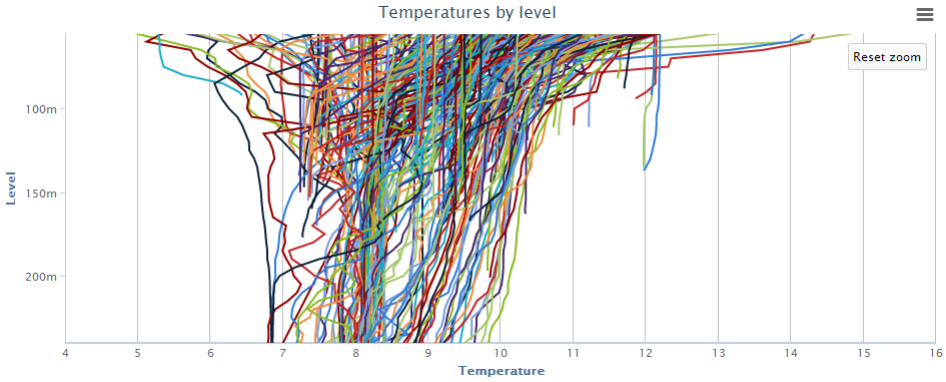


FIGURE 8.8: Viewing parameter data from several features

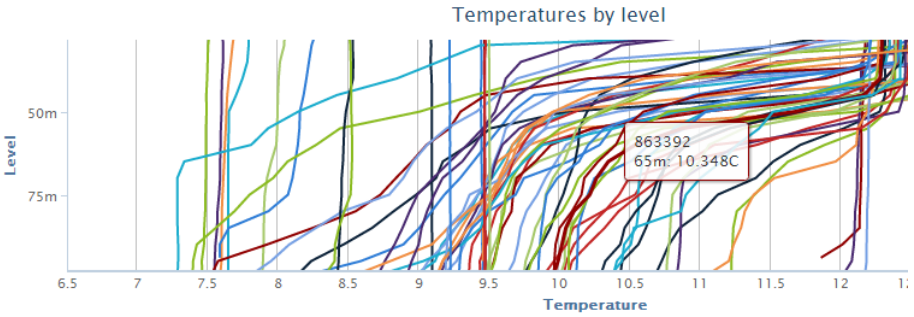


FIGURE 8.9: Graph plot interactivity

of the range of values present in a selection of features. In Figure 8.8, for instance, it is clear that the range of temperatures narrows as depth increases. Outliers, the results of extraordinary conditions or measurement errors, are also easy to pick out visually.

The graph plots in the client are interactive, not just static images. Interactivity is available on all graph plots in the client, but we demonstrate it here in context of the multiple feature–plots since it is especially useful in this context. The screenshot in Figure 8.9 display some of the graph plot functionality. A pop–up box appear on when mousing over a line in the graph (note that the mouse pointer is not captured in the screen shot, but is positioned at the pop–up box location). The box displays relevant information; here the id of the selected line and its value at the selected location (depth, and value for that depth). The graph plot also has zoom functionality for the Y–axis (depth), which can be seen in the figure as the range for this axis would normally start at 0 meters. As the user zooms, the X–range is also updated to reflect the value range present in the selected Y–range. Export of graph plots is also available, and described in Section 8.4.

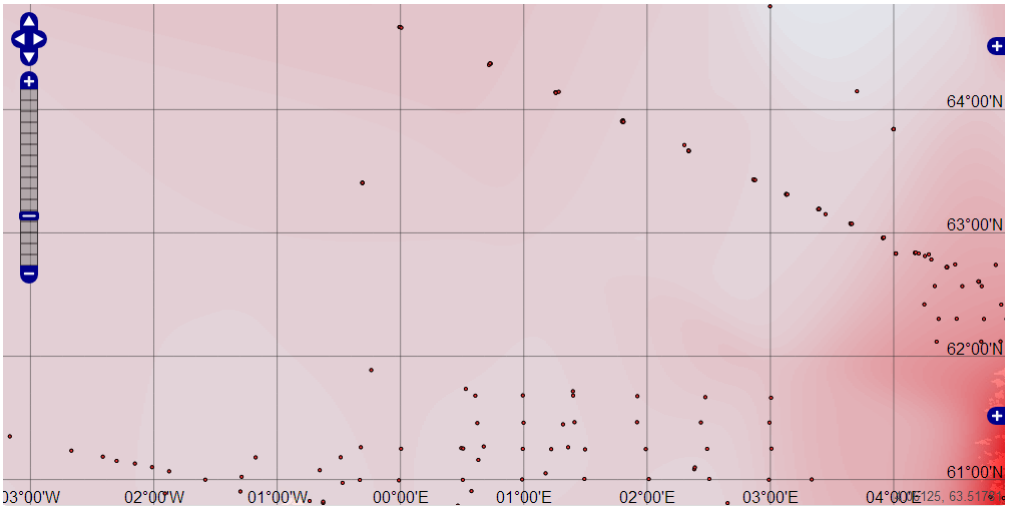


FIGURE 8.10: Interpolated contour plot

8.3 Contour plots

Basic interpolation for contour plots are implemented in the geographic data store to fulfil FR8. The implementation is described in Section 6.4. Since the contour plot is provided from the data accessor as a WMS layer, adding it in the client is done as in Code Listing 7.6. A screenshot of a contour plot rendered in the map widget is displayed in Figure 8.10. This particular contour plot is based on the temperature parameter at depth 5, represented with colour ranging from black (low temperature), through red (medium temperature) to white (high temperature). The SLD behind the interpolation is included in Appendix B. Which parameter, value and the applied colours and colour ranges the plot is generated from are editable.

8.4 Exporting features and graph plots

FR9 states that the system should be able to export the data for all tables and graph plots. The background for the implementation of this functionality was discussed in Section 6.5. The option to export data to a number of formats is available for all tables in the GUI. Figure 8.11 is a screenshot of the GUI for this operation. Graph plots may also be exported to a set of raster and vector image formats. Figure 8.12 is a screenshot of this operation.

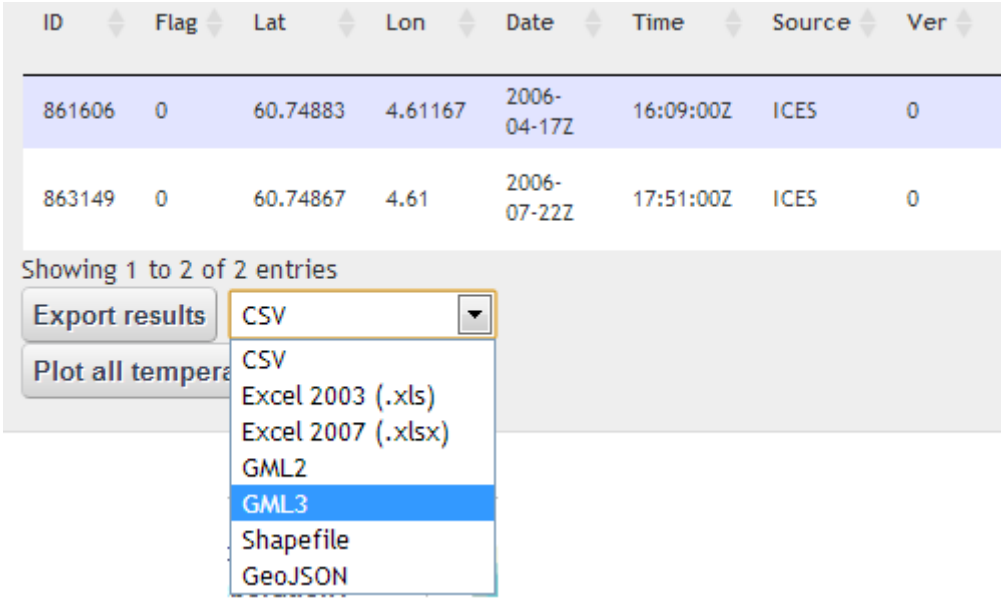


FIGURE 8.11: Exporting query results

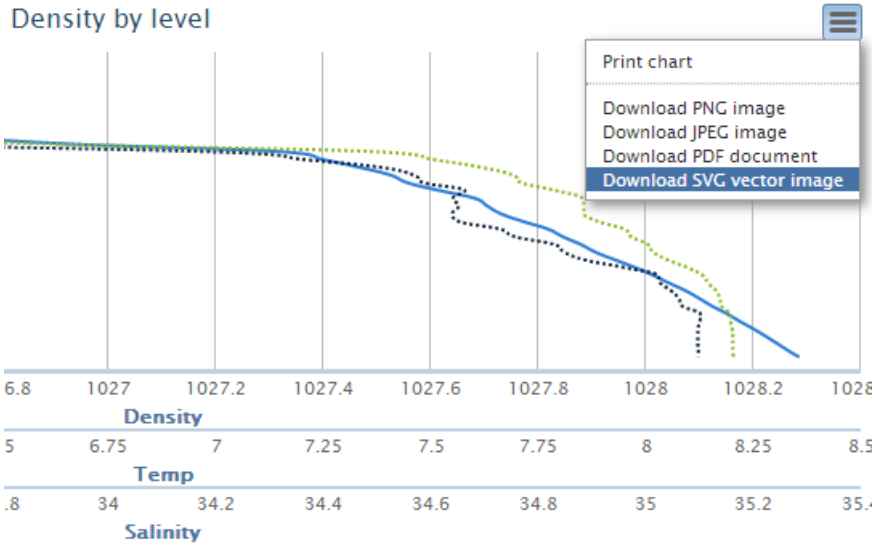


FIGURE 8.12: Exporting graph plots

9 Evaluation

The evaluation of both the work process and the result was identified as sub–goal 6 of this thesis, and is the topic of this chapter. We will evaluate the achievement of all sub–goals, attempt to answer the research questions and finally evaluate our achievement of the overall goal. This process also gives an overview of the final state of our system, which is the basis for identifying possible future work in the next chapter.

9.1 Sub–goals

9.1.1 Research and determine appropriate standards and best practices

Current geospatial and web standards are large and diverse. An overview and evaluation of our employed standards follow below, categorized by components of the software stack. Note that we here consider the functional requirements of the client application largely fulfilled, see Section 9.1.5 for this evaluation.

- Geographic data store

  Standards: SFA

  Data is stored in a SFA compliant database. SFA was suitable to represent our data: the `POINT` class is well suited to store longitude/latitude–tuples. The operations SFA provides were sufficient to realise the functional requirements of our client application.

- Geographic data accessor

  Standards: SFA, OWS (WMS, WFS, WPS), SLD

The data accessor communicates with the SFA data store and correctly handles SFA data types and operations. The data is then made accessible through the WFS and WMS OWS. All data in the underlying database were made available through these protocols, though with the use of a vendor extension. Optimally, WFS should have this extension as an official parameter for us to claim complete compliance. SLD stylings were successfully used to achieve contour plot functionality on the back–end.

- Geographic presentation service

  Standards: W3C (HTML5, CSS3), ECMAScript, OWS (WMS, WFS), GML, GeoJSON

  The client application is built on JavaScript (implementation of ECMAScript) and HTML5/CSS3. These languages were sufficient and suited for realisation of the functional requirements. Useful tools and libraries that follows these standards were employed, this aided development. All communication with the back–end is done using WMS and WFS, and these protocols were sufficient in fulfilling the functional requirements. The GeoJSON encoding format (based on GML) was used internally to represent features, and in communication with the data store. It integrated very in the JavaScript environment.

Overall, both selection of and compliance to standards was successful. It is however possible that other standards that have not been explored may be applicable, for example the *Sensor Observation Service* (SOS) [109, 110]. SOS is a standard that is structured similarly to WFS, but focused on reading and presenting real–time sensor data. Our employed standards have the advantage of being general in purpose, and should be applicable for performing any similar task.

9.1.2 Determine criteria for, and make selections to construct software stack

We were able to construct a software stack to our requirements and selection criteria. This stack is functional and was pleasing to work with during development. PostGIS and GeoServer have been reliable and are quite well documented. On the client side, OpenLayers has been very useful even beyond its role as the map widget: OpenLayers objects and functions are used extensively in the application source code. The remaining JavaScript libraries that were used, while not essential, have greatly simplified the achievement of the functional requirements.

While the goal was achieved, we must also note the availability of other software alternatives. Some of these were presented in Chapter 4 (referenece), others may also exist. We can not guarantee that our stack is the absolutely best or correct choice, but it has served our purpose well.

9.1.3 Transform geographic data to standard–compliant encoding

The structure of the ODB database was initially simple to work with due to it only having a single true geographic column and typical database structure and clear primary key (`absnum`). We successfully transformed this data to a standard compliant database encoding.

9.1.4 Provide data through standardised access methods

Parts of the data are available directly through completely standard WMS/WFS. However, other layers rely on a GeoServer vendor parameter to function properly. This solution was chosen to avoid greatly increasing the complexity of the system. When the vendor parameter is provided, these layers also act exactly like regular layers. See Section 6.3 for more on this challenge and decision.

In conclusion, this goal was largely, but not completely achieved. There may be some combination of database structure and WFS functionality that allows the implementation of our system with full WFS standard compliance; this is a topic for future work.

9.1.5 Develop client application

To assess the achievement of this goal, we will examine the status of the identified requirements and attempt to give an indication of the quality of the underlying software. Below, the requirements of the front–end client application are listed with an assessment of how well each requirement has been fulfilled by the implemented application. The list corresponds to Table 2.2.

FR1: Query of data based on various parameters (bounding box, time, attribute values)

As demonstrated in Section 8.1, querying data based on the described parameters is fully implemented.

FR2: Presentation of features and parameters in tabular form

Figures 8.4 and 8.6 demonstrate tabular presentation for features and parameters, respectively. The tables also have rich functionality: sorting on columns, pagination, search and more courtesy of the DataTables library.

FR3: Presentation of parameter values as graph plots

This functionality is also demonstrated in Figure 8.6. The graph plots are also interactive: users may zoom in on sections and save the plot in several raster and vector formats.

FR4: Interactive (pan, zoom) map widget

A map widget as described as been fully implemented. It can be seen in Figure 7.2 and the implementation is described in Section 7.6.2.

FR5: Map widget/query interactivity: search based on selected area of map

Interactivity is present and implementation is described in Section 7.6.2 and in Code Listing 7.7.

FR6: Plot graph of parameter value for multiple features

This functionality is fully implemented and described in Section 8.2.2.

FR7: Calculate and present (table, graph plot) density for feature

The implementation of the calculation is described in detail in Section 7.7.5 and is displayed in Figure 8.7.

FR8: Interpolate and display contour plots based on parameter values

This support for this functionality is present in the data accessor – this is described in Section 6.4. Basic support is also implemented in the front end, but the functionality has not been fully realised due to difficulties with combining Filter Encoding filters with the rendering transformations of GeoServer.

FR9: Export functionality for all tables and graph plots

All data presented in the client is also available for export. The list of supported formats is large (see Table 6.1). The GUI for exporting query results can be seen on the bottom of Figure 8.1.

9.1.6 Evaluate goal achievement

During this chapter, we thoroughly examine the achievement of our goals.

9.2 Research questions

We now attempt to answer the research questions for the thesis.

- *Is it possible to construct a modern and standard compliant replacement for ODB using open source tools and software?*

  Our constructed system is a *proof–of–concept*: it is indeed possible as demonstrated by our prototype. This research question is formulated towards a specific case, focusing on ODB. A more general variation – on the feasibility of constructing a replacement for systems *similar to* ODB is also interesting. Based on our experience, this question also has a positive answer: the standards and tools we have employed are very general in nature and suited to a large variety of geographic applications. We are also hopeful that the development of OSS GIS will persist and expand.

- *What are the current relevant standards in constructing web–GIS software?*

  The central geospatial standards were determined to be SFA, WMS and WFS. Related standards or substandards are also involved, such as GML, SLD and Filter Encoding. These standards are very general in nature, so more specialised standards (such as SOS) may be better suited for other applications. For each case, research must be made into which standards are best suited, and this may be a question of generality versus specialization.

- *What challenges are associated with modernising GIS systems such as ODB to be web–based and standard compliant, and how may these challenges be approached?*

  If the application to be modernised does not already adhere to standards, they may use a unusual or unique representation. Our initial data structure in particular was quite simple to convert into an SFA encoding, but presenting it through OWSs was a challenge due to the multi–dimensional nature of the non–geographic data. With more complex functionality to replicate or construct, it may not be possible to implement a system relying only on WFS as the communication protocol between back– and front–end without sacrificing quality in, for example, database structure.

  Beyond the data structure and access challenges, we experienced an ordinary software development process. This is made possible due to the existence of good and standard–aware OSS tools that solve problems in the GIS domain. If, for example, JavaScript libraries such as OpenLayers did not exist, the process would have been much more challenging and time consuming.

9.3 Evaluation conclusion – overall goal

The construction of a prototype ODB replacement Web-GIS has largely been successful. We have fulfilled most requirements and achieved our goals. However, the current set of features is relatively small, and not all functional requirements are completely fulfilled. We believe the selected/constructed software stack and data/code structure accommodates for extension of both the offered functionality and underlying infrastructure. Future development is a central topic in the next chapter.

10 Conclusion

In this chapter, we summarise the status of the constructed system and thesis and outline possible future work on the components of the system and the system as a whole. We conclude with a discussion on the work process and learning experience of the author.

10.1 Status summary

A prototype Web–GIS system has been constructed as a replacement for the ageing ODB system. It consists of a *software stack* with PostGIS [42] as a data store, GeoServer [54] as a data accessor and a client implemented in JavaScript with HTML5/CSS3. The client utilises the OpenLayers [58] JavaScript library, as well as other JavaScript utility libraries. The application is compliant with current standards for storing and presenting and communicating geographic data, as well as current standards in web development. The most central geospatial standards employed are SFA [14, 15], WMS [22] and WFS [29].

The utilised software, standards, work process and experiences acquired in the construction of this system system are described and documented in the thesis. As such, the thesis may provide findings and advice useful for carrying out similar or related projects. Work on publishing and presenting marine data (not from ODB, and with dissimilar structure) has been done by the author in parallel with completion of the thesis; techniques and software from the thesis have been successfully re–applied there.

10.2 Future work

The constructed system and its components are at a prototype stage. In this section, we explore possible future work for each components of the system. Some are improvements

that could be implemented immediately, while others are more focused on the long term potential of the system.

10.2.1 Geographic data store

- Database restructuring and optimisation

  The original naming scheme used for tables in the ODB data is slightly redundant. Removing the "st" prefix on all column names would increase readability without any loss of information. Some column names could also be expanded into more immediately readable forms (such as "longitude" instead of "lon"). This is admittedly more a matter of taste than a necessity.

  There are several constraints in our data that are not explicitly expressed, applying them in the database would improve robustness [111]. For example, the `absnum` column of parameter tables is a *foreign key* and this property could be enforced in the database management system.

  Indexes improve the speed of certain queries in database systems [112]. They should be added to the database according to analysis of which operations are most costly when the system is in use. Spatial indexing is especially interesting for the geographic data, and is provided by PostGIS in the form of GiST [113].

- Import full ODB data set

  Only a subset of the original ODB data was imported for development of the system during the thesis work. Importing the remaining station rows, as well as all parameter types and the other excluded tables is an important next step for the system. Work on this has already started by the author and will continue after this thesis is submitted.

- Import new data sets

  New data sets from other sources or types could be added to the system. These data types include remote sensing and model forecast data.

- Harness SFA potential

  We are currently only using the SFA `Point` data type and the client makes relatively simple queries. SFA allows for much more complex functionality (Section A.1 gives some examples). Some functionality that could be implemented: set of `Polygon` objects to mark regions, enabling searching by region (such as "in the Barents Sea"), `LineString` objects between measurements originating from the same ship/station, marking its routes, and more. Implementing such features in the data store mean they could also be present in the presentation service; see Section 10.2.3.

10.2.2 Geographic data accessor

- Filter encoding and rendering transformation cooperation

  The usage of rendering transformations to fulfill FR8 (contour plots) was described in Section 6.4. As mentioned in Section 9.1.5, this functional requirement is currently not fully realised due to challenges with combining GeoServer rendering transformations with Filter Encoding filters. This issue must be remedied, as a contour plot based on *all* available data is both computationally expensive and not very useful. It must be possible to apply a filter, like in the query functionality, before contour plots are realised as a useful functionality.

- Expand range of rendering transformations

  Rendering transformations are work–in–progress in GeoServer. The transformation applied for contour plots should be reconsidered if more options become available. Due to the open source nature of the GeoServer project, it is also possible to implement custom rendering transformation to match exactly the requirements of desired contouring procedure.

10.2.3 Geographic presentation service

- Contour plot GUI support

  Only basic support for contour plots has been added to the client GUI. For a full implementation, the issues with filtering and rendering transformations must be solved (see Section 10.2.2). The contour plot GUI should have options for selecting which parameter and depth to compute plots from, as well as the possibility to customize the graphical aspects of the plot (colours, colour ranges, etc.).

- Data set size consequences

  As mentioned in Section 10.2.1, the complete ODB set should be imported. This will have consequences for the client application. The total set of stations will increase, meaning some limit on the displayed amount of features in the map widget and query GUI may be necessary. This is trivial to implement on the technical side since WMS and WFS have `limit` parameters, but deciding on an appropriate limit is a potential challenge. The GUI should also be updated to accommodate display of more than two parameter tables simultaneously or allowing selection from the set of available table/graph plots.

- Test coverage, GUI testing

A basic framework for and set of unit tests were described in Section 7.8.3. The current test coverage has room for improvements. The system would also benefit from integration and acceptance testing.

10.3 Conclusion

A usable and useful system has been constructed. While at a prototype stage, it has great potential due to a number of advantages over the ODB system. Some functionality that is missing in ODB, such as any related to data export, is already present in the client. The software architecture of the system as a whole has also improved:

- Distributed architecture

  The system has a fully realised client–server model and can (aside from traditional load issues) handle any number of simultaneous clients. The back–end is also distributed (data store and data accessor are independent components). Each individual client handles presentation which lightens server load, while the server is responsible for heavy operations like spatial queries.

- Standard compliance

  A thorough focus on and realisation of standard compliance means the system is highly interoperable with other GIS software. The underlying data structure is SFA compliant, ensuring that it is ready for a greatly extended set of functionality (discussed in Section 10.2.1). The data store is immediately available to any SFA complaint client that can communicate with PostgreSQL. All data may be loaded directly and queried on via WMS and WFS through the data accessor. All data presented in the client is also exportable to several standardised formats, allowing it to be imported into any other compliant GIS system for presentation or analysis.

- Independently usable components

  The multi–layer distributed architecture also produces the advantage of each component being individually usable. Both SFA– and regular SQL operations may be performed directly on data by users with database access. This enables advanced query and analysis. The data accessor presents the data set through OWS interfaces, meaning they are accessible to any compliant client. Layers are easily drawn on other map surfaces or in combination with data from other sources.

- Reusable module components in the client

While the client is built for the specific data set from ODB, the JavaScript module components (especially the utility modules) should be easily reusable in other systems.

The author has learned much from implementing a system based on a large data set and to real–world requirements. It has been especially interesting to work with a full software stack, not just at a particular, isolated layer. Reading the documentation of and designing in compliance to standards has been laborious and challenging, but also a valuable experience.

A

OGC standard examples

This appendix provides some examples of SFA SQL Option queries, as well as example files for several of the OGC standards described in Chapter 3. The query examples are constructed for demonstration purposes, while the file examples are taken directly from the system constructed in the thesis.

A.1 SFA methods

The following examples are of SFA methods as implemented in PostGIS, using the SFA SQL option. The exact naming given to the SFA methods vary among implementing database systems. In PostGIS, all SFA methods are named with a "`ST_`" prefix. These functions are highlighted in purple in the code listings.

A.1.1 Query methods

```
1 SELECT thing
2 FROM sometable
3 WHERE ST_Distance(thing, ST_GeomFromText('POINT(100 200)'))
    < 10
```

Code Listing A.1: SFA SQL Option: distance query

Code Listing A.1 is an example of the SFA `distance` query method. The query will find all rows of `sometable` where the `point` data type column `thing` is within 10 units of the point at *(100, 200)*. The meaning of these coordinates and units depend on the SRS of `thing`. Notice the usage of `geometry_from_text`, an SFA basic method, and the WKT `point` representation.

```
1 SELECT ST_AsText(
2   ST_Intersection(
3     'LINESTRING ( 0 2, 2 0 )', 'LINESTRING ( 0 0, 2 2 )'
4   )
5 )
```

Code Listing A.2: SFA SQL Option: intersection query

Code Listing A.2 is an example of the SFA `intersection` query method. This method checks for intersection between two `geometry` instances, and returns any intersection. A line between *(0, 2)* and *(2, 0)* intersects a line between *(0, 0)* and *(2, 2)* at the point *(1, 1)*. A WKB representation of this point is returned, and then converted to WKT by the `AsText` SFA basic method. The result of this query is "`POINT(1 1)`". Intersections may also be lines, polygons, etc.

A.1.2 Analysis methods

```
1 SELECT *
2 FROM areas
3 WHERE ST_Area(areaGeometry) > 10000
```

Code Listing A.3: SFA SQL Option: area analysis query

Code Listing A.3 is an example of the SFA `area` analysis method. The query will return any row from the table `areas` where the area of the geometry in column `areaGeometry` is larger than 10000 units. The precision and nature of this operation will depend on the underlying SRS and geometry.

A.2 GML example

Code Listing A.4 is a GML file representing a single feature. The feature has an identifier (line 3), a series of non–geographic attributes (lines 4–15) and a geographic attribute (lines 16–20) . This file was produced by exporting the result of a query in the client to GML3.

```
<wfs:FeatureCollection xmlns:cite="http://www.opengeospatial
   .net/cite" xmlns:ogc="http://www.opengis.net/ogc" xmlns:
   gml="http://www.opengis.net/gml" xmlns:xsi="http://www.w3
   .org/2001/XMLSchema-instance" xmlns:xlink="http://www.w3.
   org/1999/xlink" xmlns:ows="http://www.opengis.net/ows"
   xmlns:wfs="http://www.opengis.net/wfs" numberOfFeatures="
   0" timeStamp="2013-05-29T09:31:46.887Z" xsi:
   schemaLocation="http://www.opengeospatial.net/cite http
   ://localhost:8080/geoserver/cite/wfs?service=WFS&version
   =1.0&request=DescribeFeatureType&typeName=cite%3Afloats
   http://www.opengis.net/wfs http://localhost:8080/
   geoserver/schemas/wfs/1.1.0/wfs.xsd">
  <gml:featureMembers>
    <station gml:id="station.862365">
    <stflag>0</cite:stflag>
    <stdate>2006-04-13Z</stdate>
    <sttime>10:46:00Z</sttime>
    <stsource>ICES</stsource>
    <stversion>0</stversion>
    <stcountryname>NORWAY</stcountryname>
    <stvesselname>G.O. SARS (LLZG)</stvesselname>
    <stdepthsource>842</stdepthsource>
    <stlastlevel>822</stlastlevel>
    <stdepthgrid>851</stdepthgrid>
    <stdepthgridmin>767</stdepthgridmin>
    <stdepthgridmax>903</stdepthgridmax>
    <stpoint>
      <gml:Point srsDimension="2" srsName="urn:x-ogc:def:crs
   :EPSG:4326">
        <gml:pos>61.23983 -7.75033</gml:pos>
      </gml:Point>
    </stpoint>
  </station>
  </gml:featureMembers>
</wfs:FeatureCollection>
```

Code Listing A.4: GML example: single feature

A.3 GeoJSON example

Code Listing A.5 is a GeoJSON file representing the same feature as the GML of Section A.2. The client application handles all features in the GeoJSON format internally.

```
{
  "type":"FeatureCollection",
  "features":[
    {
      "type":"Feature",
      "id":"station.862365",
      "geometry":{
        "type":"Point",
        "coordinates":[
          61.23983,
          -7.75033
        ]
      },
      "geometry_name":"stpoint",
      "properties":{
        "stflag":0,
        "stdate":"2006-04-13Z",
        "sttime":"10:46:00Z",
        "stsource":"ICES",
        "stversion":0,
        "stcountryname":"NORWAY",
        "stvesselname":"G.O. SARS (LLZG)",
        "stdepthsource":842,
        "stlastlevel":822,
        "stdepthgrid":851,
        "stdepthgridmin":767,
        "stdepthgridmax":903
      }
    }
  ],
  "crs":{
    "type":"EPSG",
    "properties":{
      "code":"4326"
    }
  }
}
```

Code Listing A.5: GeoJSON example: single feature

A.4 Filter Encoding example

Code Listing A.6 is a Filter Encoding XML file. The filter described has the following rules:

1. Feature must be within the bounding box defined by the GML `box` *60,2 62,4*.
2. Feature must have the property `stdate` with a lexicographic value between the strings "2006–01–01" (lower boundary) and "2006–02–01" (upper boundary).

This file was generated by the `Fiter` module (see Section 7.7.4) for a query performed in the client (demonstrated in Section 8.1) within a bounding box and for a date between the 1. of January and 1. of February of 2006.

```
<ogc:Filter xmlns:ogc="http://www.opengis.net/ogc">
  <ogc:And>
    <ogc:BBOX>
      <ogc:PropertyName>stpoint</ogc:PropertyName>
      <gml:Box xmlns:gml="http://www.opengis.net/gml">
        <gml:coordinates decimal="." cs="," ts=" ">60,2 62,4
   </gml:coordinates>
      </gml:Box>
    </ogc:BBOX>
    <ogc:PropertyIsBetween>
      <ogc:PropertyName>stdate</ogc:PropertyName>
      <ogc:LowerBoundary>
        <ogc:Literal>2006-01-01</ogc:Literal>
      </ogc:LowerBoundary>
      <ogc:UpperBoundary>
        <ogc:Literal>2006-02-01</ogc:Literal>
      </ogc:UpperBoundary>
    </ogc:PropertyIsBetween>
  </ogc:And>
</ogc:Filter>
```

Code Listing A.6: Filter Encoding example: bounding box and date filtering

A.5 SLD example

Code Listing A.7 is the SLD applied to features on the map widget in the client application. The SLD produces a small, red circle with a black outline for each feature. The result can be seen on the screenshot of Figure 7.2. This SLD was partially based on the default SLD selection from GeoServer.

Another SLD example can be found in Appendix B, but it utilises GeoServer–specific extensions that are not part of the standard.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <StyledLayerDescriptor version="1.0.0"
3  xsi:schemaLocation="http://www.opengis.net/sld
       StyledLayerDescriptor.xsd"
4  xmlns="http://www.opengis.net/sld"
5  xmlns:ogc="http://www.opengis.net/ogc"
6  xmlns:xlink="http://www.w3.org/1999/xlink"
7  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
8    <NamedLayer>
9      <Name>default_point</Name>
10     <UserStyle>
11       <Title>Station point</Title>
12       <FeatureTypeStyle>
13         <Rule>
14           <Name>rule1</Name>
15           <Title>Station</Title>
16           <Abstract>Small red circle with black outline</
       Abstract>
17           <PointSymbolizer>
18             <Graphic>
19               <Mark>
20                 <WellKnownName>circle</WellKnownName>
21                 <Fill>
22                   <CssParameter name="fill">#33FF33</
       CssParameter>
23                 </Fill>
24                 <Stroke>
25                   <CssParameter name="stroke">#000000</
       CssParameter>
```

```
26                    <CssParameter name="stroke-width">1</
   CssParameter>
27                  </Stroke>
28                </Mark>
29                <Size>3</Size>
30              </Graphic>
31            </PointSymbolizer>
32          </Rule>
33        </FeatureTypeStyle>
34      </UserStyle>
35    </NamedLayer>
36 </StyledLayerDescriptor>
```

Code Listing A.7: SLD example: red circle with black outline

B

Barnes Surface Interpolation in GeoServer

This appendix provides an SLD file for the GeoServer rendering transformation described in Section 6.4.

B.1 Rendering transformation SLD file

```
<?xml version="1.0" encoding="UTF-8"?>
<StyledLayerDescriptor version="1.0.0"
xsi:schemaLocation="http://www.opengis.net/sld
   StyledLayerDescriptor.xsd"
xmlns="http://www.opengis.net/sld"
xmlns:ogc="http://www.opengis.net/ogc"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <NamedLayer>
    <Name>Barnes surface</Name>
    <UserStyle>
      <Title>Barnes Surface</Title>
      <Abstract>A style that produces a Barnes surface
   interpolation using a rendering transformation</Abstract>
      <FeatureTypeStyle>
        <Transformation>
          <ogc:Function name="gs:BarnesSurface">
            <ogc:Function name="parameter">
              <ogc:Literal>data</ogc:Literal>
            </ogc:Function>
            <ogc:Function name="parameter">
```

```
20              <ogc:Literal>valueAttr</ogc:Literal>
21              <ogc:Literal>value</ogc:Literal>
22            </ogc:Function>
23            <ogc:Function name="parameter">
24              <ogc:Literal>scale</ogc:Literal>
25              <ogc:Literal>1</ogc:Literal>
26            </ogc:Function>
27            <ogc:Function name="parameter">
28              <ogc:Literal>convergence</ogc:Literal>
29              <ogc:Literal>0.3</ogc:Literal>
30            </ogc:Function>
31            <ogc:Function name="parameter">
32              <ogc:Literal>passes</ogc:Literal>
33              <ogc:Literal>3</ogc:Literal>
34            </ogc:Function>
35            <ogc:Function name="parameter">
36              <ogc:Literal>minObservations</ogc:Literal>
37              <ogc:Literal>2</ogc:Literal>
38            </ogc:Function>
39            <ogc:Function name="parameter">
40              <ogc:Literal>maxObservationDistance</ogc:
    Literal>
41              <ogc:Literal>15</ogc:Literal>
42            </ogc:Function>
43            <ogc:Function name="parameter">
44              <ogc:Literal>pixelsPerCell</ogc:Literal>
45              <ogc:Literal>8</ogc:Literal>
46            </ogc:Function>
47            <ogc:Function name="parameter">
48              <ogc:Literal>outputBBOX</ogc:Literal>
49              <ogc:Function name="env">
50                <ogc:Literal>wms_bbox</ogc:Literal>
51              </ogc:Function>
52            </ogc:Function>
53            <ogc:Function name="parameter">
54              <ogc:Literal>outputWidth</ogc:Literal>
55              <ogc:Function name="env">
56                <ogc:Literal>wms_width</ogc:Literal>
57              </ogc:Function>
```

```
58            </ogc:Function>
59            <ogc:Function name="parameter">
60              <ogc:Literal>outputHeight</ogc:Literal>
61              <ogc:Function name="env">
62                <ogc:Literal>wms_height</ogc:Literal>
63              </ogc:Function>
64            </ogc:Function>
65          </ogc:Function>
66        </Transformation>
67        <Rule>
68          <RasterSymbolizer>
69            <!-- specify geometry attribute of input to pass
     validation -->
70            <Geometry><ogc:PropertyName>stpoint</ogc:
    PropertyName></Geometry>
71            <Opacity>1</Opacity>
72            <ColorMap type="ramp">
73              <ColorMapEntry color="#FFFFFF" quantity="0"
    label="novalue" opacity="0"/>
74              <ColorMapEntry color="#000000" quantity="26.0"
     label="20" opacity="0.8"/>
75              <ColorMapEntry color="#FF0000" quantity="30.0"
     label="35" opacity="0.8"/>
76              <ColorMapEntry color="#FFFFFF" quantity="36.0"
     label="35" opacity="0.8"/>
77              <ColorMapEntry color="#FFFFFF" quantity="50"
    label="novalue" opacity="0"/>
78            </ColorMap>
79          </RasterSymbolizer>
80        </Rule>
81      </FeatureTypeStyle>
82    </UserStyle>
83  </NamedLayer>
84</StyledLayerDescriptor>
```

Code Listing B.1: Barnes Surface Interpolation SLD

C

WebFeatureService module

C.1 Description

This module in particular was selected as it adequately showcases the use of the Revealing Pattern module while still being quite small in size and scope. It also shows off some code style choices. As such, it is a good representative for the code layout and structure of entire front–end application.

The role of the WebFeatureService module is described in Section 7.7.2. In summary, it is an abstraction of a Web Feature Service (see Section 3.5) that encapsulates complexity to provide easy access to WFS operations for other components of the application.

The component is designed to be modular to the point that it could be used in any application requiring WFS services. Its depends only on external libraries, not other components of the system it is currently used in. As of writing, the module only implements the WFS `GetFeature` operation, but it is designed to be extensible. Implementing, for example, the `GetCapabilties` operation would only require the addition of another function, and the asynchronous HTTP GET–request function already present in the module could be reused as–is. The functionality for getting the parameters used in the previous request would not need editing to work also with `GetCapabilties` operations.

C.2 Source code annotations

The source code is annotated in top–to–bottom order. Concepts used that have been introduced previously will be annotated only with a reference to the original discussion.

- Line 1: Namespace management, see Section 7.3.1.

```
var myNamespace = myNamespace || {};

myNamespace.WebFeatureService = (function($, OL) {
  "use strict";

  var server = "http://localhost:8080/geoserver/cite/wfs",
   previousRequestParameters = null;

  // fires an asyn. HTTP GET request to server
  function asyncGetRequest(parameters, callback) {
    previousRequestParameters = parameters;

    OL.Request.GET({
      url : server,
      params : parameters,
      callback : callback
    });
  }

  // fires a GetFeature WFS request to server
  function getFeature(extraParameters, callback) {
    // some default parameters that will be set
   automatically if not overridden in extraParams
    var parameters = {
      REQUEST : "GetFeature",
      SERVICE : "WFS",
      VERSION : "1.0",
      OUTPUTFORMAT : "json"
    };

    // extend provided parameters onto default parameters,
   make request
    asyncGetRequest($.extend(parameters, extraParameters),
   callback);
  }

  function getPreviousRequestParameters() {
    return previousRequestParameters;
  }

  // public interface
  return {
    getFeature : getFeature,
    getPreviousRequestParameters :
   getPreviousRequestParameters
  };

}(jQuery, OpenLayers));
```

Code Listing C.1: WebFeatureService module

- Line 3, 43: Declaring module, adding it to namespace. The design pattern used here, Revealing module, is described in Section 7.3.4. `$` and `OL` become function–local aliases for the objects passed to the function (`jQuery` and `OpenLayers`, respectively). This dependency management technique is described in Section 7.3.5.

- Line 4: Initiates `strict` execution mode for the function, if supported by the runtime environment. Strict mode is feature in the newer ECMAScript language specification that enables an execution mode which addresses a set of potential issues [114]. In environments that don't support or recognize strict mode, this line will have no effect. Strict mode is used in all modules of the application.

- Line 6: Private variables of the module. This module is intended to work as a singleton, being the sole WFS access point for the entire application. If the intention was to instantiate several WebFeatureService modules, the `server` variable is likely to have been set in the constructor function.

- Line 8–17: Private function. Uses OpenLayers' `get` function to issue an AJAX HTTP GET–request to `server`. Parameters and a callback function are input to the function as parameters. The callback will be invoked with the result of the `XMLHTTPRequest` as a parameter once the request reaches `ReadyState` 4 (finished). Notice that `WebFeatureService` is not (and does not need to be) aware of what the callback function is or how it works. This encourages loose coupling between this module and other components of the system.

- Line 19–31: Private function. A utility function that makes WFS `GetFeature` requests easier to issue by providing some default parameters. For example, the `SERVICE` parameter (line 24) must always be included in a WFS request, but always has the value "WFS". Setting it as a default value here reduces clutter other places when a WFS request is invoked.

  On line 30, jQuery's `extend()` function is used to add any parameters provided as the function parameter (`extraParameters`) to the `parameters` object declared on lines 22-27. The resulting object and the `callback` function parameter are passed as arguments to `asyncGetRequest()` (discussed previously). Notice the absence of `this` to reference the function. It is not a property of an object, as in traditional JavaScript object–orientation, but available directly due to function scope.

- Line 33–35: Private function. Simple "getter" function for the private `previousRequestParameters` variable.

- Line 37–41: Return value for the function: public interface of the module. In object literal notation, public references to a selection of private functions is given. The

private `getFeature()` and `getPreviousRequestParameters()` functions are now available to the outside world through public references.

Bibliography

[1] OpenPlans. (2013) OpenGeo initiative. [Online]. Available: http://opengeo.org/

[2] The Latex project. (2010, Jan.) LaTeX. [Online]. Available: https://www.latex-project.org

[3] D. Graffox. (2009) IEEE Citation Reference. [Online]. Available: http://www.ieee.org/documents/ieeecitationref.pdf

[4] P. Brachet. (2013) Texmaker. [Online]. Available: http://www.xm1math.net/texmaker

[5] yWorks. (2013, Apr.) yEd Graph Editor. [Online]. Available: http://www.yworks.com/en/products_yed_about.html

[6] The GNOME Project. (2011, Dec.) Dia. [Online]. Available: https://live.gnome.org/Dia

[7] D. Walton. (2012, May) ckwnc. [Online]. Available: http://www.ckwnc.com/

[8] S. Gunn. (2010) Postgraduate thesis style for PhD. [Online]. Available: http://users.ecs.soton.ac.uk/srg/softwaretools/document/templates/

[9] S. Patel. (2010, Aug.) LaTeX thesis template. [Online]. Available: http://www.sunilpatel.co.uk/thesis-template/

[10] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind, *Geographic Information Systems and Science*, 2nd ed. John Wiley & Sons, 2005, p. 19.

[11] The Web Standards Project. (2013) WaSP: Fighting for standards. [Online]. Available: http://www.webstandards.org/about/mission/

[12] Open Geospatial Consortium, Inc. (2013) About OGC. [Online]. Available: http://www.opengeospatial.org/ogc

[13] ——. (2013) Compliance Testing. [Online]. Available: http://www.opengeospatial.org/compliance/

[14] ——, "OpenGIS Implementation Standard for Geographic information – Simple feature access – Part 1: Common architecture," *OpenGIS Implementation Standard*, May 2011. [Online]. Available: http://www.opengeospatial.org/standards/sfa

[15] ——, "OpenGIS Implementation Standard for Geographic information – Simple feature access – Part 2: SQL option," *OpenGIS Implementation Standard*, August 2010. [Online]. Available: http://www.opengeospatial.org/standards/sfs

[16] International Organisation for Standardization, "Geographic information – Simple feature access – Part 1: Common architecture," *ISO19125–1:2004*, 2004.

[17] ——, "Geographic information – Simple feature access – Part 2: SQL option," *ISO19125–2:2004*, 2004.

[18] Object Management Group. (2011, August) OMG Unified Modeling Language (OMG UML), Infrastructure. Version 2.4.1.

[19] ——. (2011, August) OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1.

[20] Open Geospatial Consortium, Inc., "OpenGIS Simple Features Specification For OLE/COM," May 1999. [Online]. Available: http://www.opengeospatial.org/standards/sfo

[21] ——, "OpenGIS Simple Features Specification For CORBA," June 1999. [Online]. Available: http://www.opengeospatial.org/standards/sfc

[22] ——, "OpenGIS Web Map Server Implementation Specification," *OpenGIS Implementation Specification*, March 2006. [Online]. Available: http://www.opengeospatial.org/standards/wms

[23] ——, "OpenGIS Web Map Services – Profile for EO Products," *OGC Best Practice*, November 2009. [Online]. Available: http://www.opengeospatial.org/standards/wms

[24] ——, "DGIWG WMS 1.3 Profile and systems requirements for interoperability for use within a military environment," *OGC Best Practice*, September 2009. [Online]. Available: http://www.opengeospatial.org/standards/wms

[25] ——, "OGC Web Map Service – Proposed Animation Service Extension," *OGC Discussion Paper*, July 2005. [Online]. Available: http://www.opengeospatial.org/standards/wms

[26] J. L. Beaujardiere, "Web Map Service Implementation Specification – Part 2: XML for Requests using HTTP POST," *OGC Discussion Paper*, April 2002. [Online]. Available: http://www.opengeospatial.org/standards/wms

[27] Open Geospatial Consortium, Inc., "WMS Change Request: Support for WSDL and SOAP," *OGC Discussion Paper*, April 2005. [Online]. Available: http://www.opengeospatial.org/standards/wms

[28] T. Berners-Lee, *Uniform Resource Identifier (URI): Generic Syntax.* Internet Engineering Task Force RFC, 2005, p. 11. [Online]. Available: http://tools.ietf.org/html/rfc3986#section-2.1

[29] Open Geospatial Consortium, Inc., "OpenGIS Web Feature Service 2.0 Interface Standard," *OpenGIS Implementation Standard*, nov 2010. [Online]. Available: http://www.opengeospatial.org/standards/wfs

[30] ——, "Web Feature Service Implementation Specification," *OpenGIS Implementation Specification*, may 2005. [Online]. Available: http://www.opengeospatial.org/standards/wfs

[31] ——, "OWS5: OGC Web feature service, core and extensions," *Discussion Paper*, sep 2008. [Online]. Available: http://www.opengeospatial.org/standards/wfs

[32] ——, "Gazetteer Service – Application Profile of the Web Feature Service Best Practice," *OGC Best Practice*, feb 2012. [Online]. Available: http://www.opengeospatial.org/standards/wfs

[33] International Organisation for Standardization, "Geographic Information – Web Feature Service," *ISO1942:2010*, 2010.

[34] Open Geospatial Consortium, Inc., "OpenGIS Geography Markup Language (GML) Encoding Standard," *OpenGIS Standard*, August 2007. [Online]. Available: http://www.opengeospatial.org/standards/gml

[35] H. Butler et al. (2008, June) The GeoJSON Format Specification. [Online]. Available: http://www.geojson.org/geojson-spec.html

[36] D. Crockford, *The application/json Media Type for JavaScript Object Notation (JSON).* Internet Engineering Task Force RFC, July 2006. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt?number=4627

[37] Open Geospatial Consortium, Inc., "Styled Layer Descriptor profile of the Web Map Service Implementation Specification," *OGC Implementation Specification*, June 2007. [Online]. Available: http://www.opengeospatial.org/standards/sld

[38] ——, "OGC OWS–6 Styled Layer Descriptor (SLD) Changes," *Public Engineering Report*, September 2009. [Online]. Available: http://www.opengeospatial.org/standards/sld

[39] North Bridge. (2012) Open Source 2012 Survey Results. [Online]. Available: http://northbridge.com/2012-open-source-survey

[40] R. Stallman. (2013, March) Why open source misses the point of free software. [Online]. Available: http://www.gnu.org/philosophy/open-source-misses-the-point.html

[41] Open Geospatial Consortium, Inc. (2013) Registered Products. [Online]. Available: http://www.opengeospatial.org/resource/products

[42] PostGIS contributors. (2013) Postgis. [Online]. Available: http://postgis.net/

[43] The PostgreSQL Global Development Group. (2013) Postgresql. [Online]. Available: http://www.postgresql.org/

[44] PostGIS Contributors, *PostGIS Manual*, 2nd ed. Online Manual, 2013, p. 24. [Online]. Available: http://postgis.net/stuff/postgis-2.0.pdf

[45] The PostgreSQL Global Development Group. (2013) Postgresql advantages. [Online]. Available: http://www.postgresql.org/about/advantages/

[46] R. O. Obe and L. S. Hsu, *PostGIS In Action*. Manning publications Co., 2011, pp. 9–15.

[47] The PostgreSQL Global Development Group. (2013) PostgreSQL Documentation – Copy. [Online]. Available: http://www.postgresql.org/docs/9.2/static/sql-copy.html

[48] The Oracle Corporation. (2013) Why MySQL? [Online]. Available: http://www.mysql.com/why-mysql/

[49] ——. (2013) MySQL 5.7 Manual – Introduction to MySQL Spatial Support. [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/gis-introduction.html

[50] Microsoft. (2013) Spatial Data (SQL Server). [Online]. Available: http://msdn.microsoft.com/en-us/library/bb933790(v=sql.110).aspx

[51] The Oracle Corporation. (2013) Oracle Spatial and Graph. [Online]. Available: http://www.oracle.com/us/products/database/options/spatial/overview/index.html

[52] MapServer Contributors. (2013) MapServer. [Online]. Available: http://mapserver.org

[53] ——. (2013) About MapServer. [Online]. Available: http://mapserver.org/about.html

[54] GeoServer Contributors. (2012, October) GeoServer 2.2. [Online]. Available: http://geoserver.org

[55] ——. (2012) GeoServer 2.2 User Manual – MySQL. [Online]. Available: http://docs.geoserver.org/2.2.0/user/data/database/mysql.html

[56] ——. (2009) OGC Compliance. [Online]. Available: http://geoserver.org/display/GEOS/OGC+Compliance

[57] V. Agafonkin and CloudMade. (2013) Leaflet. [Online]. Available: http://leafletjs.com

[58] OpenLayers Contributors. (2013) OpenLayers. [Online]. Available: http://openlayers.org

[59] M. Zalewski, *The Tangled Web.* No Starch Press, 2012, p. 127.

[60] OpenPlans. (2013) OpenGeo Suite. [Online]. Available: http://opengeo.org/products/suite/

[61] The jQuery Foundation. (2013) jQuery. [Online]. Available: http://jquery.com/

[62] ——. (2013) jQuery User Interface. [Online]. Available: http://jqueryui.com/

[63] SpryMedia. (2013) DataTables. [Online]. Available: http://www.datatables.net/

[64] Highsoft Solutions AS. (2013) Highcharts JS. [Online]. Available: http://www.highcharts.com/

[65] National Imagery and Mapping Agency, "Department Of Defense World Geodetic System 1984," 2000. [Online]. Available: http://earth-info.nga.mil/GandG/publications/tr8350.2/wgs84fin.pdf

[66] H. Butler, C. Schmidt, D. Springmeyer, and J. Livni. (2007) Spatial Reference – EPSG projection 4326 – WGS 84. [Online]. Available: http://spatialreference.org/ref/epsg/4326/

[67] ESRI, "ESRI Shapefile Technical Description," *ESRI White Paper*, July 1998. [Online]. Available: http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf

[68] R. Elmasri and S. Navathe, *Database Systems*, 6th ed. Pearson Education, Inc., 2011, pp. 949–952.

[69] The Open Web Application Security Project. (2010) OWASP Top 10 – 2010. [Online]. Available: http://owasptop10.googlecode.com/files/OWASP%20Top%2010%20-%202010.pdf

[70] ——. (2013) OWASP Top 10 – 2013. [Online]. Available: https://www.owasp.org/index.php/Top_10_2013-Top_10

[71] GeoServer Contributors. (2013) GeoServer 2.2 User Manual – SLD Extensions in GeoServer – Rendering Transformations. [Online]. Available: http://docs.geoserver.org/stable/en/user/styling/sld-extensions/rendering-transform.html

[72] Open Geospatial Consortium, Inc., "OpenGIS Web Processing Service," *OpenGIS Standard*, June 2007. [Online]. Available: http://www.opengeospatial.org/standards/wps

[73] S. L. Barnes, "A Technique for Maximizing Details in Numerical Weather Map Analysis," *J. Appl. Meteor*, vol. 3, pp. 369–409, August 1964.

[74] GeoServer Contributors. (2013) Excel WFS Output Format. [Online]. Available: http://docs.geoserver.org/stable/en/user/extensions/excel.html

[75] R. Martin, J. W. Newkirk, and R. S. Koss, *Agile Software Development.* Pearson Education, Inc., 2012.

[76] The Eclipse Foundation. (2013, Jun.) Eclipse IDE for Java Enterprise Edition. [Online]. Available: http://www.eclipse.org

[77] Liferay, Inc. (2013) Liferay IDE. [Online]. Available: http://www.liferay.com/community/liferay-projects/liferay-ide/overview

[78] Appcelerator, Inc. (2011) Aptana Studio 3. [Online]. Available: http://aptana.com

[79] Google Inc. (2013, Apr.) Chrome DevTools. [Online]. Available: https://developers.google.com/chrome-developer-tools

[80] A. Abdelnur, E. Chien, and S. Hepper, "JSR–000168 Portlet Specification," *Java Community Process – Final Release*, 2003. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr168/index.html

[81] S. .Hepper, "JSR–000268 Portlet Specification," *Java Community Process – Final Release*, 2008. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr268/index.html

[82] Liferay, Inc. (2013) Liferay Portal. [Online]. Available: http://liferay.com

[83] Git Contributors. (2013) Git. [Online]. Available: http://www.git-scm.com/

[84] M. Zalewski, *The Tangled Web*. No Starch Press, 2012, ch. 6.

[85] ——, *The Tangled Web*. No Starch Press, 2012, ch. 9.

[86] ——, *The Tangled Web*. No Starch Press, 2012, p. 143.

[87] OpenLayers Contributors. (2013) OpenLayers FAQ – Why do I need a ProxyHost? [Online]. Available: http://trac.osgeo.org/openlayers/wiki/FrequentlyAskedQuestions#ProxyHost

[88] L. Ullmann, *Modern JavaScript*. Peachpit Press, 2012, ch. 14.

[89] D. Crockford, *JavaScript: The Good Parts*. O'Reilly Media, Yahoo Press, May 2008, ch. A.

[90] C. Larman, *Applying UML and Patterns*, 3rd ed. Pearson Education, Inc., 2005, pp. 278–280.

[91] M. Fowler. (2004, September) Lambdas. [Online]. Available: http://martinfowler.com/bliki/Lambda.html

[92] ECMA International, "ECMAScript language specification," *ECMA standard*, pp. 31–34, 2011. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[93] A. Osmani, *Learning JavaScript Design Patterns*. O'Reilly Media, July 2012, pp. 26–32.

[94] R. Martin, J. W. Newkirk, and R. S. Koss, *Agile Software Development*. Pearson Education, Inc., 2012, ch. 16.

[95] A. Osmani, *Learning JavaScript Design Patterns*. O'Reilly Media, July 2012, pp. 37–38.

[96] C. Heilmann. (2007, August) Again With the Module Pattern – Reveal Something To The World. [Online]. Available: http://christianheilmann.com/2007/08/22/again-with-the-module-pattern-reveal-something-to-the-world/

[97] World Wide Web Consortium (W3C). (2013) HTML4 Index of Elements. [Online]. Available: http://www.w3.org/TR/html4/index/elements.html

[98] A. Kesteren and S. Pieters, "HTML5 differences from HTML4," *W3C Working Draft*, oct 2012. [Online]. Available: http://www.w3.org/TR/html5-diff/

[99] R. Martin, J. W. Newkirk, and R. S. Koss, *Agile Software Development*. Pearson Education, Inc., 2012, ch. 8.

[100] Leithead et al., "Document Object Model (DOM) Level 3 Events Specification," *W3C Working Draft*, 2012. [Online]. Available: http://www.w3.org/TR/DOM-Level-3-Events/

[101] UNESCO, "Tenth report of the joint panel on oceanographic tables and standards," *UNESCO technical papers in marine science*, vol. 36, pp. 17–20, 1981.

[102] I. Hickson, "Web Workers," *W3C Candidate Recommendation*, May 2012. [Online]. Available: http://www.w3.org/TR/workers/

[103] B. Wilkinson and M. Allen, *Parallel Programming*, 2nd ed. Pearson Education, Inc., 2005, ch. 3.

[104] World Wide Web Consortium (W3C). (2013) Markup Validation Service. [Online]. Available: http://validator.w3.org/

[105] ——. (2013) CSS Validation Service. [Online]. Available: http://jigsaw.w3.org/css-validator

[106] D. Crockford. (2012) JSLint. [Online]. Available: http://www.jslint.com/lint.html/

[107] A. Spillner et al., *Software Testing Foundations*, 3rd ed. Rocky Nook Inc., 2011, pp. 43–50.

[108] The jQuery Foundation. (2013) QUnit JavaScript Unit Testing Framework. [Online]. Available: http://qunitjs.com/

[109] Open Geospatial Consortium, Inc., "Sensor Observation Service," *OpenGIS Implementation Standard*, Octoberr 2007. [Online]. Available: http://www.opengeospatial.org/standards/sos

[110] ——, "OGC Sensor Observation Service Interface Standard," *OpenGIS Implementation Standard*, April 2012. [Online]. Available: http://www.opengeospatial.org/standards/sos

[111] R. Elmasri and S. Navathe, *Database Systems*, 6th ed. Pearson Education, Inc., 2011, pp. 63–71.

[112] ——, *Database Systems*, 6th ed. Pearson Education, Inc., 2011, ch. 17.

[113] PostGIS Contributors. (2012) PostGIS 2.0 Manual – GiST Indexes. [Online]. Available: http://postgis.net/docs/manual-2.0/using_postgis_dbmanagement.html#gist_indexes

[114] ECMA International, "ECMAScript language specification," *ECMA standard*, pp. 235–236, 2011. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf