



Design and development of EYK

A data management system for geographically distributed data sources

Bjarki Ásbjarnarson



Faculty of Industrial Eng., Mechanical Eng. and Computer Science
University of Iceland
2014

Design and development of EYK, a data management system for geographically distributed data sources

Bjarki Ásbjarnarson

60 ECTS thesis submitted in partial fulfillment of a
Magister Scientiarum degree in Software Engineering

Advisors

Helmut Neukirchen
Steinn Guðmundsson

Faculty Representative
Heiðar Einarsson

Faculty of Industrial Eng., Mechanical Eng. and Computer Science
School of Engineering and Natural Sciences
University of Iceland
Reykjavik, May 2014

Design and development of EYK, a data management system for geographically distributed data sources

Design and development of EYK

60 ECTS thesis submitted in partial fulfillment of a M.Sc. degree in Software Engineering

Copyright © 2014 Bjarki Ásbjarnarson

All rights reserved

Faculty of Industrial Eng., Mechanical Eng. and Computer Science

School of Engineering and Natural Sciences

University of Iceland

VRII, Hjardarhagi 2-6

107, Reykjavik, Reykjavik

Iceland

Telephone: 525 4000

Bibliographic information:

Bjarki Ásbjarnarson, 2014, Design and development of EYK, a data management system for geographically distributed data sources, M.Sc. thesis, Faculty of Industrial Eng., Mechanical Eng. and Computer Science, University of Iceland.

Printing: Háskólaprent, Fálkagata 2, 107 Reykjavík

Reykjavik, Iceland, May 2014

Dedication:

I dedicate this thesis to my fiancé and our boys. Without their support and encouragement I would have given up on this dream for long since.

Abstract

A custom data management system is currently in use at Reykjavik Energy. While being an excellent data management system, the system has some usability issues. These issues are addressed and improved upon in the design of EYK.

EYK is a web application which is designed as a framework. EYK shows assets at their geographical locations on an interactive map. Assets are grouped by their types where each type has its own plugin in the framework. When an asset is selected on the map, its data is shown by a custom collection of apps. Each app is a module which can be added or removed from the system without interfering with the rest of it. Each module provides a single, isolated functionality. Some manage a specific type of data while others visualize or process data. When customizing their system, customers select which modules are used with which types of assets.

To connect to other databases EYK employs a second type of plugin. This plugin is used to interface with databases or data sources and scrape their data into EYK's own database. When specifying their plugins, customers can choose whether data is stored in EYK or if an interface is made to an external database. Data from a database interface is kept up to date by scraping data with a defined frequency.

The framework of EYK was successfully developed during this project along with a few example plugins. These are demonstrated at <http://eyk.hugfimi.is>.

Preface

The idea of EYK was conceived when I was working a student job at an Icelandic energy company, Reykjavik Energy. My job was to create small programs to accept raw data in a specific format and import it into a database. These programs also gave access to the imported data, plotted it or exported it in raw format. The collection of these programs, called Gagnor, still serves as data management software for researches and projects.

The Gagnor collection has its limitations, which is why I decided to do my master's thesis on the design and development of EYK, the next geographical data management system.

Contents

List of Figures	xi
List of Tables	xiii
Abbreviations	xv
Acknowledgments	xvii
1. Introduction	1
2. System background and requirements	3
2.1. Current system	3
2.1.1. Limitations of Gagnor	4
2.2. System requirements	6
3. System outline	7
3.1. Asset identification	7
3.2. Assigning functionality to assets	9
3.2.1. Modules	10
3.3. Database interfaces	10
3.4. Accessibility	11
3.5. Sales model	11
4. Technology used	15
4.1. Web applications	15
4.1.1. Static media	15
4.1.2. Cascading style sheets	16
4.1.3. JavaScript	17
4.1.4. Frontend libraries	18
4.2. Web application framework	19
4.2.1. Django	19
4.2.2. Database	24
4.2.3. Task queue	25
4.3. Diagram notation	26
5. Architectural design	27
5.1. Map	28
5.2. Oats	29
5.2.1. OAT plugins	30
5.2.2. The asset page, <code>oat.html</code>	31
5.2.3. Module callbacks	31

5.3. Core	33
5.4. Modules	34
5.4.1. Structure	35
5.5. Database Interfaces	35
6. Implementation	37
6.1. Map	37
6.1.1. Map page	38
6.1.2. Markers and maker groups	41
6.2. Oats	44
6.2.1. Viewing an asset	44
6.2.2. Displaying a module	47
6.2.3. Providing modules with data callback	49
6.3. Core	49
6.3.1. Abstract Models	50
6.3.2. Data Types	51
6.4. Modules	53
6.4.1. Properties module	53
6.4.2. Plot module	55
7. Example system implementation	63
7.1. Database interface	63
7.2. OAT plugin	64
7.2.1. Properties module	65
7.2.2. Plot module	65
8. Results	69
8.1. Customer options	70
8.2. Selectable functionality	70
8.3. Improvements over Gagnor	71
8.4. Further requirements	72
9. Conclusions	73
9.1. Improvements	73
9.2. Outlook	73
9.2.1. Release 1.0	74
9.2.2. Release 2.0	74
Bibliography	75
Appendices	77
A. Diagrams	79
B. Contributions	81

List of Figures

2.1. Example plot, made by the steam purity program	4
3.1. An example of how an asset is located on a map	8
3.2. An example of how a sub-asset within an asset is located	8
3.3. Module selector layout	9
4.1. An example model hierarchy	21
4.2. The example HyperText Markup Language (HTML) from listing 4.8 shown in a browser	25
4.3. Diagram notation	26
5.1. Component structure of EYK	27
5.2. Internal components of the map	28
5.3. The internal design of the OAT framework	29
5.4. Example rendering of the module selector of oat.html	31
5.5. Architecture of the core component of EYK	33
5.6. An example structure of two modules in the module collection	34
5.7. The architecture of the database interfaces	36
6.1. The initial view of the system	37
6.2. The maps layer control, shown with two marker groups	42
6.3. The clustering of markers	43

LIST OF FIGURES

6.4. An example rendering of the property module, shown with data from a borehole	53
6.5. The plot module, implemented for borehole measurements	56
7.1. The Properties module of the air quality sensor	65
7.2. The plot module of the air quality sensor	67
9.1. Planned feature releases of EYK	74
A.1. The complete architecture of EYK	80

List of Tables

2.1. Example location identifiers	5
2.2. Requirements made to EYK	6
3.1. Example types and modules for a customer in the geothermal energy sector	12
6.1. Example series collection	52
6.2. The series collection from table 6.1 converted into a table	52
8.1. Fulfillment of requirements made in section 2.2	69
B.1. Contribution of each developer to specific packages, all tests excluded . . .	81

Abbreviations

AQ	Air Quality
AJAX	Asynchronous, JavaScript and XML
CSS	Cascading Style Sheets
CSV	Comma Separated Values
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
JSON	JavaScript Object Notation
OAT	Organization, Asset Type
PNG	Portable Network Graphics
UML	Unified Modeling Language
URL	Unified Resource Locator
TSV	Tab Separated Values

Acknowledgments

I wish to thank Helgi Gylfason, my friend, colleague and co-author of EYK. This would have been a tough, long and lonely journey without him.

I also wish to thank my advisors, Steinn Guðmundsson and Helmut Neukirchen, for their guidance during this project.

I wish to thank my mother, which proofread this thesis in record time and aided me in eliminating grammatic errors.

Last but not least I wish to thank the guys at Reykjavík Energy's Research and Development department for giving me the opportunity to experience what it is to develop and maintain one's own software.

1. Introduction

This thesis describes the design and development of EYK, a data management solution designed for geographically distributed data sources. EYK is developed in collaboration with Helgi Örn Gylfason who will cover the subject of testing EYK in his thesis. The contributions of each developer is discussed in appendix, chapter B. Parallel to the development, a business plan was made concerning the founding of Hugfimi ehf. and marketing and financing of EYK. While making this business plan, the concept of EYK was presented to three of the leading geothermal energy companies in Iceland. These presentations and the following discussions lead to improvements in the design and a better sense of the need for data management systems. The result of these presentations, the business plan and a personal experience in servicing a geothermal energy company with programs to manage data is used when designing and developing EYK.

During the development of EYK a simplified version of the Scrum project management methodology, was employed. Traditionally, Scrum has three roles: product owner, development team and scrum master [Schwaber and Sutherland, 2013]. This simplified approach excludes the product owner and scrum master and lays focus on task management. The development period is composed of multiple, one week, sprints. At the end of each sprint a collection of tasks is to be completed. These deadlines help keep the pace of the development up. At the start of each sprint developers list up the tasks at hand and estimate its difficulty. The difficulty of each task is evaluated by assigning it task points the points are then used to regulate the workload of each sprint. Each developer is assumed to be able to work the equivalent of forty task points per week which limits the amount of tasks assigned to each sprint Each sprint is assigned a board at the projects trello workspace (trello.com). Each board consists of three lists, backlog, in progress and done. Initially all tasks are in the backlog but as the sprint starts, developers assign themselves tasks and move them to the “in progress” list. The objective of the sprint is to get all cards to the “done” list.

The data management system currently in use at Reykjavík Energy along with the requirements for EYK are described in chapter 2. Based on these requirements, the key design aspects of EYK are described in chapter 3. The topics of the chapter include: the identifications of geographical assets, how the system and external data is accessed and how customers can customize EYK. This chapter also describes design approaches that were tested during the development process of EYK but eventually discarded. The technical foundations which support the remaining chapters of the thesis are described in chapter 4. Also included in the chapter are discussions on the choice of the various tools and packages used to implement EYK. Chapter 5 outlines the architectural design of EYK and how the design decisions described in chapter 5 were implemented. The implementation of the EYK-framework is described in chapter 6 while the development and

1. Introduction

implementation of example plugins is described in chapter 7. The complete framework, with the plugins described in chapter 7, is available at <http://eyk.hugfimi.is>. Finally in chapters 8 and 9 the design and development of EYK is evaluated. The future of EYK is discussed and the roadmap for future releases revealed.

2. System background and requirements

EYK is developed as a successor to Gagnor, a data management system currently in use at Reykjavík Energy. Gagnor is used to manage data from geographically distributed data sources like boreholes and power plants. The concept of EYK was conceived from working on Gagnor and the shortcomings that Gagnor has. To explain how the concept of EYK was developed, the functionality of Gagnor will be described in the following section. Thereafter a requirement specification for the development of EYK will be made. At last EYK will be defined in detail and its sales model outlined.

2.1. Current system

Gagnor is a collection of independent small programs designed to manage and process data. Each program manages a specific collection of data which is typically either a collection of measurements, time series or samples. The programs are specific to the collection of data they manage, some provide a custom presentation of the data, some perform predefined calculations and others interface with external software to extend the functionality of both parties.

The functionality of Gagnor is best described by its individual programs. Gagnor includes programs to: plot borehole measurements; manage, plot and analyze chemical samples; manage steam purity data; manage and plot time series data from ongoing research projects (e.g. the CarbFix and SulFix projects); manage and plot sample data from a tracer project and manage and plot depth, temperature and water level data from groundwater- and drain boreholes. The first three programs exhibit the core functionality of the collection and are further described in the following paragraphs.

Borehole measurements [measurement series]: This program plots temperature and pressure against depth in boreholes. As an example of custom data processing provided by Gagnor, this program corrects the depth measurements taken to the curve of the respective borehole. This is done by applying a polynomial which simulates the curve of borehole and applying it to each depth value.

2. System background and requirements

Chemical analysis [samples]: This program manages chemical samples. It manages samples of two types, high-temperature and low-temperature samples. These samples can be exported as PDF documents, plotted over time or exported as Tab Separated Values (TSV). This program additionally offers an interface to an external geothermal sample analysis software, Watch [Arnarsson et al., 2010]. The samples are converted to the form needed by the analysis software and analyzed with conditions specified by the user. Multiple samples can be analyzed in a single batch and the results plotted or exported as TSV.

Steam purity [time series]: This program stores and plots the concentration of sodium in selected locations within the power cycle of thermal power plants. Figure 2.1 shows an example of a specific, custom presentation provided in Gagnor. The figure shows sodium concentration in Hellisheiði power plant plotted from 2008 to 2012. The red, dashed, line indicates a reference value which the sodium concentration is not supposed to surpass for longer periods.

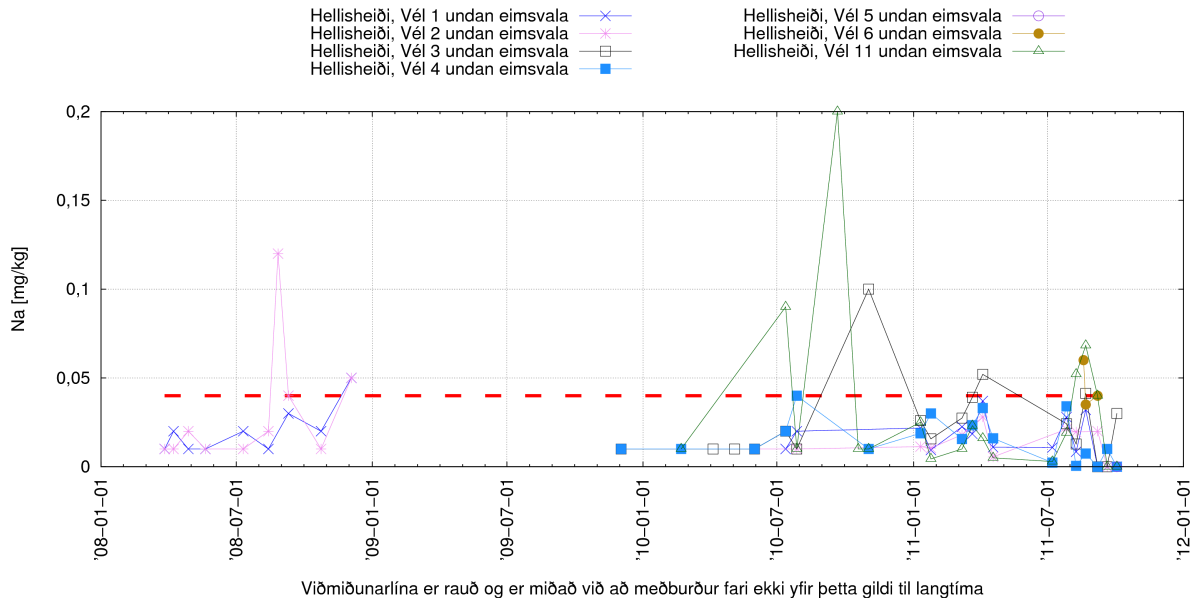


Figure 2.1: Example plot, made by the steam purity program

Each program is custom made for a specific collection of data. This makes the programs highly intuitive as the number of irrelevant features each user is presented with when using the program is minimal. Gagnor has its limitations, both in respect to the end user and the developer. From the developers perspective, the programs have a lot of duplicate code and are hard to maintain while the user is presented with somewhat larger limitations.

2.1.1. Limitations of Gagnor

The limitations of Gagnor include site identification and the accessibility to its programs.

Site identification

The programs of Gagnor use string based asset identifiers to identify the source of data. Each string represents a geographical location, or a location within an asset at a geographical location. Example identifiers are listed in table 2.1. When selecting the desired asset, users are presented with 955 distinct identifiers. A search mechanism is provided to locate the desired identifier from the bunch but even so, this approach is fairly unintuitive.

Table 2.1: Example location identifiers

Identifier	Location
HHEVHHE003HOT	The top of borehole number 3, which is located at Hellisheiði
HHENHHN013ONI	Down the shaft of borehole number 13, which is located at Hellisheiði
HHEVIGS019TBS	Steam separator number 19, located at Hellisheiði power plant.
NEVKHJN006HOT	The top of borehole number 6, which is located at Nesjavellir

Due to the complexity of asset identification, data is often misplaced. This error is hard to correct and is currently corrected by comparing the misplaced data to measurements from similar identifiers.

Program accessibility

The programs were developed on a remote server running Linux. To access the programs a user has to remotely connect to the server machine and run the programs within that remote connection. As the server runs Linux, users of the Windows operating system cannot use their conventional method of connecting to a remote server. To solve this a X window system, Xming, is used [Harrison, 2014]. Connecting to the remote machine is tedious. Users have to know the local host name or IP address of the machine and know which parameters to pass to the Xming session to properly connect. Furthermore users must navigate around the Linux environment to access the programs. For security reasons the remote host is only accessible at Reykjavík Energy’s local area network. As assets which generate the data are outside the reach of this network, their data is collected, stored on an external drive and transported to where connection can be made. At that point data collectors must remember or figure out from which asset each dataset was collected.

2.2. System requirements

Gagnor is used as a guideline for the design of EYK. The requirements made to EYK are listed in table 2.2. EYK aims to provide the same customizability for its users. To achieve this, requirements 1, 2 and 3 in table 2.2 are made. Customers need to have access to the same functionality as in Gagnor when working with data collections. For samples, time series and measurements the system must be able to provide the functionality listed in requirements 4 to 18 in table 2.2. Furthermore, EYK aims to eliminate the accessibility and asset identification limitations present in Gagnor by fulfilling requirements 19, 20 and 21. Additional requirement is made to increase the usability of the system as well as simplifying the setup of the system for new customers

Table 2.2: Requirements made to EYK

ID	Requirement
Customer options	
1	Have their data presented in a custom way
2	Have custom calculations performed
3	Select functionality for each data collection.
Selectable functionality	
4	Store samples
5	Add samples
6	Display samples
7	Interface with the Watch software [Arnarsson et al., 2010]
8	Allow batch analysis of samples through Watch
9	Store time series
10	Import time series
11	Plot time series
12	Export time series as TSV
13	Perform calculations on time series
14	Store measurements
15	Import measurements
16	Plot measurements
17	Export measurements as TSV
18	Perform calculations on measurements
Improvements over Gagnor	
19	Be accessible to users without the use of third party software
20	Be accessible to data collectors on site
21	Identify assets in an intuitive way
Further requirements	
22	Interface to and use data from any accessible data source

3. System outline

EYK aims to provide the same flexibility and functionality as the current system while meeting the requirements stated in section 2.2. The sections of this chapter will describe: How assets will be identified, what options will be available when presenting data, how data from other data sources will be accessed and how the system is accessed by users and data collectors.

3.1. Asset identification

EYK uses a map to simplify the identification of assets. This allows users to zoom and pan to the desired location instead of using tedious and unintuitive strings. To ease transition between systems, the old form of identification will still be available through a search.

Asset is a broad term, a whole power plant is an asset but a single borehole is also an asset. The fundamental difference of the two in regard to data management is the fact that the power plant is not treated as a single data source. It consists of multiple smaller assets which are considered the source of data. If they were all to be displayed at the same location, users would have a hard time selecting the desired one from the map. To solve this problem, the contained assets, or "sub-assets", will be displayed on a second map. This second map will be accessible once the base-asset has been selected. This map will provide a higher resolution of the desired area, or the floor plan of the base asset. To demonstrate this method of identification an example of how a computer is located from within an office is shown. Initially, the user locates the office on the map. He can either pan and zoom, or search for its identification, to locate it. This has been done in figure 3.1.

3. System outline

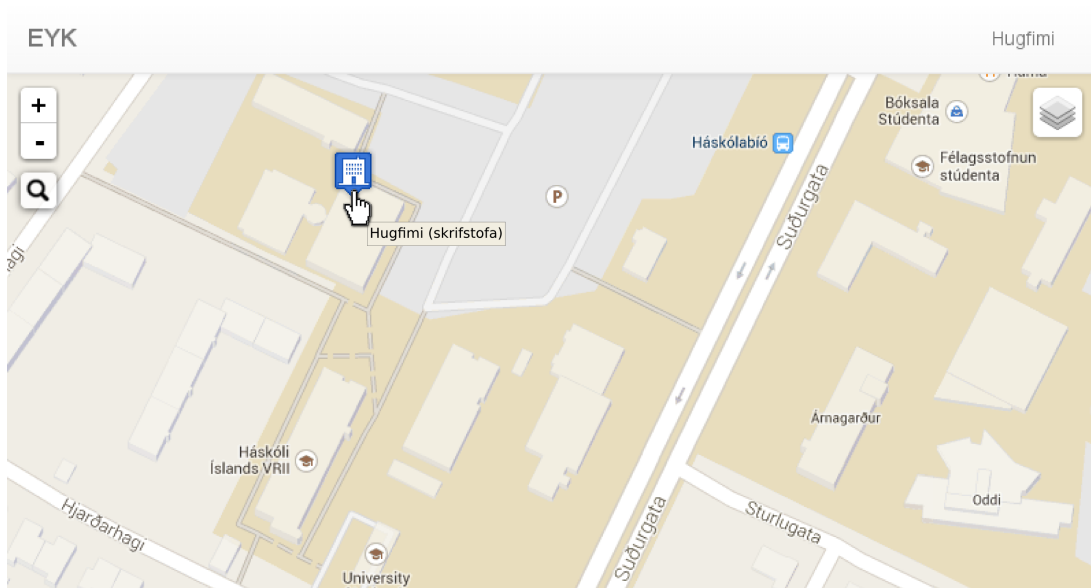


Figure 3.1: An example of how an asset is located on a map

The user continues to select the office and is presented with the floor plan of the office, figure 3.2.

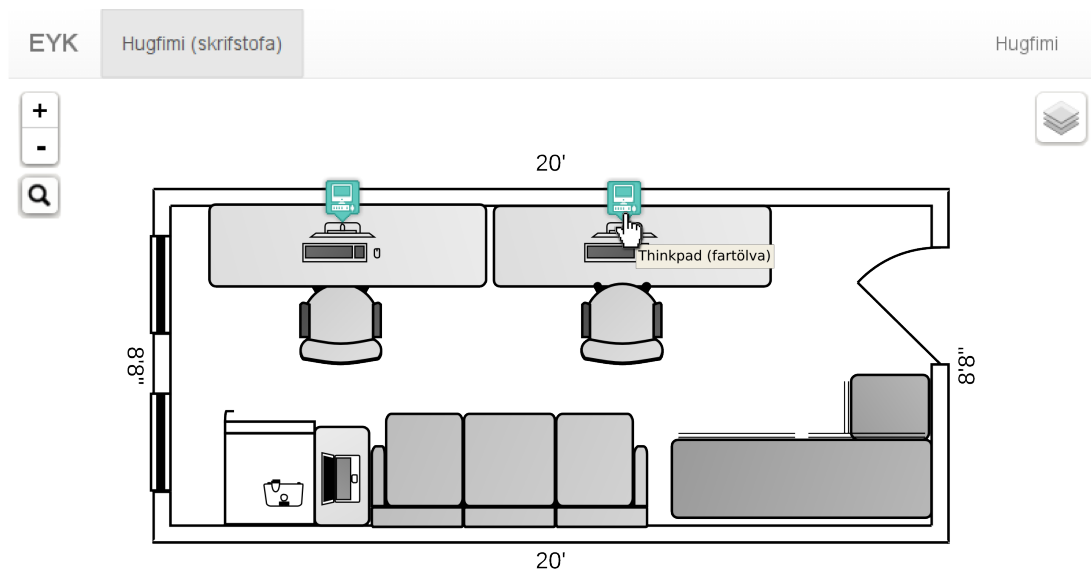


Figure 3.2: An example of how a sub-asset within an asset is located

On the floor plan, all assets contained by the office are displayed at their locations. The user can then finally select the computer and access data related to it. This hierarchy could be repeated multiple times if needed.

3.2. Assigning functionality to assets

Data presentation is one of the main functionalities of the system. Users must be able to view their data in multiple ways and be able to request a custom data presentation. To provide this flexibility the system is designed as a framework. The framework has a single extension point and is configured by two types of plugins.

The EYK framework The framework is extended by adding modules to it. The modules take care of displaying and receiving data. Each module has a defined function, like plotting, showing samples and so forth. The modules will be described further in the next section, section 3.2.1. One type of plugin, database interfaces, is responsible for interfacing to external data sources and storing data. They will be described further in section 3.3. The other plugin type, the Organization, Asset Type (OAT) plugins, define how assets are displayed when they are selected on the map. All assets that share the same type and organization belong to the same OAT plugin. Each OAT plugin contains a definition of modules and database interfaces which are used to show the data belonging to the assets. This way, boreholes can have their modules and power plants a completely different set of modules. Furthermore boreholes from Reykjavík Energy are independent from the boreholes of other companies.

Asset types This structure limits the definition of an asset to a single type. This is inconvenient as some assets have multiple roles. A demonstration of this is a common borehole which is used in a tracer test.

A tracer test is performed by releasing, easily traceable, chemicals into a borehole and waiting for it to show up in nearby boreholes.

The borehole will need to have the modules of the borehole type but also additional modules for the tracer data. This is solved by allowing assets to have subtypes. All assets will have a generic type, which defines the geographical location, icon and the modules that are shared between all assets of that type. Additionally assets can have multiple subtypes where each subtype has its own specification of modules. In the example of the borehole, the generic type is borehole while tracer sample site is its subtype. Upon selecting an asset the user is presented with the combination of modules defined by its generic- and subtypes (figure 3.3). From this selection the user can then choose the module of interest.



Figure 3.3: Module selector layout

3. System outline

3.2.1. Modules

EYK is intended as a successor to Gagnor. This requires EYK to provide *at least* the same functionality as Gagnor. To do so, a collection of modules is provided. The first release of EYK, which this thesis covers, will focus on data visualization and manipulation. It will contain the following modules:

- A **plot module** allows users to visualize their data and comment on it. It also gives them the ability to download raw data.
- A **sample module** manages samples. Through it users can view, add and edit samples. Additionally the ability to lock samples from editing will be provided.
- A **property module** shows asset properties and an image of it, if available. The properties have a key-value format and are uneditable.
- A **component module** allows users to view the components of the selected asset. The function of the module was described in section 3.1 where it was used to locate a computer within an office.
- A **data collection module** allows users to import data through EYK. With EYK being a web based system the users can import data straight from the source. This module thus simplifies the flow of data and in return minimizes the probability of human error.

These modules are designed to fulfill requirements 4-18 in table 2.2.

3.3. Database interfaces

EYK is designed to be able to display data that has already been collected and stored elsewhere. It does not require its users to move their data from the storage media it's currently on to use it in EYK. This means that after installing the system, users can continue to use the old systems they are familiar with *and* access their data in EYK. This flexibility is required to allow customers to transition between systems in their own pace.

Initially, EYK used a direct connection to data sources to make their data available to modules. This approach required the system to download, process and return data to each user, each time he requested data. No data was stored on EYK so if users requested the same data twice, it had to be downloaded and processed twice. This design introduced an unacceptable delay and poor usage of resources and was therefore changed to a different, two piece, design.

The new design introduced the database interfaces, a intermediate storage. The database interfaces use scrapers download data from their data sources, process it and store in EYK's database. Each scraper looks for new data with a defined frequency and imports

it. This frequency is defined by customers when they specify their interfaces and gives them the ability to determine the worst case delay between data availability in data source and EYK. This redesign resulted in much quicker data responses and a more stable network traffic. This design, additionally, allows the implementation of data validators in the database interface plugins. Data validators could for example notify users when a parameter exceeds some value or drops below another. The addition of data validators is not discussed further in the thesis but is merely a possibility provided by a successful redesign.

3.4. Accessibility

To make the system accessible to most users it will be web based. A web based system is accessible to a wide variety of devices, including mobile phones, tablets and more. By being web based data collectors can access the system while on-site and import data directly. This simplifies the flow of data and minimized the risk of human error.

Web browsers are strictly considered third party software. Designing EYK as a web application thus violates requirement 19 in 2.2. However, browsers are installed by default on most operating systems and most users are very familiar with them. On these grounds the violation of requirement 19 will be overlooked in the case of the browser.

3.5. Sales model

The sales model of EYK has three units, base system, module and maintenance plan. A new customer purchases the base system, which has a fixed price. Upon buying the base system customers also define their assets types. When defining an asset type customers need to specify two things, the modules the asset type will have and the data each of them is to display. Customers then pay for each implementation of a module they use. This means that a plot module for a borehole asset type and a plot module for a power plant asset type are two implementations of the plot module. The module price is fixed and kept low to encourage customers to add modules.

When customers have defined their system they agree to a maintenance plan which entitles them business hour support for their system. The price of the maintenance plan is dependent on the number of individual modules in their system The levels are 5:

- 1-10 modules
- 11-25 modules
- 26-50 modules
- 51-100+ modules

3. System outline

The focus of this sales model is to keep initial expenses for the system in the low end and the subscription in the high end.

Example: The geothermal research department of Reykjavík Energy wants to use EYK. They buy the base system and define their asset types. They need a total of five generic types, five subtypes and 25 modules, see table 3.1.

Table 3.1: Example types and modules for a customer in the geothermal energy sector

Generic Type	Sub-types	Modules
Borehole	-	Property
-	-	Report
-	Hot borehole	Plot
-	-	Data collection
-	-	Sample
-	Cold borehole	Data collection
-	-	Sample
-	Tracer sample borehole	Plot
-	-	Data collection
-	-	Sample
-	CarbFix borehole	Plot
-	-	Data collection
-	SulFix borehole	Plot
-	-	Data collection
Power plant	-	Property
-	-	Components
Steam separator	-	Property
-	-	Plot
-	-	Data collection
Machinery unit	-	Property
-	-	Plot
-	-	Data collection
Steam condenser	-	Property
-	-	Plot
-	-	Data collection

In table 3.1 the boreholes have 5 subtypes and each subtype has its own definition of modules. The power plants have no subtypes and two modules, a property module and a component module. The component module is needed to access the assets contained by the power plant. These assets are also of a generic type. They are: Steam separator, machinery unit and steam condenser. These components are listed because the geothermal research department is responsible for analysis on sodium concentration in these parts of the geothermal power plant. Others are left out as they do not need to be accessed by the departments employees.

This system includes the following sales units: a base system, 25 modules and a maintenance plan for 10-25 modules.

4. Technology used

This chapter will cover the basics of web applications, the software chosen to power EYK and how the software is used in the development of EYK. Additionally the diagram notation used in the thesis is clarified.

4.1. Web applications

EYK will be developed as a web application. A web application consists of a backend and frontend. Backend code is executed on a web server which hosts the application and frontend code is executed in the users browser.

The backend software manages data and generates HTML web pages for the frontend to display. When users access a Unified Resource Locator (URL), the backend takes care of interpreting that URL and returning the HTML page corresponding to it. This happens on a server where the web application is stored.

The server has its own URL which is often the root URL of the application. The development server used in this project, for example, has the URL `http://eyk.hugfimi.is/`. When this URL is entered in a browser the server sends back the HTML page corresponding to the URL. This page is then rendered in the users browser for the user to view.

If the URL is changed by adding some string behind the slash, another HTML page is returned. If the added string is connected to a HTML page that page is returned, it's not connected to any page, a 404 ("page not found") error is returned and shown to the user.

The HTML pages often contain links to URLs which direct the user user to another HTML page when selected. This way users only have to know the "root URL" of the web application they are using and move from that page to others by a click of a mouse.

4.1.1. Static media

As well as linking to other HTML pages, HTML pages often link to static media. This static media can be an image, video, JavaScript files or Cascading Style Sheets (CSS) files. The media is then, along with the markup of the HTML used to render the final look of the web-page. These static media files have their own URLs so they can be properly

4. Technology used

delivered to the browser. These files are also stored on a server, often separate from the one that delivers the HTML pages.

4.1.2. Cascading style sheets

CSS files contain the definition of how HTML elements are displayed. CSS styles can control color, size, order and transparency of elements. Styles can be applied to a single element by referencing its id applied to a group of elements by referencing their class or type. CSS styles can exclude elements from being rendered, effectively hiding them.

4.1.2.1. Bootstrap

There exist several frontend frameworks that aid developers in giving their pages added functionality and a uniform style. This project uses one of these frameworks, called Bootstrap. Bootstrap offers multiple components, responsive design and customizability [Otto and Thornton, 2013a]. The main benefit of using Bootstrap is its responsive design.

Responsive grid system Bootstrap provides its users a grid system which consists of twelve columns for each row. Each element placed in the grid system can then be assigned a number of columns. The specification of column numbers is dependent on the size of the screen currently viewing the page. For example, an element can be configured to cover 3 columns on a laptop (medium screen), 6 columns on a tablet (small screen) and the whole 12 columns on a smart phone (extra small screen) [Otto and Thornton, 2013b].

This is done by assigning classes to elements. The element described in the example would be configured like shown in listing 4.1.

```
1 <div class="row">
  <button type="button" onClick="alert('I was clicked')"
3      class="col-md-3 col-sm-6 col-xs-12">
    Click me
5  </button>
  <!-- additional components
7      taking up the remaining 9 columns
      on medium and large sized screens
9  -->
</div>
```

Listing 4.1: Assigning columns to a button depending on screen size

The outermost division element shown in listing 4.1 denotes a new row. This row is expected to contain elements which are assigned a sum of 12 columns. The button is then configured to span 3 columns (25% of screen width) on medium to large sized screens, 6 columns for small screens (50%) and 12 columns on extra small screens (100%). The

remaining elements inside the division will either fill up the space left along side the button or be moved below it in case there aren't enough columns available.

By using this grid system, users can always see the whole width of the page, independent of screen size.

4.1.3. JavaScript

To provide interactive web-pages, **js!** (**js!**) is used. It can be used to perform calculations, update a part of the web page or perform animations.

JavaScript introduces some problems programmers are not faced with in conventional programming. One which was encountered in this project is the fact that code can be overwritten. If a HTML page uses two JavaScript files that both define a function named **myFunc** the first definition will be overwritten [Mozilla Developer Network, 2014]. All code in the first JavaScript file will thus be executing the **myFunc** defined in the second JavaScript file. The same applies to variables.

To avoid this, JavaScript code is given a namespace by inserting it into a object. By doing so the **myFunc** functions would be separated by defining them in separate namespaces. This method is shown in listing 4.2.

```

> var namespace1 = {};
2 > var namespace2 = {};

4 > namespace1.myFunc = function(){
    console.log('I am defined first');
6 };

8 > namespace2.myFunc = function(){
    console.log('I am defined second');
10 };

12 > namespace1.myFunc()
    "I am defined first"
14 > namespace2.myFunc()
    "I am defined second"

```

Listing 4.2: Namespacing in JavaScript

JavaScript classes

JavaScript classes are implemented as functions whose functions and variables are all bound to the **this** object. These classes are then evaluated using the **object.call(context, arg1, arg2, ...)** method provided by JavaScript. This method executes the object with

4. Technology used

the given arguments and binds the results to the context, [Mozilla Developer Network, 2008]. This method is shown in listing 4.3.

```
1 > var MyClass = function(constructorArgument){
    this.instanceVariable = constructorArgument;
3   this.otherVariable = 'I am a classy class!';
    this.description = function(){
5       console.log(this.instanceVariable +
        ' because ' + this.otherVariable);
7   };
};
9
> var oneInstance = {};
11 > var otherInstance = {};

13 > MyClass.call(oneInstance, 'I love my Class');
> MyClass.call(otherInstance, 'I do not like my Class');
15
> oneInstance.description();
17 "I love my Class because I am a classy class!"

19 > otherInstance.description();
    "I do not like my Class because I am a classy class!"
```

Listing 4.3: The class instantiation method used in this project

4.1.4. Frontend libraries

There exists a selection of libraries, commercial and open source, that aid in development of web based applications. These libraries speed up the development process and increase the quality of web based applications. A few of these are used in this project and will be described in the following subsections.

4.1.4.1. Leaflet

Leaflet is an open source JavaScript library for interactive maps [Agafonkin, 2013]. Leaflet has a large collection of open source extensions which help increase the functionality of mapping applications. When deciding which interactive map library to use, OpenLayers and Google Maps API were also considered. These mapping libraries have similar features and could all be used in EYK. In the end, Leaflet was chosen because of its clean looks, clean API and extendability.

Leaflet doesn't come with maps servers built-in to the API but upholds various web mapping standards. This means that customers of EYK can use their own maps. This is preferable as most energy companies possess high resolution imagery of their operation

site. The default maps EYK will use however come from Google. They are added to Leaflet by using the `Google.js` plugin provided by Peter Shramov [Shramov, 2014].

4.1.4.2. Highcharts

To visualize data, Highcharts was chosen [Highsoft, 2013]. This library is not open source but free for non-commercial software under the Creative-Commons licence¹. When EYK becomes commercial a licence must be purchased at the price of \$390. The benefit of Highcharts over an open source library like d3² is how easy it is to create interactive visualization of data. For later versions of EYK some module might require the use of an extensive visualization library like d3.

4.2. Web application framework

After a short research, developers decided the viable web application frameworks for EYK were Django and node.js. The main difference between the two is that Django is written in Python while node.js is written in JavaScript.

Apart from being written in Python, the developers choice of a programming language, Django offers a geographical extension, GeoDjango [Django Software Foundation, 2013g]. GeoDjango enables Django to manage spatial data in an intuitive way. This is valuable considering EYK will use maps and geographical locations to identify assets.

4.2.1. Django

A Django web application is composed of multiple independent Django “apps”. It is usually structured as a collection of apps along with a `manage.py` module, a `settings.py` module and a `urls.py` module.

```

project name.....project root folder
├── app 1
├── app 2
├── project name
│   ├── settings.py.....project configuration
│   ├── urls.py.....URL routing
└── manage.py.....project management module

```

¹<http://creativecommons.org/licenses/by-nc/3.0/>

²<http://d3js.org>

4. Technology used

4.2.1.1. The settings module

The `settings.py` module is generated when a new Django project is started and can be used to customize the project. This module contains, among other things, the database settings, timezone settings, which Django apps are used in the project and more [Django Software Foundation, 2013e].

4.2.1.2. The urls module

The `urls.py` module routes the incoming URLs to Django apps. When a HTML page is requested by accessing a URL a Django HyperText Transfer Protocol (HTTP) request object gets created. A request object contains various information about the HTTP request including any GET or POST data, cookies, sessions and more [Django Software Foundation, 2013c].

To route a URL the Django project tries to match it to patterns defined in the `urls.py` module. If a URL matches a defined pattern its HTTP request gets forwarded to the app specified. If it doesn't match any defined pattern the request a page not found (404) error is returned.

The patterns defined in the `urls.py` module can include variables. Take for example the URL `http://eyk.hugfimi.is/asset/12`, which is used in EYK. The last part of the URL (12) is a variable but all URLs that start with `http://eyk.hugfimi.is/asset/` and end with a non-negative number should all be routed to the same app. To match a URL like this the patterns are defined with regular expressions [Django Software Foundation, 2013f]. The pattern matching the example URL is shown in listing 4.4

```
patterns('app.views',
    url('^asset/(?P<asset_id>\d+)$', 'view_asset')
)$
```

Listing 4.4: The URL pattern matching the example URL

Line one in listing 4.4 implies that all URL patterns defined within the parenthesis will be routed to the `views.py` module in the Django app named `app`. Line two defines the pattern itself.

The `^asset/` part matches all URLs that start with `asset/` immediately after the root URL (`http://eyk.hugfimi.is/`). A named regular expression is dubbed `asset_id` and defined as one or more positive digits by adding `(?P<asset_id>)\d+` and finally a `$` is added to inform that nothing should trail the digits.

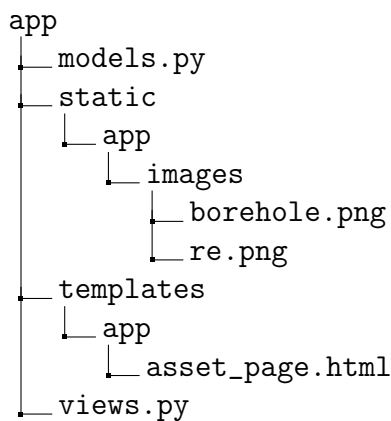
A URL matching this pattern will cause a call to the `view_asset` function in the `views.py` module in `app`. Two parameters are provided to the function call. The HTTP request object and the named regular expression. The signature of the `app.views.view_asset` function is thus: `view_asset(request, asset_id)`, where `asset_id` is in this case equal to 12

4.2.1.3. The manage module

The `manage.py` module contains a collection of functions which are used to manage the project. Among these functions are functions to: run a development server, create a new Django app, synchronize with its database and more.

4.2.1.4. Django app

Django apps can contain their own models, templates and views. The views and templates are optional but the models must exist in an app for it to be considered a Django app. A typical structure of a Django app is as follows



Models The models store data. Each model defines its fields which can be assigned data and saved in a database. The fields can be of various types e.g. character, floating point and foreign key.

To demonstrate how these fields work an example model hierarchy for Django app is shown in figure 4.1. Figure 4.1 uses the Unified Modeling Language (UML) class diagram notation The Object Management Group [2010].

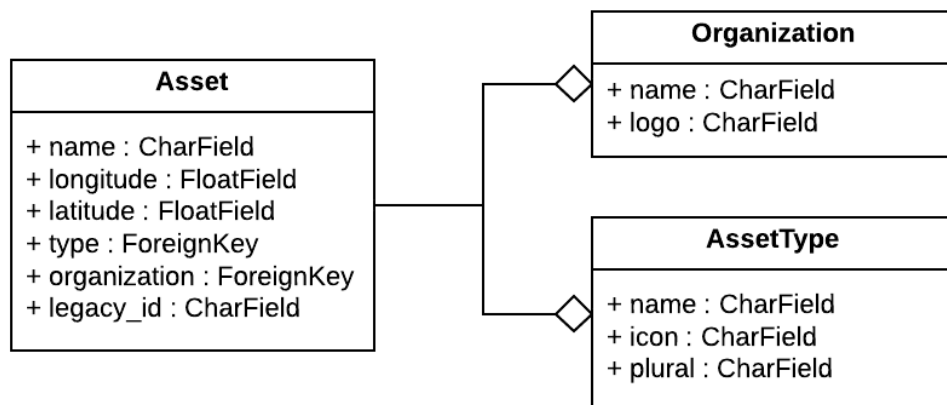


Figure 4.1: An example model hierarchy

4. Technology used

The `Asset` in figure 4.1 has six fields: a `name`, a `longitude`, a `latitude`, a `type`, an `organization` and a `legacy_id`. The `name` and `legacy_id` are character fields, which means they are composed of multiple characters. The `longitude` and `latitude` are floating point fields, which means they are floating point numbers.

The `type` and `organizations` however are foreign key fields. Foreign key fields contain a reference to other model. In this case they contain a reference to the `Organization` and `AssetType` models. Each `Asset` can have one, and only one reference to a `Type` and `Organization` while the `Type` and `Organization` can be referenced by many `Assets`.

To store an instance of `Asset` in the database it is created and saved like shown in listing 4.5

```
1 borehole = AssetType(  
    name = 'Borehole'  
3     icon = 'app/images/borehole.png'  
    plural = 'Boreholes'  
5 )  
borehole.save()  
7  
reykjavik_energy = Organization(  
9     name = 'Reykjavik Energy'  
    logo = 'app/images/re.png'  
11 )  
reykjavik_energy.save()  
13  
my_borehole = Asset(  
15     name = 'HE-03'  
    longitude = -21.334  
17     latitude = 64.040  
    type = borehole  
19     organization = reykjavik_energy  
    legacy_id = 'HHEVHHE003HOT'  
21 )  
my_borehole.save()
```

Listing 4.5: Creating and saving models

When these models are created, they get assigned a `private_key` from a sequence. For example, if `my_borehole` is the twelfth borehole stored in the database it will get assigned the `private_key` with the value of 12. This number is a definitive identifier for this borehole and is accessible by the name `pk`,

```
> my_borehole.pk  
12
```

The borehole can then be retrieved by referring to this number.

```
> my_borehole = Asset.objects.get(pk=12)
> my_borehole.name
'HE-03'
```

Views The `views.py` module is where HTML pages are generated. HTML pages are generated from templates, like the one located at `app/templates/app/asset_page.html`. To generate a HTML page from a template, a view function gathers the data the template needs and generates it with the data as context. To provide a better idea of this process, the example from the `urls.py` discussion, section 4.2.1.2 is continued.

The `urls.py` module has matched the URL `http://eyk.hugfimi.is/asset/12` to the `app.views.view_asset` view function. The `view_asset` function accepts two parameters, the HTTP request and the `asset_id` (12 in this case).

The `asset_id` is used to access a `Asset` model, like the one described in the models section above, along with its related models, see listing 4.6. The three models are added to a context dictionary, which is passed to the rendering of the `asset_page.html` template, see listing 4.6.

```
def view_asset(request, asset_id):
2     asset = Asset.objects.get(pk=asset_id)
    asset_type = AssetType.objects.get(pk=asset.type)
4     organization = Organization.objects.get(pk=asset.organization)

6     context = {
        'asset': asset,
8         'asset_type': asset_type,
        'organization': organization
10    }
    return render(request, 'app/asset_page.html', context)
```

Listing 4.6: An example implementation of a view function

The parameters defined in the `request` object can be used to influence how the HTML page is generated before its passed to the `render` function, but for simplicity it is just passed to the `render` function.

Listing 4.6 also shows how models can be retrieved. The models stored in listing 4.5 are retrieved by using the private key of the asset. Once the models have been loaded from database by issuing `objects.get`, all their fields become accessible. This is shown in listing 4.6 in lines 3 and 4 where the assets type and organization are accessed.

Templates Each HTTP request is answered with a HTTP response. As stated in the Django documents [Django Software Foundation, 2013a]:

In contrast to `HttpRequest` objects, which are created automatically by Django,

4. Technology used

HttpResponse objects are your responsibility. Each view you write is responsible for instantiating, populating and returning an HttpResponse.

HTTP responses can be created and written within the view function. This method can be tedious when writing long HTML pages so Django provides a complete template language, which is able to generate HTTP responses from HTML templates.

The `render` function shown in listing 4.6 takes a path to a template as an argument along with a context dictionary. All the keys of the context dictionary are available as variables in the template file. The values of the variables match the values of the keys in the dictionary.

The variables can be accessed from within the template file by encasing them in double curly-braces like shown in listing 4.7.

```
1 <h1> {{ asset.name }} </h1>
  <h2> {{ asset_type.name }} </h2>
3 <img src='/static/ {{ organization.logo }}' />
  <h2> {{ organization.name }} </h2>
5 <h3>
    Located at longitude: {{ asset.longitude }}, latitude: {{ asset.latitude }}
7 </h3>
```

Listing 4.7: An example implementation of the `app/asset_page.html` template

With the context provided in listing 4.6 the template in listing 4.7 will result in the HTML page shown in listing 4.8.

```
1 <h1> HE—03 </h1>
  <h2> Borehole </h2>
3 <img src='/static/app/images/re.png' />
  <h2> Reykjavik Energy </h2>
5 <h3> Located at longitude: —21.334, latitude: 64.040 </h3>
```

Listing 4.8: The example template from listing 4.7 rendered with the context provided in listing 4.6

The HTML page shown in listing 4.8 will then be returned to the user's browser, resulting in the web page shown in figure 4.2.

4.2.2. Database

There are only a handful of databases which work with GeoDjango. The GeoDjango documentation describes how to use MySQL, Oracle and PostgreSQL [Django Software Foundation, 2013g]. PostgreSQL was chosen because developers had positive prior experience using it and it is open source [The PostgreSQL Global Development Group, 2014].



Figure 4.2: The example HTML from listing 4.8 shown in a browser

4.2.3. Task queue

Task queues are useful when executing resource heavy tasks where the output is not used immediately. They enable routines to add their task to a queue and continue their own execution. The task will then be executed once it has reached the front of the queue. For EYK, the fact that tasks can be scheduled for execution periodically is especially valuable. Defining reoccurring tasks allows EYK to scrape new data from data sources with a defined frequency and keeping its data up to date. The choice of a task queue was relatively easy. Celery, a task queue for Python, interfaces directly with Django [Solem, 2013]. Celery auto detects tasks in Django applications which are stored in a `tasks.py` module. The tasks can be executed once or repeatedly with a defined delay. To add a reoccurring task to an app like the one described in section 4.2.1.4, a tasks module is added to it.

```

app
├── models.py
├── static
├── templates
├── tasks.py ..... The added module
└── views.py

```

Celery then looks for functions defined as tasks from within the module. An example reoccurring task is shown in listing 4.9.

```

1 @periodic_task(run_every=(crontab(day_of_week='*', hour='*', minute='*', second='00'))
  def update_clock()
3     clock.update

```

Listing 4.9: An example of a reoccurring task which reoccurs every minute

4. Technology used

The task defined in listing 4.9 is added to a queue every day of the week, every hour at the beginning of each minute.

4.3. Diagram notation

The diagrams in this thesis largely follow the UML 2.0 standard for class diagrams [The Object Management Group, 2010, p. 107]. The used relationships and notations are shown in figure 4.3. The addition to the common standard is the link notation shown at the bottom of figure 4.3. It is used to denote a connection from a HTML page to view. Additionally the URL of the link is shown on the dotted line of the connection.

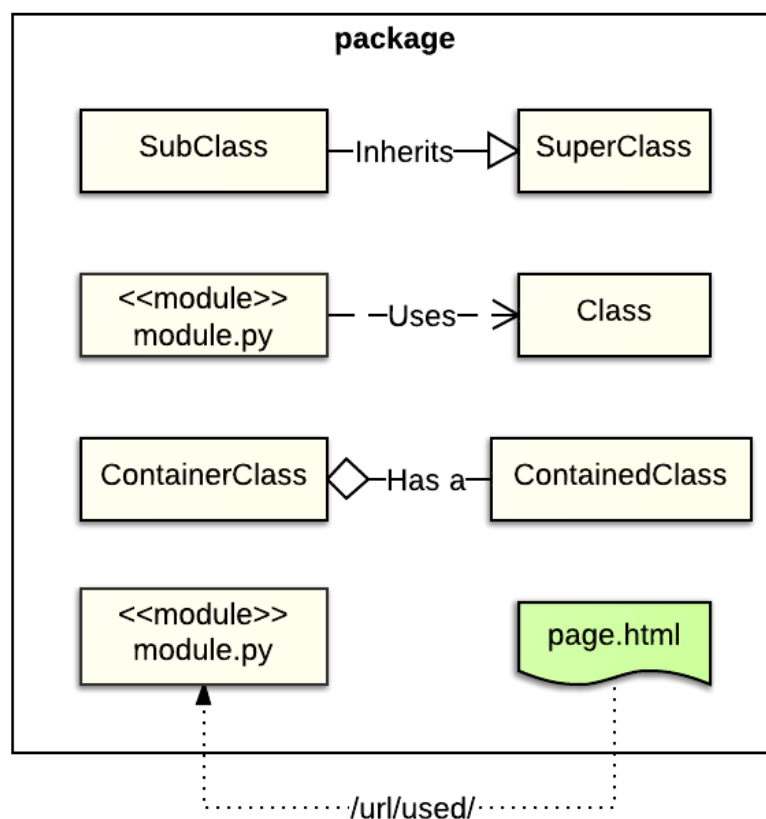


Figure 4.3: A legend describing the notation used in diagrams in this thesis

All relationships in the figure move from left to right, which means:

- The SubClass inherits the SuperClass
- The python module, module.py, uses Class
- The ContainerClass has a ContainedClass
- The HTML page page.html as a link to a URL which is handled by module.py

All the items in figure 4.3 are a part of a package name **package**.

5. Architectural design

Each customer of EYK will have its own custom version of EYK. These versions will contain the assets each and every customer wants, the assets will use the chosen modules to display data from the database interfaces required. To be able to set up a system specific to a customer needs, the system architecture must be flexible. Additionally the work required when specifying assets, their modules and the database interfaces used by these modules, must be minimized without compromising flexibility.

To achieve this, EYK is designed as a framework. As stated in section 3.2, the framework will have two types of plugins, database interfaces and OAT plugins. Furthermore to provide variety in data management, a collection of modules will be provided. These components and their interaction is shown in figure 5.1.

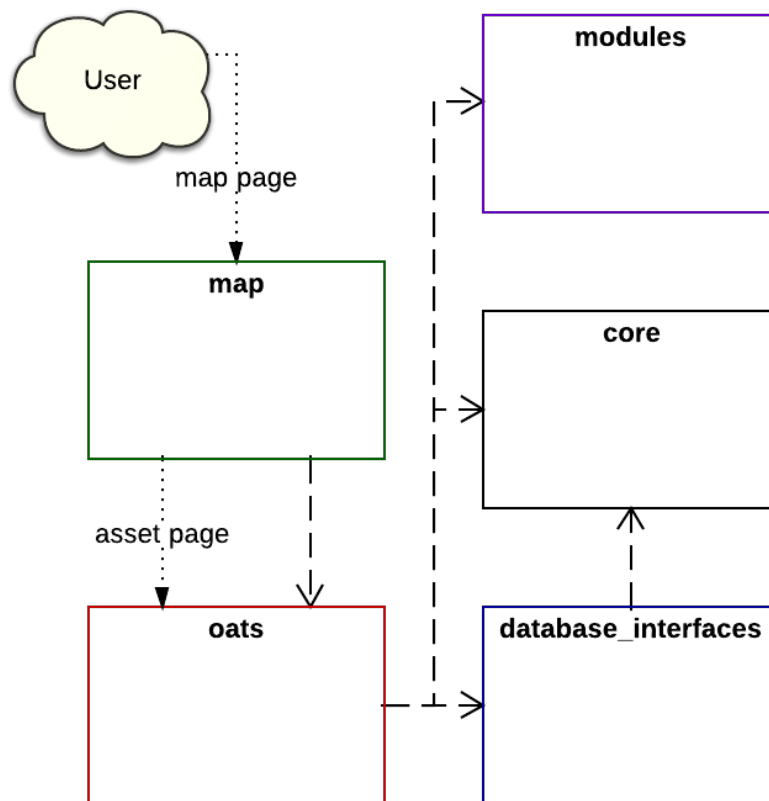


Figure 5.1: A component diagram describing the basic component interaction

The architecture consists of three Django apps and two app collections. The map app and oats app provide the main functionality of the framework along with its two pages, the map page and the asset page. Additionally the system contains a core app, a module collection and a database interface collection.

5. Architectural design

The map app displays a map with all the assets stored in the oats app. When an asset is selected the user is presented with the “asset page” of that asset. The asset page is shown by the oats app with the help of the remaining three components, the modules, the core and the database interfaces. Customers can customize their own EYK system by designing their own OAT and database interface plugins and even have custom modules made. The architecture of apps, plugins and modules will be described in detail in the following sections.

5.1. Map

The map is the landing page of the system. From there the user is presented with a collection of markers, placed on a map. Each marker represents a specific asset and is located at the geographical location of that asset. The internal structure of the map component is relatively simple. It consists of a views module and a single template as shown in figure 5.2.

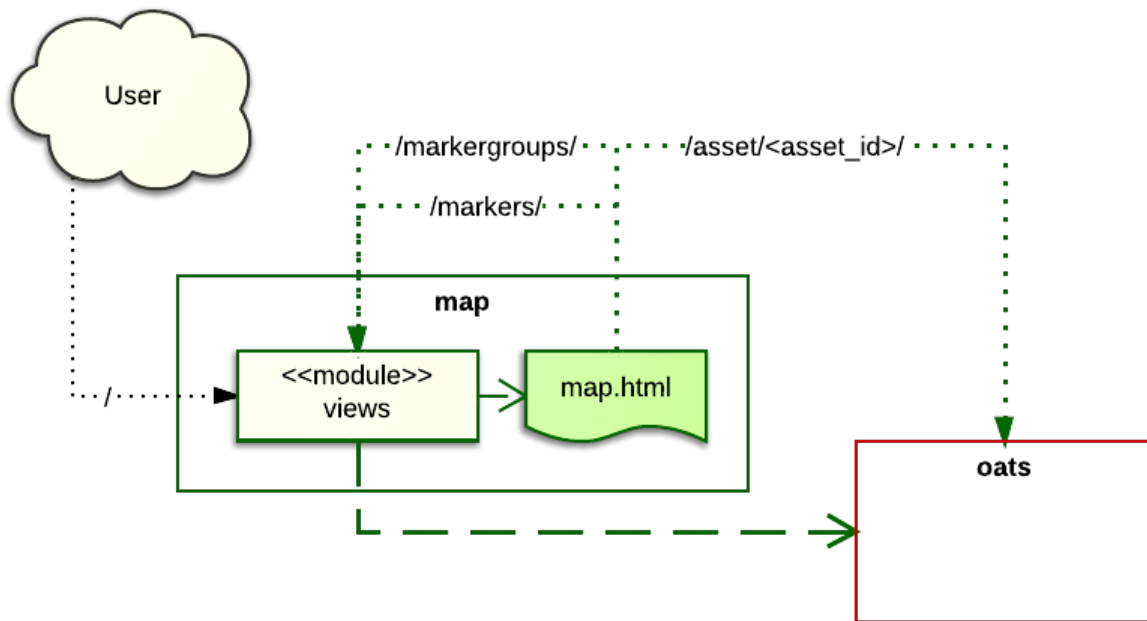


Figure 5.2: Internal components of the map

The map page (`map.html`) makes two callbacks for data. One is a request for the markers and the other is for the marker groups. This is done to minimize the loading time of the map page. When the map page is requested, the user is promptly presented with an empty map and moments later the markers are loaded into the map after the requests have been made.

When the markers are requested, the `views` module collects all available assets, which are stored in the oats component, and returns them to the map page. Each asset has a defined longitude and latitude which the map page uses to position its marker. The assets are constructed much like the `Asset` model in foundations, section 4.2.1.4.

The marker groups are used to layer the markers. Each marker group represents a group of markers whose assets share organization and type. Users can then choose to hide a specific group of marker from the map.

Each marker contains a link to the asset page for the asset it represents. To identify the asset page of each asset the `asset_id` is provided in the URL to the page. Each asset is given its own unique `asset_id` when it is stored in the oats component, this process is introduced in foundations, section 4.2.1.4. When the oats component gets a request for an asset page it can use that id to get a reference to the asset selected.

5.2. Oats

The oats app contains one of the plugins of the framework, the Organization, Asset Type (OAT) plugin. Each OAT plugin contains the implementation detail for a asset page. Its `views.py` module receives the request for the asset page and loads the `Asset` model referenced by the request. Each `Asset` has its own OAT plugin. This OAT plugin uses the Django apps from the modules and database interfaces to render `oat.html`, the asset page, see figure 5.3. The figure displays a single oat plugin (OATClass) for reference

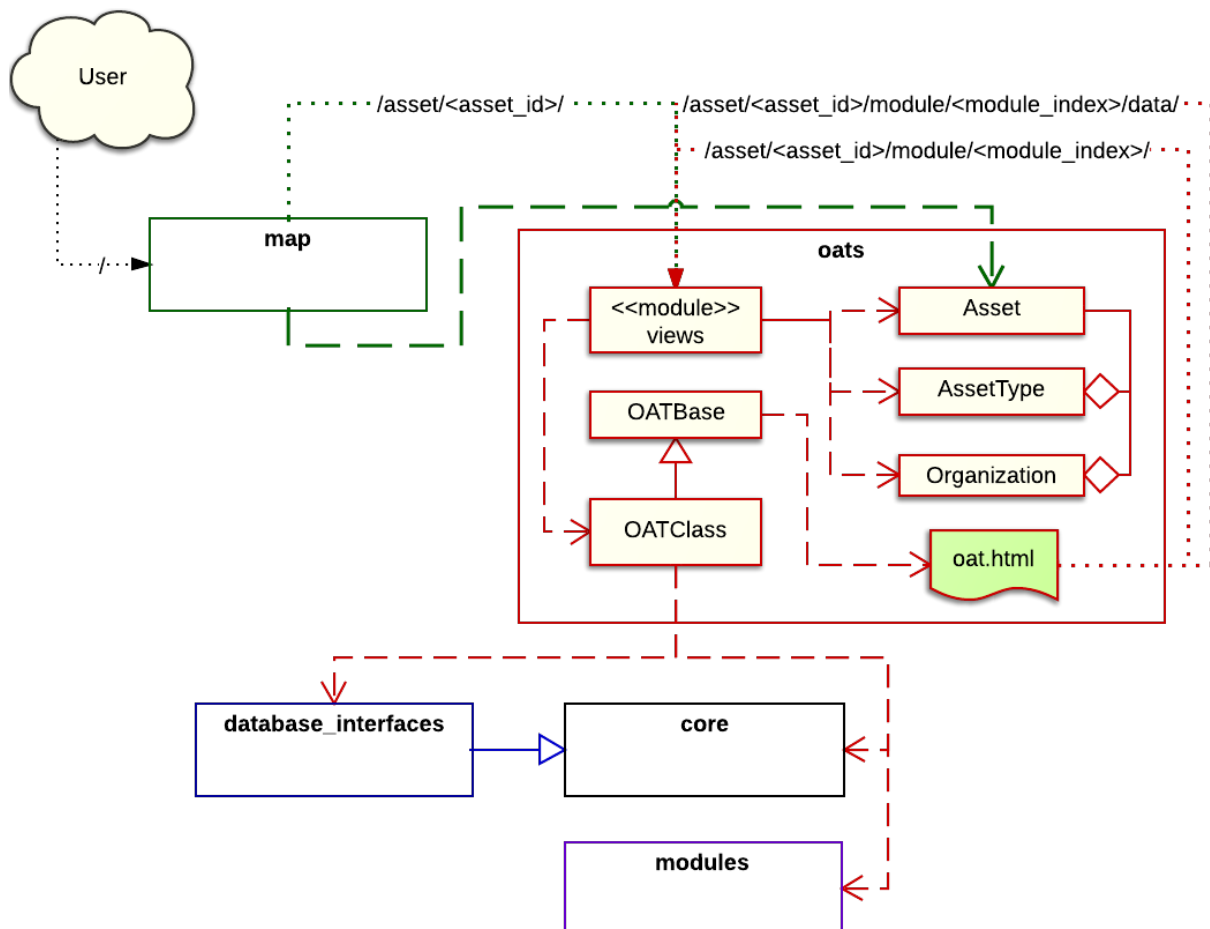


Figure 5.3: The internal design of the OAT framework

5. Architectural design

while each system will contain one OAT plugin for each combination of organization and asset type it has.

To be able to reference OAT plugins by the organizations and asset types they represent, they are stored on a specific path defined by the name of its organization and asset type. The class is named `AssetTypeName`, which is stored in a file (python module) named `asset_type_name` within a folder (python package) named `oats.organization_name`. The full path of the OAT plugin is thus `oats.organization_name.asset_type_name.AssetTypeName`. Each `Asset` can compose this path from its type and organization to access its OAT plugin.

The remainder of this section will describe the OAT plugins and how the components of the oats app enable them to be plugged-in to the EYK framework and define the asset page.

5.2.1. OAT plugins

Each OAT plugin inherits from an abstract base class, `OATBase` (fig 5.3). This base class uses three variables its subclasses must define, `BASE_MODULES`, `MODULES` and `MODULE_LABELS`

The `BASE_MODULES` variable is a list that contains the names of the module apps used by the OAT plugin as strings.

The `MODULES` variable is a list that contains a string id for each instance of a module app used. These ids are a reference to the methods used to access the modules.

The `MODULE_LABELS` variable is a dictionary which contains the labels for the modules in the `MODULES` list. These labels are user visible and can contain special characters.

The structure of an example OAT plugin is provided in listing 5.1

```
1 class OATClass(OATBase):
    BASE_MODULES = ['module0', 'module1']
3    MODULES = ['module0', 'module1_d1', 'module1_d2']
    MODULE_LABELS = {
5        'module0': "I'm module number 0",
        'module1_d1': "I'm module number 1, and I display data1",
7        'module1_d2': "I'm module number 1, and I display data2"
    }
9
11    def _get_module0_html(...
13
15    def _get_module1_d1_html(...
    def _get_module1_d2_html(...
```

```

17  def _get_module0_data(...)
19
    def _get_module1_d1_data(...)
19
    def _get_module1_d2_data(...)

```

Listing 5.1: Example OAT plugin

Listing 5.1 shows an example of where a single module app, `module1`, is used twice in one OAT plugin. To solve this conflict, the module ids are made distinct by appending the module app name with a reference to the data each instance displays.

As shown in listing 5.1, each module id references either one or two methods in the OAT plugin. The methods whose names end with `_html` are called the “module callback methods” and provide the HTML markup for the respective module. The methods whose names end with `_data` are called the “module data callback methods” and provide the modules with data. Each of these methods is expected to return a HTTP response, which is typically a rendered HTML template as described in foundations, section 4.2.1.4

5.2.2. The asset page, `oat.html`

The `oat.html` page consists of a module selector and a module body. The module selector corresponding to the example in the previous section would look like the banner in figure 5.4.

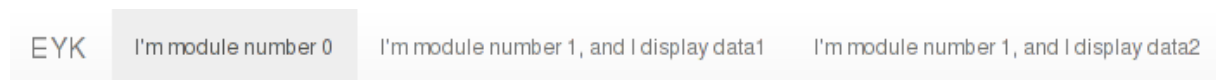


Figure 5.4: Example rendering of the module selector of `oat.html`

Below the module selector the selected module is shown. When a module is first selected, it is loaded into the body by making a “module callback” then each time the user requests some data in the module, the module can make a “module data callback”.

To avoid reloading modules as the user switches between them, each module is placed in its own container. When a module is selected its container is made visible and the previously active module container is hidden.

5.2.3. Module callbacks

To display the modules in the asset page two callbacks are provided. The module callback, which is provided through the `/asset/<asset_id>/module/<module_index>/` URL and the module data callback which is provided through the `/asset/<asset_id>/module/<module_index>/data?` URL. The callbacks originate in the `oat.html` and are sent to the `oats.views` module as shown in figure 5.3.

Module HTML callback

Upon receiving a module callback, the `oats.views` module retrieves the OAT plugin corresponding to the `<asset_id>` in the callback. The views module then executes the “module callback method” of the module referenced by the `<module_id>`. The rendered template returned by this method is then returned to the asset page which displays it in the module container.

Example: “Module number 0” in figure 5.4 is requested. The module can be requested by the module callback URL. For this example the URL `eyk.hugfimi.is/asset/12/module/0` is used. From this URL the framework reads:

- An asset with `asset_id=12`.
 - This asset has an OAT plugin structured like the one shown in in listing 5.1.
- A module name at index 0 in the `MODULES` list in the OAT plugin referenced by the `asset_id`.
 - `'module0'`

Once the module name has been deduced from the URL the framework uses that name to determine the signature of the method that handles the callback. In this case the `OATClass._get_module0_html()` method is executed and the template rendered by the method is returned.

Module data callback

The same architecture is used in the module data callback except for the added option of providing keyword arguments to the callback. Keyword arguments are added to the request URL in its query string. Arguments provided in the query string are available in the Django HTTP request object.

Example: “module number 0” in figure 5.4 requests data for itself. It needs to include in the callback that the data is to be for the last 10 days. To do so it requests the following URL, `eyk.hugfimi.is/asset/12/module/0/data?num_days=10`. The HTTP request generated with this callback is passed all the way to the module where the keyword arguments can be extracted. The `OATClass._get_module0_data` method is executed which returns the data as a HTTP response.

5.3. Core

The core app is a container for data types and abstract models. The abstract database models, like the ones shown in figure 5.5, are used as bases for the models implemented in the database interfaces. The abstract models are used to define common fields, used by all models of the given type.

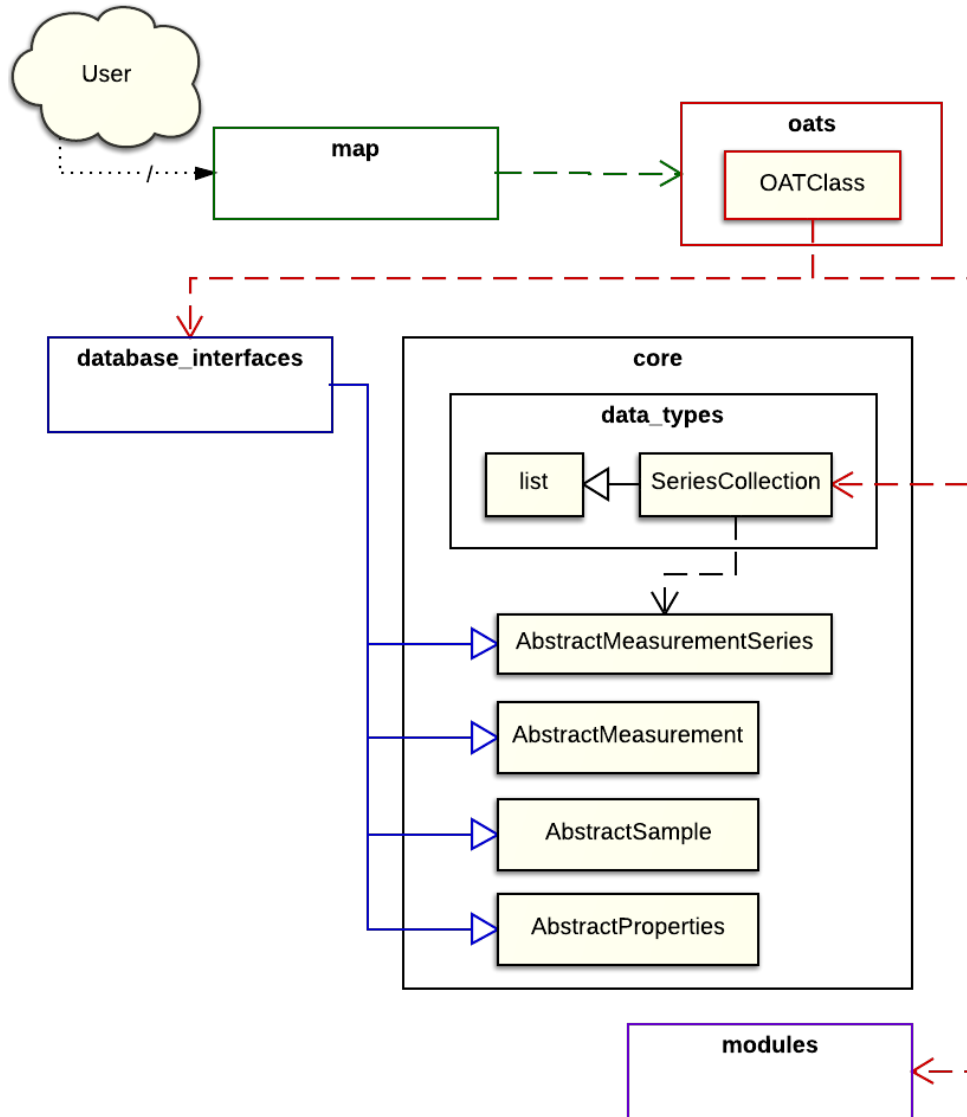


Figure 5.5: Architecture of the core component of EYK

As the database interfaces are plug-ins in the system their models are not known until runtime. By defining abstract models for the most commonly used data structures the system is able to use generic data types.

A data type can be thought of as a helper class that implements methods the models cannot implement themselves. Not all models need helper classes but a model like the measurement series does.

A data type, `SeriesCollection`, is implemented. This data type stores a collection of

5. Architectural design

measurement series and has some helpful methods to transform their data.

The abstract models shown in figure 5.5 are not a finite collection and are only displayed for clarification of the purpose of the core app. Their number will presumably grow as the number of database interfaces grows.

5.4. Modules

The modules of EYK are independent extensions of the framework. They provide the functionality available to OAT plugins when displaying the asset page. The EYK framework aims to offer a diverse selection of modules. To be able to accommodate a variety of modules their structural constraints on the modules are kept to a minimum.

There are some restrictions that the modules must adhere to. A module must be a valid Django app and list the path to the static files it depends on in their `__init__.py` module. The OAT plugin imports each module it uses and renders it by calling its views functions. The relationship between the modules and OAT plugins is shown in figure 5.6.

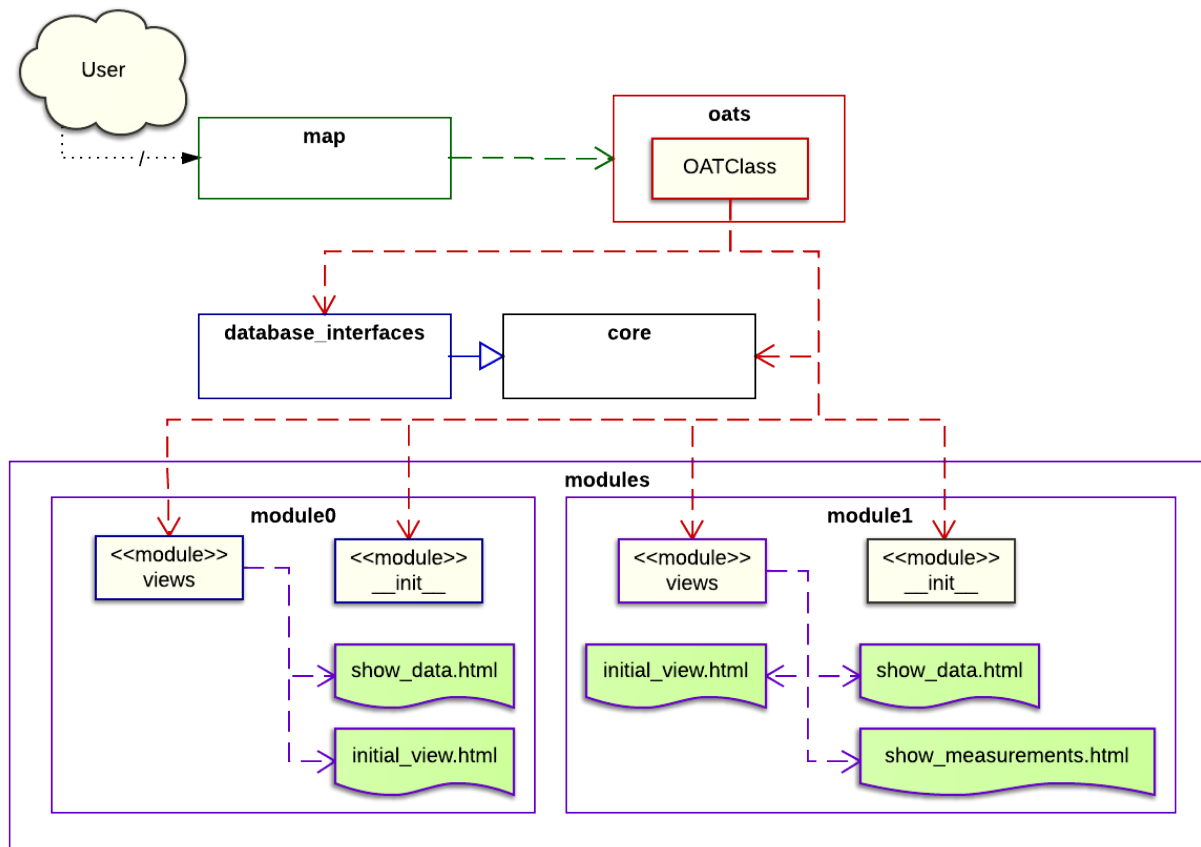
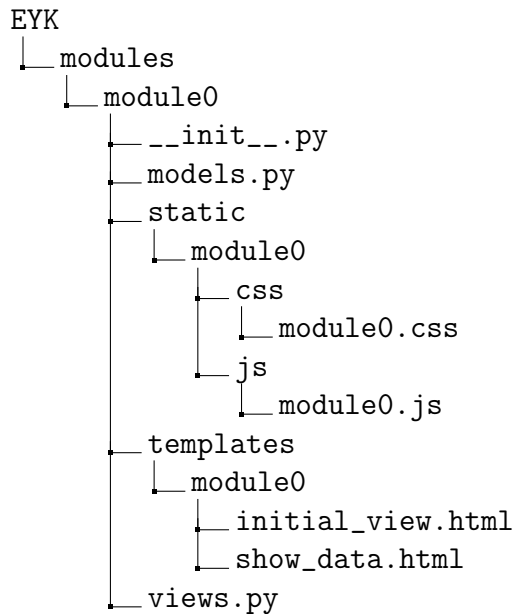


Figure 5.6: An example structure of two modules in the module collection

5.4.1. Structure

A typical Django app is described in foundations, section 4.2.1.4. The modules have a very similar structure but with a minor tweak. In addition to the default structure the modules use the `__init__.py` module to declare the static files they need. The `module0` module has the following structure:



The module has two static files, `module0/static/module0/js/module0.js` and `module0/static/module0/css/module0.css`. To ensure these files are loaded in the asset page before the module is loaded they are listed in `__init__.py` by declaring:

```

USED_SCRIPTS = ['module0/js/module0.js']
USED_STYLES = ['module0/css/module0.css']

```

If the module depends on other static files, they are simply added to these lists.

Modules can have as many templates as they need and their naming is not restricted by the architecture. A typical module has the two templates, like `module0`, one for its initial view and one to add data to the view.

5.5. Database Interfaces

The database interfaces are the second plug-in in the EYK framework. They gather and store data. Each interface consists of one or more database models and optionally a scraper or tasks. Figure 5.7 shows the architecture of two modules. This is the last component of the architecture to be revealed, a complete diagram of it is shown in appendix A.

As shown in 5.7, the database interfaces are primarily used by OAT plugins which is also

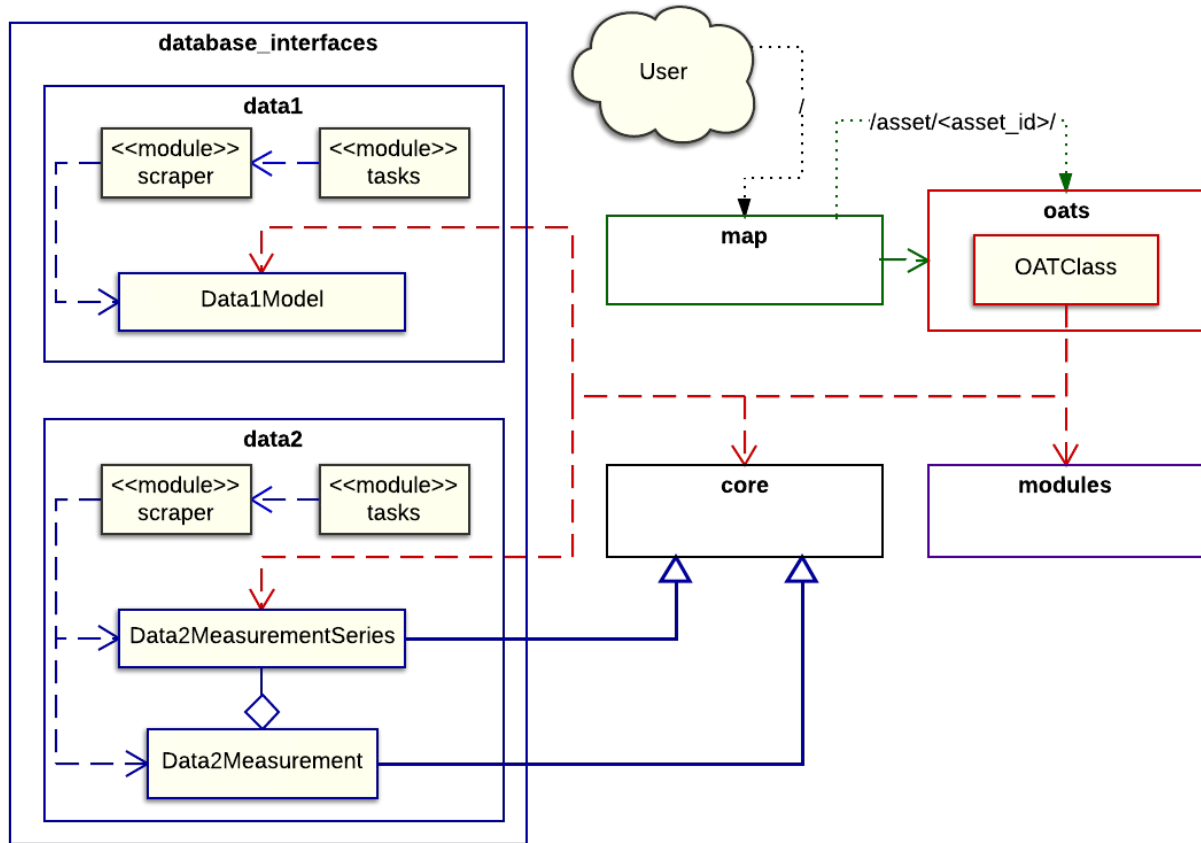


Figure 5.7: The architecture of the database interfaces

a plug-in in the EYK framework. What makes them a separate plug-in is their connection to the frameworks task scheduler.

Scraper and tasks The scraper module contains a functions to scrape data from external data sources and store it in EYK’s database. If the external data source is updated frequently, the scraper can be set to execute periodically by specifying it as a reoccurring task in the `tasks` module. To execute tasks, EYK uses Celery, a distributed task queue¹. Celery looks for `tasks.py` modules in available Django apps and executes the reoccurring tasks defined therein.

Not all database interfaces need to have a scraper and a task module. If a database interface only stores data which is collected through EYK, database models are sufficient.

¹Celery is described in Technology used section 4.2.3

6. Implementation

This chapter will provide a description of the implementation of the EYK framework. Included is the implementation of the map app, the oats app, the core app and select modules. The usage of the framework will be demonstrated in the following chapter.

6.1. Map

The map is where the user selects assets to work with. It is implemented using LeafletJS and along with maps from Google. Asset icons are shown on the map, grouped by their types.

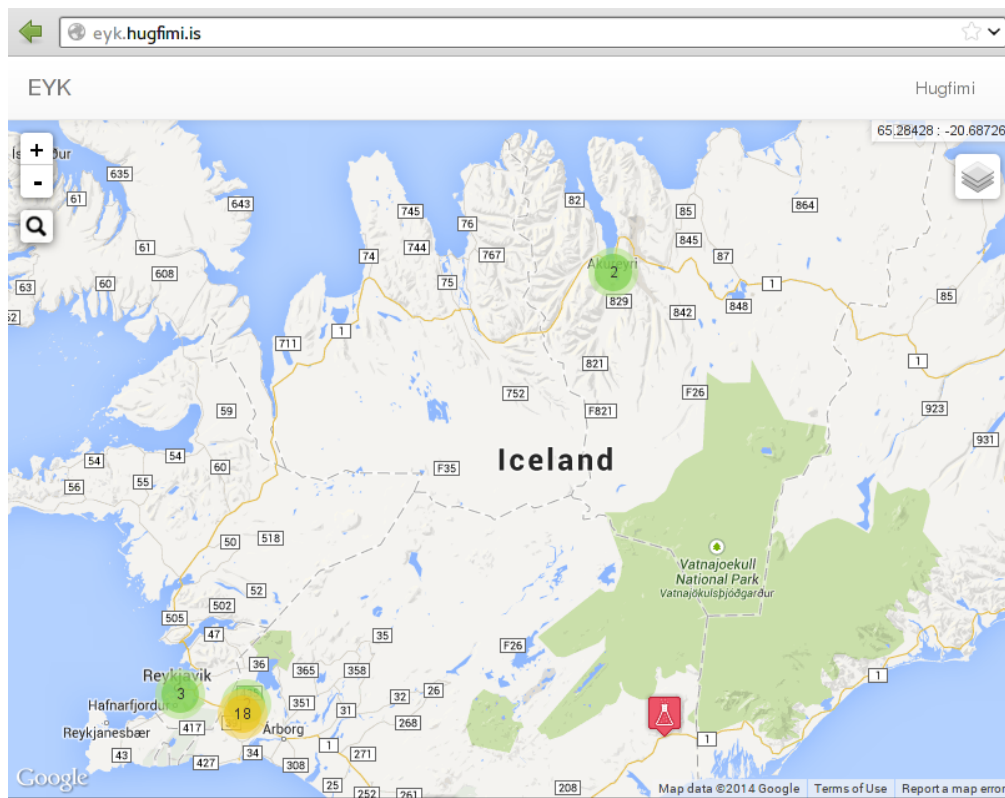


Figure 6.1: The initial view of the system

Multiple assets with high proximity get grouped together into a single icon. The yellow icon in figure 6.1 for example denotes 18 boreholes which are located within a small area of the map.

6. Implementation

Most of the configuration of the map is done in JavaScript on the client side. The backend of the map app consists of a single views module which contains 3 view functions, `map_page`, `markers` and `marker_groups`. These functions will be described in the following sections

6.1.1. Map page

The URL request for the map includes 5 optional parameters, `latitude`, `longitude`, `zoom`, `baselayer` and `layers`. These parameters are added to the query string of the URL in key value format. Together, these parameters define the position, zoom level, active base layer and visible overlays of the map. The values of these parameters are then extracted from the HTTP request passed to the `map_page` function, (listing 6.1).

```
1 def map_page(request):
    try:
2         latitude = float(request.GET['latitude'])
3         longitude = float(request.GET['longitude'])
4
5         zoom = int(request.GET['zoom'])
6         base_layer = request.GET['baselayer']
7         active_layers = request.GET['layers']
8
9     except (TypeError, MultiValueDictKeyError):
10        latitude = INITIAL_LATITUDE
11        longitude = INITIAL_LONGITUDE
12        zoom = INITIAL_ZOOM
13        base_layer = INITIAL_BASELAYER
14        active_layers = INITIAL_OVERLAYERS
```

Listing 6.1: Initialization of the variables of the map

If the query string is not provided, or malformed, the initial parameters of the map are used. These parameters are then sent as context to the rendering of the map template which proceeds to initialize the map in the users browser, listing 6.2.

```
1 context = {
2     'latitude': latitude,
3     'longitude': longitude,
4     'zoom': zoom,
5     'baselayer': base_layer,
6     'layers': active_layers
7 }
8 return render(request, "map/map.html", context)
```

Listing 6.2: Rendering map.html with contextual variables

6.1.1.1. The map.html template

The map.html template uses the header template from the shared libraries, eyk/header.html, see listing 6.3. This header is used both as the banner in the map page and as the module selector in the asset page.

```
1 {% include "eyk/header.html"%}
<div id="map"></div>
3 <script src="{% static "map/js/map.js" %}"></script>
```

Listing 6.3: Inserting the map header and creating a container for the map

A container is made for the map and the map.js script is imported.

The template then initializes the map by calling the initMap JavaScript function provided with map/js/map.js, listing 6.4.

```
<script type='text/javascript'>
2   initMap({{ latitude }}, {{ longitude }}, {{ zoom }}, '{{ baselayer }}');
   $(document).ready(function(){
4       addMarkers('{{ layers }}');
       });
6 </script>
```

Listing 6.4: Initializing the map via JavaScript

When the map has been initialized and displayed to the user, the markers are added to it by calling the addMarkers function, also provided by the map/js/map.js. These functions will be described in the next section

6.1.1.2. Initializing the map, initMap

The base layers are defined before the map is initialized. They are created using the Leaflet Plugins provided by Shramov [Shramov, 2014], listing 6.5.

```
function initMap(latitude, longitude, zoom, baseLayer) {
2   MAP.bases = {
       'Roads': new L.Google('ROADMAP', {detectRetina: true}),
4       'Terrain': new L.Google('TERRAIN', {detectRetina: true}),
       'Satellite': new L.Google('SATELLITE', {detectRetina: true}),
6       'Hybrid': new L.Google('HYBRID', {detectRetina: true}),
   };
}
```

Listing 6.5: Creating base layers

From the created base layers, the active base layer is selected based on the baseLayer argument, listing 6.6.

6. Implementation

```
1 // select active base layer
2 $.each(MAP.bases, function(obj, layer){
3     var layerType = String(layer._type).toLowerCase();
4     if(layerType == baseLayer){
5         MAP.activeBaseLayer = layer;
6         MAP.activeBaseLayerTitle = layerType;
7     }
8 });
```

Listing 6.6: Setting active base layer

The map is initialized in the map container provided in listing 6.3 and given. It is given an array of settings where, amongst other, the map center and zoom level are set. As the overlays (markers and markergroups) have not yet been requested from the server, the only layer added is the active base layer.

```
1 MAP.map = L.map('map', {
2     center: [latitude, longitude],
3     zoom: zoom,
4     layers: [MAP.activeBaseLayer]
5 });
```

Listing 6.7: Initialization of the map

The map is positioned as requested by `latitude`, `longitude` and `zoom` and the active base layer added to the map. To keep the current URL up-to-date as the user interacts with the map a selection of on-event-functions are created. When events, such as pan, zoom or base layer change are triggered the browser URL is updated to include the current state of the map. To update the URL a function, `updateBrowserHistory`, is provided, listing 6.8.

```
function updateBrowserHistory(){
1     window.history.replaceState(
2         "state",
3         "title",
4         '/?latitude='
5         + MAP.map.getCenter().lat
6         + '&longitude='
7         + MAP.map.getCenter().lng
8         + '&zoom='
9         + MAP.map.getZoom()
10        + '&baselayer='
11        + MAP.activeBaseLayerTitle
12        + '&layers='
13        + MAP.activeLayers.join(",")
14    );
15 }
```

Listing 6.8: The URL update function

This function replaces the query string of the URL for one that reflects the current state of the map. This design lets the user return to the last map state when he is finished working with an asset.

Once the map has been initialized and displayed the markers and markergroups are requested.

6.1.2. Markers and maker groups

Before markers can be added to the map, the groups they belong to need to be added as layers on the map. The marker groups are retrieved by issuing an Asynchronous, JavaScript and XML (AJAX) request via the `/markergroups/` URL, listing 6.9.

```
1 function addMarkers(activeLayersString){
    // get all markergroups in JSON format
3    $.getJSON("/markergroups/", function(markerGroups){
        // create markerGroups for map
```

Listing 6.9: The markergroup callback

This request is delivered to the `marker_groups` view function. A query for assets that have a distinct combination of asset type and organization is made to the database. This is done by chaining query functions. The first query is made for all available assets, a query for the ones that have distinct values for `type` and `organization` is chained to it and at last their related models are added to the query, listing 6.10.

```
def marker_groups(request):
2     org_type_distinct = Asset.objects.all().distinct('type', 'organization')
    org_type_distinct.select_related()
4
    org_type = []
6     for asset in org_type_distinct:
        org_type.append([asset.type.name, asset.type.plural,
8                        asset.type.icon_path, asset.organization.name])
10    return HttpResponse(simplejson.dumps(org_type), mimetype='application/json')
```

Listing 6.10: The marker_groups view function

By requesting related models (`select_related()`), the models referenced by foreign key, like `AssetType` and `Organization` are included in the data returned to the query.

Django database queries are lazy by default so the query is not made until the data is accessed. This means that fields in these foreign key objects can be accessed without requiring another database transaction. Without requesting related models the for loop in listing 6.10 would make three transactions to the database. A transaction would be made for the `Asset` model when the `org_type_distinct` list is iterated. Another transaction,

6. Implementation

this time for `AssetType` model, would be made when `asset.type.name` is accessed and the third, for the `Organization` model, when `asset.organization.name` is requested.

The query result is then filtered to minimize data transmission and returned as a JavaScript Object Notation (JSON) string.

When the marker groups are returned to the AJAX request each of them is processed and added to the map. The marker groups are grouped by their organization name, like shown in figure 6.2.



Figure 6.2: The maps layer control, shown with two marker groups

All marker groups are layers of the type `MarkerClusterGroup`, listing 6.11.

```
1 function createMarkerGroup(typeName, groupId, orgName){  
    var markers = new L.MarkerClusterGroup({ spiderfyDistanceMultiplier: true });  
3    MAP.map.addLayer(markers);
```

Listing 6.11: Adding a marker group to the map

This type groups single markers together when their proximity on the visible map is high. When the user zooms enough into the map, the cluster is expanded and its markers are shown at their locations. This is demonstrated in figure 6.3.

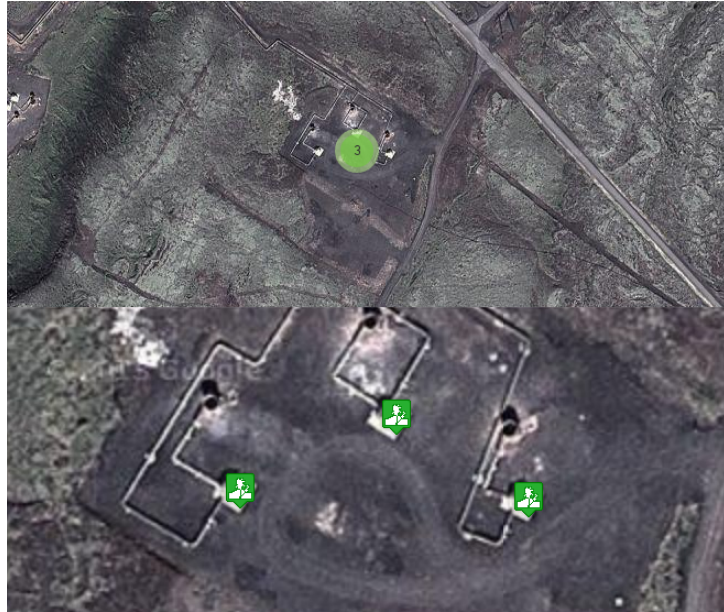


Figure 6.3: The clustering of markers

When all the marker groups have been created the markers are requested. They are also collected through a AJAX request which is handled by a function in the views module of the map app. The function collects all available assets and returns them to the request. As the markers get created and added to the map, each of them is given an “on click” function, see listing 6.12.

```

1 function addMarker(latitude, longitude, title, iconPath, groupId, assetId){
    var marker = new L.marker([latitude, longitude],
3       {title: title, icon: getIcon(iconPath)}
    );
5
    marker.on('click', function() {
7       var link = '/asset/'.concat(assetId.toString());
        window.location.assign(link);
9     }
    );
11  MAP.markerGroups[groupId].addLayer(marker);
}

```

Listing 6.12: Adding a single marker to the map

The function is called when a marker is clicked and redirects the user to the asset page of the selected marker. Finally the marker is added to its group and thereby displayed on the map.

6.2. Oats

The oats app contains a collection of OAT plugins. The OAT plugins define which modules are shown at which asset page, render the modules and provide them with data. This section will describe the implementation of the oats app and how it connects with the OAT plugins.

6.2.1. Viewing an asset

The asset page is handled by the views module of the oats app. Its URL (`/asset/<asset_id>`) contains a single variable, “`asset_id`”, which is passed as a argument to the views function, `oats.views.asset_page`, shown in listing 6.13.

```
def asset_page(request, asset_pk):  
2     asset = get_object_or_404(oats.models.Asset, pk=asset_pk)  
     oat = asset.oat_instance  
4     return oat.get_asset_page(request, asset)
```

Listing 6.13: Implementation of `oats.views.asset_default_view`

The process of extracting variables from URLs is described in chapter 4, section 4.2.1.2. The asset corresponding to the `asset_id` is retrieved using a built-in Django function. As the name implies, the `get_object_or_404` function, retrieves the model requested or returns a 404 error (page not found) to the request. The OAT plugin corresponding to the asset is imported and instantiated by using the `oat_instance` property of the `Asset` model and used to get the asset page and return it.

As the requested OAT plugin is unknown until at runtime it has to be imported and instantiated dynamically. This process and the one of rendering the asset page will be described in the following sections

6.2.1.1. Dynamically importing and instantiating an OAT plugin

The `Asset` model has a property, `oat_instance`, which imports and instantiates the OAT plugin of the respective asset. As stated in section 6.2, each OAT plugin is located at a unique path, composed from the names of its organization and asset type. This path is used to import the module of the OAT plugin, listing 6.14. The OAT plugin itself is then instantiated with the help the `inspect` module.

```
@property  
2     def oat_instance(self):  
         oat_module = self._import_oat_module()  
4         oat_class_name = self._get_oat_class_name()  
  
6         oat_classes = inspect.getmembers(oat_module, predicate=inspect.isclass)
```



```

8      for name, handle in oat_classes:
          if oat_class_name in name:
              return handle()

```

Listing 6.14: Determining oat path and class name

The `inspect` module is a Python module which can be used to list the members of modules at runtime [Python Software Foundation, 2013]. When inspecting the `oat_module` the `inspect.get_members` function is provided a predicate, informing it to only list classes. The list returned contains both the name and the handle of the classes found. The findings are then searched for a name matching `oat_class_name` and if found, an instance of the class is returned.

6.2.1.2. Requesting `oat.html`

Once an OAT plugin has been imported and instantiated the asset page for that class is rendered. All OAT plugin classes inherit from `OATBase`, which provides the `get_asset_page` method. This method renders the `oat.html` template with context provided by the OAT plugin, see listing 6.15.

```

2      def get_asset_page(self, request, asset):
          [scripts, styles] = self._get_imports()
          return render_to_response("oats/oat.html",
4                                     {'modules': self.MODULES,
                                     'module_labels': self._get_module_labels(asset),
6                                     'module_js_imports': scripts,
                                     'module_css_imports': styles
8                                     })

```

Listing 6.15: The implementation of `OATBase.get_asset_page`.

As described in the Architectural design chapter, the module apps list their static file dependencies in their `__init__.py` modules. These are gathered and provided as a part of the template context. Additionally the `MODULES` list, defined by the OAT plugin, and the corresponding labels are provided as context.

6.2.1.3. The `oat.html` template

The asset page has two visual components, the module selector and the module container. The module selector displays the labels of all the modules and the module container displays the selected module.

To initialize the asset page, the `oat.html` template does three things: it links to static files for the module apps, initializes the module selector and connects the module selector to the module callbacks.

Linking to the static files The template is provided with two context variables, `module_css_imports` and `module_js_imports`. Both of which are lists containing links to static files. To add these links, the template runs through them in a for-loop, listing 6.16.

```

2  <!-- importing stylesheets required by oat modules -->
    {% for css in module_css_imports %}
        <link rel="stylesheet" type="text/css" href="{% static css %}">
4  {% endfor %}
    <!-- importing scripts required by oat modules -->
6  {% for js in module_js_imports %}
        <script src="{% static js %}"></script>
8  {% endfor %}

```

Listing 6.16: Static files imported in `oat.html`

The Django template language allows iterating through lists as shown in section 4.2.1.4. The elements of each list are thereby added to the header of the page using their respective syntax

The module selector The module selector is designed as a navigation bar with a list of buttons. Each button is labeled with by an entry of the `module_labels` variable provided by the OAT plugin, see listing 6.17.

```

1  <ul class="nav navbar-nav">
    {% for label in module_labels %}
3      <li>
        <a id="module-{{ forloop.counter0 }}"
5        href="#module-{{ forloop.counter0 }}-pane" data-toggle="tab">
            {{ label }}
7        </a>
    </li>
9    {% endfor %}
</ul>

```

Listing 6.17: Laying out buttons for modules in `oat.html`

Each of the buttons is composed of an anchor within a list item. The anchors are identified by an id based on their index in the `module_labels` list and are linked to a pane with a similar id. When the anchor is selected the connected pane is shown in the module container. To hide the inactive module panes the `tab.js` library from Bootstrap is used. All the module panes are initialized hidden. When an anchor is selected, `tab.js` shows the pane it references by adding the `.active` CSS class to its properties. If another pane has the `.active` property its removed. This method of displaying module panes allows users to switch between modules without losing configuration made in the previous modules.

Connecting to module callbacks The `oat.html` template links to a JavaScript file, `oat.js`, which provides `inputModule`, a function to load a module into a module pane.

To connect the anchors in the module selector to this function a small JavaScript code is executed in the `oat.html` template, listing 6.18.

```

    $.each({{ modules|safe }}, function(index, name){
2      $('#module—' + index).click(function(){
        OAT.inputModule(index, name);
4      })
    });

```

Listing 6.18: The input module trigger used in `oat.html`

For each module in the `modules` list there is a anchor in the module selector. The id of the anchor is composed from the index of the module in the module list. By using the id of the anchor a click function is added to its properties that sends the `index` and `name` of the module as arguments to the `inputModule` function.

6.2.2. Displaying a module

To display a module selected in the module selector an AJAX request is sent for the module. The request is handled by the respective OAT plugin and the HTML of the module returned. The HTML is then inserted into the modules pane.

6.2.2.1. Requesting a module

Each time a module is selected in the module selector the `OAT.inputModule` functions is called. The function is provided with the index and name of the module to load as shown in listing 6.19.

```

OAT.inputModule = function(moduleIndex, moduleName) {
2   var moduleNS = OAT.createNamespace('OAT.'+moduleName);
   var $modulePane = $('#module—' + moduleIndex + '—pane');
4   if (!moduleNS.hasLoaded){
       var url = window.location.pathname + '/module/' + moduleIndex;
6       $modulePane.load(url, function(){
           moduleNS.hasLoaded = true;
8       });
   }
10   window.history.replaceState("state", "title",
       window.location.pathname + "#module—" + moduleIndex);
12 };

```

Listing 6.19: Implementation of `inputModule` in `oat.js`

This function takes care of creating a namespace for the module, requesting it via the module callback and adding it into the respective module pane. This process is described in the following paragraphs

Namespacing modules As the asset page can contain many modules, and some of the same type, some effort must be made in separating their JavaScript code. If all the modules define their functions and variables in the global context of the page chances are some of their names will collide. In the case of collision the first definition of the variable or function is overwritten. To avoid this the modules are assigned namespaces. Before a module is inserted, a namespace is created for it. The namespace is basically an empty JavaScript object which the module can use to define its code in. An example is provided in foundations, section 4.1.3. The `createNamespace` function essentially checks if the requested name space exists. If it does, it's returned, if it doesn't it's created and returned.

Requesting the module By using the name of the module, a unique namespace is created for it. To determine if the module has already been loaded, the `hasLoaded` boolean is used. The first time the module is loaded the variable is undefined but when it has loaded it is set as `true` to avoid loading the module again. The module callback URL is composed and requested. The response to the request is then implicitly loaded into the module pane. At last the current URL of the users browser is updated to denote which module is active.

6.2.2.2. Rendering the module HTML

The module load URL (`/asset/<asset_id>/module/<module_id>`) is handled by the `oats.views.view_module` function. This function obtains the OAT plugin like described in section 6.2.1.1 and returns the output of its `get_module` method, see listing 6.20.

```
1 def view_module(request, asset_pk, module_index):  
    asset = get_object_or_404(oats.models.Asset, pk=asset_pk)  
3    oat = asset.oat_instance  
    return oat.get_module(request, asset, module_index)
```

Listing 6.20: The implementation of `view_module` in `oats.views`

The `get_module` method is provided by the `OATBase` class. It uses the `module_index` provided to get the id of the module from the `MODULES` list defined by the `oat` subclass. With the name of the module requested the name of the method that renders the module can be determined. Method names that render module HTML follow the pattern: “`_get_<modulename>_html`”. The OAT plugin is then inspected for a method matching this name as shown in listing 6.21.

```
def get_module(self, request, asset, module_index):  
2    module_name = self.MODULES[int(module_index)]  
    method_name = "_get_%s_html" % module_name  
4    handle = self._get_method_handle(method_name)  
    return handle(request, asset, module_name)
```

Listing 6.21: The implementation of `get_module` in `OATBase`

The module callback method is then executed and its HTTP response returned.

6.2.3. Providing modules with data callback

To allow modules to make a callback for data, a similar approach as the module callback is implemented. A data callback URL (`/asset/<asset_id>/module/<module_id>/data?`) is provided which the modules can use to request data from the server. To supply parameters to the data request, modules can use the query string.

The data callback URL is handled by the `oats.views.get_data_for_module` function which is shown in listing 6.22. It propagates the call to the respective OAT plugin by calling `oat.get_module_data`.

```
def get_data_for_module(request, asset_pk, module_index):
2     asset = get_object_or_404(oats.models.Asset, pk=asset_pk)
    oat = asset.oat_instance
4     return oat.get_module_data(request, asset, module_index)
```

Listing 6.22: The implementation of `oats.views.get_data_for_module`

Much like with the `oat.get_module` method shown in listing 6.21 the `oat.get_module_data` method uses the module index to get the name of the method that compiles the data for the module in question. Its implementation is shown in listing 6.23.

```
def get_module_data(self, request, asset, module_index):
2     module_name = self.MODULES[int(module_index)]
    method_name = "_get_%s_data" % module_name
4     handle = self._get_method_handle(method_name)
    return handle(request, asset, module_name)
```

Listing 6.23: The implementation of `get_module_data` in `OATBase`

The `oat._get_<modulename>_data` method of the OAT plugin is provided the HTTP request of the callback, from where it can retrieve the parameters provided in the query string.

6.3. Core

The core contains abstract models and data types used by the rest of the system to simplify their implementation. The implementation of the components of the core will be described in the following subsections.

6.3.1. Abstract Models

Abstract models are implemented to minimize duplicate code amongst database interfaces. Abstract models in Django do not get created as tables in the database but are merged with the model that subclasses them into a single table [Django Software Foundation, 2013b]. This imposes some restrictions on what abstract classes can do. One which this system is affected by is the restriction of not allowing foreign key references to abstract models.

6.3.1.1. Measurement series

The `AbstractMeasurementSeries` are designed as a container for measurements. It has six fields, a foreign key reference to the asset the series pertain to, the date when the measurement series were performed and labels and units of the measurements it contains.

To access the measurements the measurement series contain a method, `data_as_list` is provided, see listing 6.24.

```
def data_as_list(self):
    #TODO (remove implicit dependency on "measurement_set".)
    data_list = []
    for entry in self.measurement_set.all():
        data_list.append([entry.x_value, entry.y_value])

    return data_list
```

Listing 6.24: The methods and properties of the abstract measurement series

This method retrieves all measurements which reference to the measurement series and returns their x and y values in as a list. When accessing all measurements referencing this model the `measurement_set` property is used. This property is dependent on the name of the model class being referenced [Django Software Foundation, 2013d]. In this case the `data_as_list` method depends on the model which implements the `AbstractMeasurement` being named `Measurement`. If the model would be named `DataMeasurement` this relation would become `datameasurment_set`. This design is less than ideal and needs to be investigated further.

6.3.1.2. Measurement

The `AbstractMeasurement` model contains a single measurement. It has two fields, `x_value` and `y_value`

As the measurement series model described in section 6.3.1.1 is an abstract model it cannot be referenced to by foreign key references. Because of this, the foreign key relationship must be added in the model that inherits from `AbstractMeasurment`.

6.3.1.3. Time series

The `AbstractTimeSeries` model is a container for timed measurements. It has many of the same fields as `AbstractMeasurementSeries` series although their `data_as_list` methods differ. The signature for time series is shown in listing 6.25.

```
1 def data_as_list(self, date_from, date_to):
```

Listing 6.25: The signature of the `data_as_list` method

The method accepts a range of dates which the returned list will contain. This is done as time series are a continuous collection which can get very large while measurement series are a singular event with a single time stamp.

6.3.1.4. Timed measurement

The `TimedMeasurement` model contains a single timed measurement. It has two fields, `date_time` and `y_value`. In an ideal setting this model would inherit the `AbstractMeasurement` model and override its `x_value` field. This however is not allowed, fields of abstract models cannot be overridden.

6.3.1.5. Properties

The `AbstractProperties` model stores key-value properties. It has three fields, a foreign key to the asset it pertains to, a dictionary of properties and a list of the keys of the dictionary, in the order they are to be shown.

6.3.2. Data Types

The data types are a collection of normal Python classes that manipulate the abstract models discussed in section 6.3.1. The currently available data types are: series collection, time series collection, table and time series table. These will be described in the following subsections.

6.3.2.1. Series collection

The `SeriesCollection` data type groups `AbstractMeasurementSeries` together.

The data type has many properties: `name`, `x_type`, `y_type`, `series_data`, `series_names` and `series_names_and_data`. Most are somewhat self explanatory and only implemented as syntactic sugar. The last three implement the access point to the measurement data. The `series_data` properties return the list of each measurement series in a list

6. Implementation

The `series_names` returns the name of all the measurement in a list and `names_and_data` adds them both to a list, like shown in listing 6.26.

```
1 >> my_collection.names_and_data
[['series1', [[x1, y1], [x2, y2], ..., [xn, yn]], ['series2', ...], ... ]
```

Listing 6.26: The form of the output of the `SeriesCollection.names_and_data` property

6.3.2.2. Table

The table data type accepts a series collection and converts it into a table where all measurements have been synchronized on either x or y axis. For *example*, the three series from table 6.1 are to be made into a table, ordered by the x column and ascending.

Series 1		Series 2		Series 3	
1	3	1	8	1	19
2	5	3	1	2	21
3	7	5	3	4	15

Table 6.1: Example series collection

Once processed by the table class the result would be as shown in table 6.2. The table uses a custom implementation of a stack to move through each measurement series and add their measurements to the right row.

x-column	Series 1	Series 2	Series 3
1	3	8	19
2	5		21
3	7	1	
4			15
5		3	

*Table 6.2: The series collection from table 6.1 converted into a **table***

6.3.2.3. Time series collection

The time series collection has the same purpose as the series collection although it made to store time series. It inherits from series collection but overrides the `series_data` property to accomodate for the difference in signature of the `data_as_list` method between `time_series` and `measurement_series` (listings 6.24 and 6.25)

6.3.2.4. Time series table

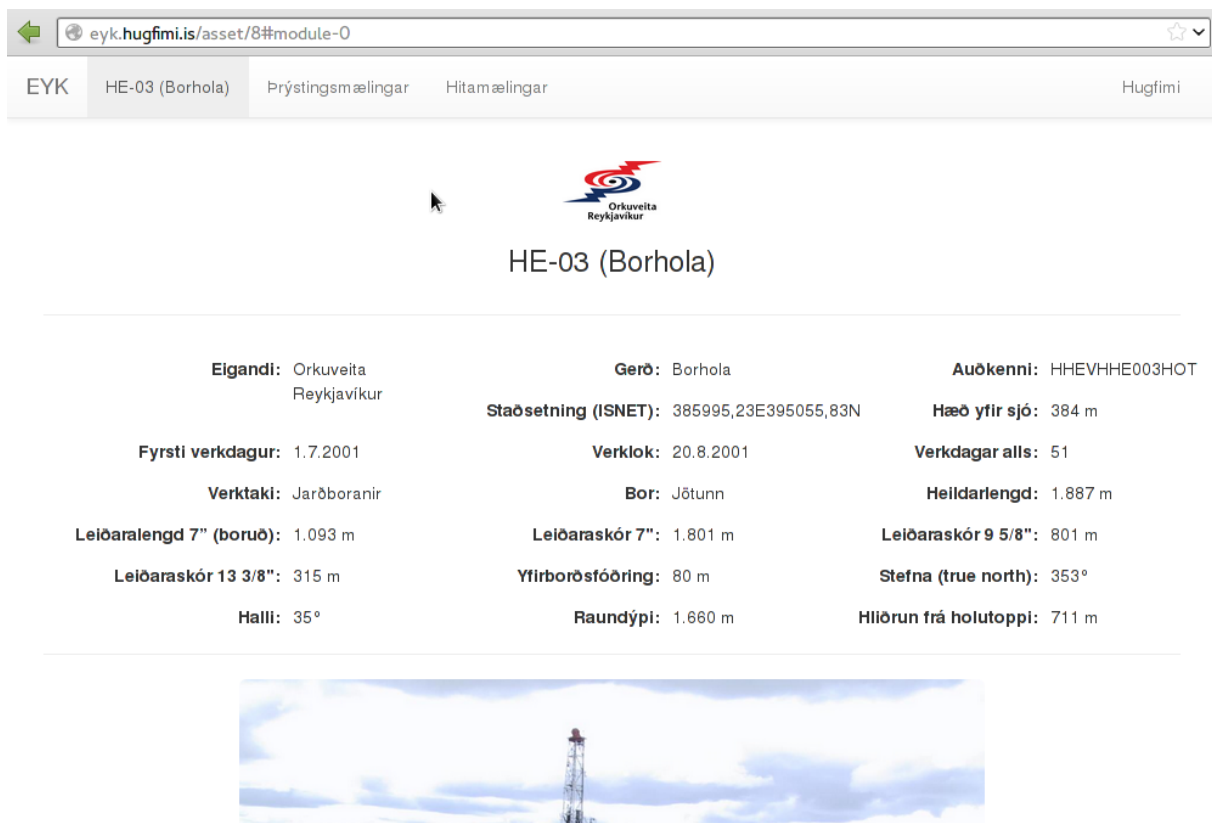
The time series table has the same function as the table but accepts time series collections.

6.4. Modules

During the course of this project, two modules were implemented. A properties module, which displays key-value properties of assets and a plot module, which visualizes measurement series and allows users to download raw measurements. These modules implementation will be described in the following subsections.

6.4.1. Properties module

The properties module app displays asset properties and image, if available. The properties, implemented for a borehole is displayed in figure 6.4.



Eigandi: Orkuveita Reykjavíkur	Gerð: Borhola	Auðkenni: HHEVHHE003HOT
Fyrsti verkdagur: 1.7.2001	Staðsetning (ISNET): 385995,23E395055,83N	Hæð yfir sjó: 384 m
Verktaki: Jarðboranir	Verklok: 20.8.2001	Verkdagar alls: 51
Leiðaralengd 7" (boruð): 1.093 m	Bor: Jötunn	Heildarlengd: 1.887 m
Leiðaraskór 13 3/8": 315 m	Leiðaraskór 7": 1.801 m	Leiðaraskór 9 5/8": 801 m
Halli: 35°	Yfirborðsfóðring: 80 m	Stefna (true north): 353°
	Raundýpi: 1.660 m	Hliðrun frá holutoppi: 711 m

Figure 6.4: An example rendering of the property module, shown with data from a borehole

The implementation of the properties module is quite simple. It consists of only one template, has no JavaScript or CSS style dependencies and uses no data types from the core. The structure of the module app is:

6. Implementation

```
properties
├── __init__.py
├── models.py
├── templates
│   └── properties
│       └── initial_view.html
└── views.py
```

As it has no static file dependencies the `USED_SCRIPTS` and `USED_STYLES` lists in `properties.__init__.py` are empty. The module has no need for its own database models and thus `properties.models.py` is empty. Both of these files do need to exist for the module to be considered a Django app.

6.4.1.1. Views

The `properties` module contains a single view which renders the `initial_view.html` template, listing 6.27

```
def initial_view(request, asset, namespace, asset_properties=None):
2     properties = [('Owner', asset.organization.name), ('Type', asset.type),
                  ('Identification', asset.legacy_id)]
4     if not asset_properties is None:
        for key in asset_properties.properties_order:
6             properties.append((key, asset_properties.properties[key]))

8     context = {"asset": asset,
                "module_namespace": namespace,
10             "asset_properties": asset_properties,
                'property_list': properties}
12     return render(request, "properties/initial_view.html", context)
```

Listing 6.27: properties.views

The view function accepts the three default arguments: `request`, `asset` and `namespace`. An optional `asset_properties` can be provided. If provided, the `asset_properties` argument must be an instance of a model which inherits the `core.models.AbstractProperties` model described in section 6.3.1. To use the asset properties object in a template some preprocessing is needed. The properties are converted to a list of tuples, where the first element of each tuple is the property and the second is the value. The common properties from the asset, organization name, asset type and asset legacy id are added to the `properties` list along with the properties from `asset_properties`. The asset, module namespace, asset property object and property list are provided as contextual variables when rendering the `initial_view.html` template.

6.4.1.2. The initial_view.html template

The properties are iterated and added to the HTML page, listing 6.28. Each property item is inserted as a description list (<dl>) and each given a 4 element wide column. As the whole width of the page is 12 elements this aligns the properties up in three columns when the browser width is over the medium width. When the browser window width is under medium width the properties are stacked in a single column. The nature of bootstrap's column structure is described in subsection 4.1.2.1 in Technology used.

```

1      {% for key, value in property_list %}
      <div class="col-md-4">
3        <dl class="dl-horizontal">
          <dt>{{ key }}:</dt>
5          <dd>{{ value }}</dd>
        </dl>
7      </div>
      {% endfor %}

```

Listing 6.28: Inputting properties

Some database interfaces provide an image for their assets. If available these images will be shown below the property list. The Django template language provides an easy way to write conditional markup. The image condition is shown in listing 6.29.

```

{% if asset_properties.image %}
2 <hr>
  <div class="row">
4    <div class="col-md-8 col-md-offset-2">
      
    </div>
8 </div>
  {% endif %}

```

Listing 6.29: Add image if available

6.4.2. Plot module

The plot module app displays graphs and tables of asset data. The plot module, implemented for the boreholes borehole measurements is shown in figure 6.5.

The plot module is somewhat more complex than the properties module. It accepts instances of the `MeasurementSeriesCollection` or `TimeSeriesCollection` data types and plots their data using the Highcharts plotting library. The structure of the plot module app is:

6. Implementation

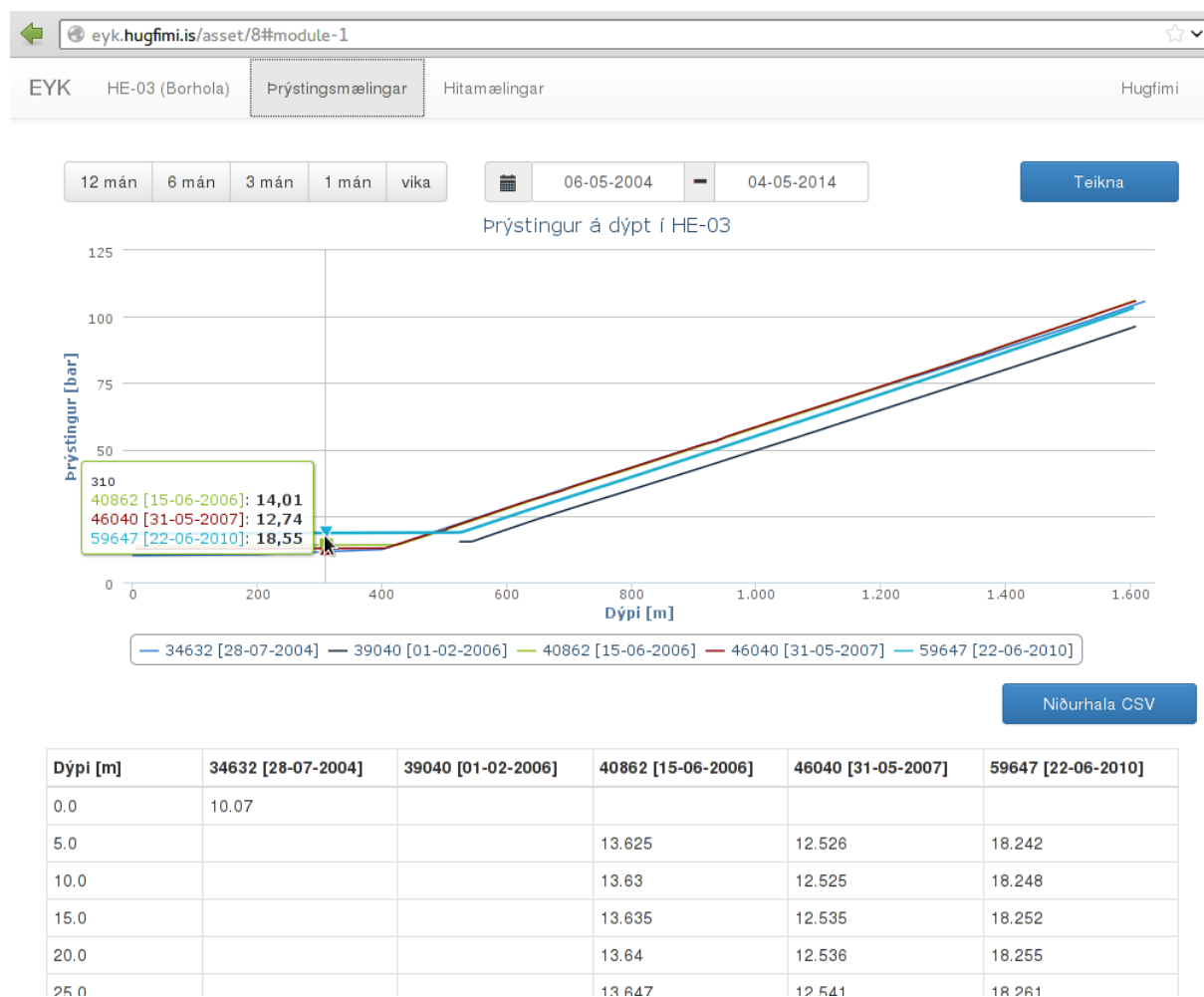
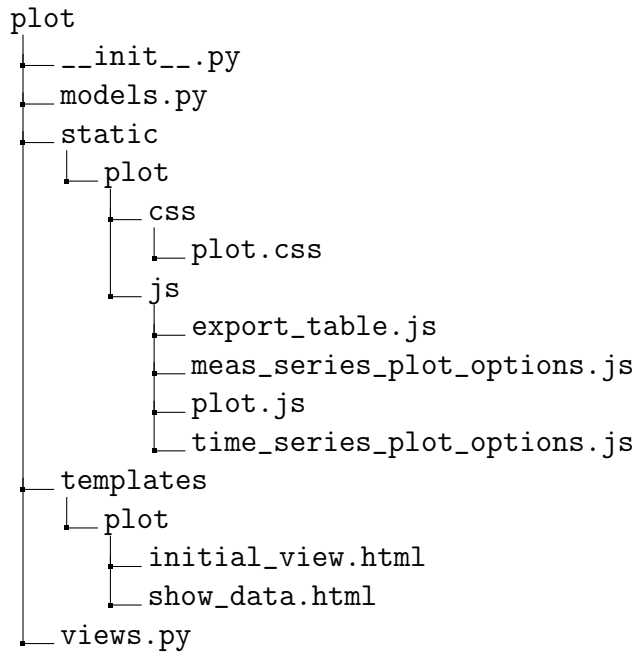


Figure 6.5: The plot module, implemented for borehole measurements



6.4.2.1. Static file dependencies

Before the module can be initialized the JavaScripts and CSS styles it depends on must have been imported. The OAT plugin relies on all modules to list their dependencies in the `__init__.py` module. The plot module dependencies are shown in listing 6.30.

```

1 USED_SCRIPTS = [
    'plot/highcharts/js/highcharts.js',
3    'plot/js/export_table.js',
    'plot/js/plot.js',
5    'utils/js/date_bar.js'
]
7 USED_STYLES = [
    "plot/css/plot.css"
9 ]

```

Listing 6.30: Static file dependency declaration for modules

6.4.2.2. Views

The plot module provides two views, `initial_view` and `show_data`. When using the plot module app, OAT plugins can specify the settings of the chart displayed by the module. This is done by supplying a Highcharts settings file. Some basic setting files are provided by the plot module which the OAT plugins can use but they can also provide their own custom settings file. The path to this settings file along with the number of days the plot will display initially are provided to the `initial_view` function, listing 6.31.

```

def initial_view(request, namespace, initial_date_range, settings_file):

```

6. Implementation

```
2     context = {
3         'module_namespace': namespace,
4         'initial_date_range': initial_date_range,
5         'options': settings_file
6     }
7     return render(request, "plot/initial_view.html", context)
```

Listing 6.31: The view function for the plot module initial view

In addition to the HTTP request and namespace, the `show_data` view accepts `series_collection` and `table`, listing 6.32.

```
1 def show_data(request, namespace, series_collection, table):
2     return render(request, "plot/show_data.html", {
3         'module_namespace': namespace,
4         'series_collection': series_collection,
5         'table': table,
6     })
```

Listing 6.32: modules.plot.views

The `series_collection` is an instance of either `SeriesCollection` or `TimeSeriesCollection` and is the data plotted on the graph. The `table` is an instance of either `Table` or `TimeSeriesTable` and will be used to populate the table.

6.4.2.3. Templates

The `initial_view` template starts by loading the `module_utils` library. This library contains the date bar and other HTML snippets that the module templates can employ. Once the library is loaded, the `date_bar` is inserted into the template and given two arguments. The first argument is the namespace of the module and the second a label to be put on its `action` button.

```
1 {% load module_utils %}
2 {% date_bar module_namespace=module_namespace label='Plot' %}
```

Listing 6.33: Adding the date bar

With the date bar in place a container for the chart is created. The module namespace is used to give it a unique id. The same thing is done when making a container for the table.

```
1 <div id='{{ module_namespace }}ChartContainer' class="row plot col-md-12">
2     </div>
3 <div id="{{ module_namespace }}Table" class="row table col-md-12">
4     </div>
```

Listing 6.34: Creating containers

The plot module is quite heavy on JavaScript, it uses three custom scripts and an external library. These scripts are all initialized at the end of the `initial_view` template. The module namespace is retrieved and used as a container for the remaining definitions, see listing 6.35.

```

1 <script src="{% static options %}"></script>
  <script type="text/javascript">
3     var ns = OAT.createNamespace("OAT.{{ module_namespace }}");
      // Create a plot object and give it the properties of Plot
5     ns.plot = {};
      Plot.call(ns.plot, "{{ module_namespace }}");
7     // Create a dateBar object and give it the properties of DateBar
      ns.dateBar = {};
9     DateBar.call(ns.dateBar,
        "{{ module_namespace }}",
11     ns.plot.load,
        {{ initial_date_range }});
13
      ns.plot.setOptions(PlotOptions());
15     ns.plot.drawChart();
17     ns.dateBar.onClick();
  </script>

```

Listing 6.35: Initializing JavaScript namespace and plot and date bar instances

Namespaces are created for the definitions of both the plot and date bar JavaScripts. These are then “called” into their namespaces using the `object.call` function. This method of instantiating objects is described in Technology used section 4.1.3. Once the plot and date bar are instantiated the date bar is initialized. The date bar needs to know which namespace to use, which function to call when its `action` button is clicked and how many days it is to display initially. The provided `PlotOptions`, imported in the first line of listing 6.35, are applied to the plot and the chart is drawn by calling `plot.drawChart()`. Finally the `onClick` function of the date bar is invoked to execute the `plot.load` function and trigger a module data callback.

Show data The show data template is rendered in response to the module data callback. It is provided with three context variables, `module_namespace`, `series_collection` and `table`. The template starts with creating a button to export the table. The buttons `onClick` event is handled by the `exportTable` function, provided by `export_table.js`, see listing 6.36.

```

1 <button type='button' class="btn btn-primary col-md-2 col-md-offset-10"
      onClick="exportTable({{table.header}}, {{table.body}})">
3     Download CSV
  </button>

```

Listing 6.36: Adding a button to download raw data

6. Implementation

The header and the body of the table is laid out using the `table.header` and `table.body` properties as shown in listing 6.37.

```
1      <thead>
      {% for entry in table.header %}
3      <th>{{ entry | safe }}</th>
      {% endfor %}
5      </thead>
      <tbody>
7      {% for row in table.body %}
          <tr>
9          {% for cell in row %}
              <td>{{ cell | safe }}</td>
11         {% endfor %}
          </tr>
13     {% endfor %}
      </tbody>
```

Listing 6.37: Composing the table

The `safe` tag is used when accessing template variables to keep the template renderer from escaping the content of the variable. This is safe to do here as the contents of the variables is only being displayed. When the table has been made the `series_collection` is added to the plot. First the module JavaScript namespace is retrieved and used to get the plot options, listing 6.38.

```
1 <script type="text/javascript">
    var ns = OAT.createNamespace("OAT.{{ module_namespace }}");
3    var plotOptions = ns.plot.getOptions();
    {% for series_name, series_data in series_collection.names_and_data %}
5        plotOptions.addSeries("{{ series_name|safe }}", {{ series_data|safe }});
    {% endfor %}
7    plotOptions.xAxis.title.text = "{{ series_collection.x_type }}";
    plotOptions.yAxis.title.text = "{{ series_collection.y_type }}";
9    plotOptions.title.text = "{{ series_collection.name }}";
    ns.plot.setOptions(plotOptions);
11    ns.plot.drawChart();
</script>
```

Listing 6.38: Applying data to plot

The `names_and_data` property of the `series_collection` variable is used to obtain the name and data for each series. The series are then added using a `addSeries` function which is provided by the `plotOptions`. The x- and y-axis labels are set as the x and y type properties of series collection and the collection name gets set as plot title. Finally, the `plotOptions` are applied to the chart and the chart redrawn.

6.4.2.4. JavaScript

The `plot.js` and `date_bar.js` scripts are essential to the function of the plot module and will be described in the following paragraphs.

date_bar.js The date bar is where the user selects the range of dates, data is to plotted for. The date bar is initialized with 3 variables, its module namespace, a function handle to a function that's run after the action button is clicked and the number of days from the current date, the `date_from` element will be set to. When the user clicks the action button of the date bar, its `onClick` function, shown in listing 6.39, is evaluated.

```

1  this.onClick = function () {
2      // The caller of this function is not datebar itself
3      // therefore "this" doesn't point to datebar.
4      // Use db instead.
5      db.disableAction();
6      db.externalOnClickFunction(
7          db.getDateFrom(),
8          db.getDateTo(),
9          db.enableAction
10     );
11 };

```

Listing 6.39: Handling on click events

The function ensures the user cannot click the action button again until the first request has been processed by disabling the action button. Then the function handle provided in the initialization of the date bar will be executed and provided with the currently selected dates. The function also receives the handle of the function to enable the action button again. In the case of the plot module, the date bar is given a handle to the `plot.load` function which will be described in the next paragraph.

plot.js The plot is structured like the classes described in Technology used, section 4.1.3. It stores three variables, the ids of the table and chart containers and its options. The container ids are used to populate these containers in the `plot.load` function, listing 6.40.

```

1  var Plot = function(module_namespace){
2      var plot = this; // stored for correct reference to plot object
3      this.tableId = '#' + module_namespace + "Table";
4      this.containerId = module_namespace + "ChartContainer";
5      this.options = Object();

```

Listing 6.40: Assembling container ids

When loading, the chart is set to provide feedback to the user by displaying the loading animation. The module data request URL is composed from the current URL and the

6. Implementation

query string parameters `date-from` and `date-to` are added. The plot is prepared for receiving new data by clearing out the old and a AJAX call is made for the plot data, see listing 6.41.

```
1  this.load = function(from, to, enableDateBar){
    // This function is not called within the context of the plot
    // therefore does the "this" variable not reference to it
    // The plot variable is used as a reference instead.
5  plot.chart.showLoading('Loading...');
    var hash = window.location.hash.replace('-', '/').replace('#', '/');
7  var url = window.location.pathname + hash +
        "/data/?date-from=" + from + "&date-to=" + to;
9  plot.clearData();
    $.ajax({
11     type: 'GET',
        url: url,
13     success: function(template) {
            $(plot.tableId).html(template);
15         enableDateBar();
        }
17     });
};
```

Listing 6.41: Loading data to plot

When the AJAX call has successfully returned the data, the `template` variable contains a rendered version of the `show_data.html` template. The whole markup is inserted into the table container where the JavaScript of the markup will automatically be evaluated. At last the date bar is enabled for user interaction.

7. Example system implementation

During this project an example system was implemented and is available at <http://eyk.hugfimi.is> (Firefox or Chrome recommended). This system contains three OAT plugins, a borehole, a power plant and an air quality sensor. These OAT plugins use the framework described in the previous chapter along with two custom database interface plugins. The following sections will describe how the OAT and database interface plugins for the air quality sensor were implemented as their implementation utilizes the most of the framework.

7.1. Database interface

The Icelandic environment agency publishes data from multiple air quality sensors from locations around the country. Data for 7 active sensors can be accessed via the agency's web page. The air quality measurements come from automatic sensors that upload their data with predetermined frequencies, ranging from a minute to an hour. This data is uploaded to a file server hosted by the agency where the data is stored in Comma Separated Values (CSV) files, named by year. Once new measurements are available they are appended to the file of the current year. This is a good example of where the periodic scraping architecture used in EYK comes handy.

Models First the models to store the data are developed. The `AbstractTimeSeries` and `AbstractTimedMeasurement` models from section 6.3.1 are used as bases for the `TimeSeries` and `TimedMeasurement` models. The `TimeSeries` model defines an extra field, `chemical`, in addition to those inherited from `AbstractTimeSeries` and imposes a uniqueness restriction on combinations of `chemical` and `asset`. This means that an asset can only have one time series for each chemical, see listing 7.1.

```
class Meta:
    unique_together = ('chemical', 'asset')
```

Listing 7.1: Imposing restriction of uniqueness on `chemical` and `asset`

The `TimedMeasurement` model adds a foreign key reference to the `TimeSeries` and adds a unique restriction on `series` and `date_time`, listing 7.2.

7. Example system implementation

```
1 class TimedMeasurement(AbstractTimedMeasurement):
    series = models.ForeignKey(TimeSeries, db_index=True)
3
    class Meta:
5        unique_together = ('series', 'date_time')
```

Listing 7.2: The implementation of TimedMeasurement

Scraper The scraper gathers data for each available chemical, for each sensor, starting at the newest measurements. When the scraper has found a measurement that violates the unique restraint on series and chemical imposed by the `TimedMeasurement` it assumes all subsequent data for the current chemical for the current sensor has been imported and moves on to the next chemical.

Tasks To execute the scraper periodically, a tasks module is made. The task module defines a reoccurring task as described in section 5.5. This task is auto-detected by the Celery task queue and is executed according to its specification. The scrape task for the air quality data defined is executed every day of the week, 10 or 40 minutes past every hour. The task is shown in listing 7.3.

```
1 @periodic_task(run_every=(crontab(hour="*", minute="10, 40", day_of_week="*")))
def scrape():
3     scraper.scrape_new()
```

Listing 7.3: The definition of the scraper task

These intervals are chosen as most sensors measure in half hour intervals. Their data is measured at the start of and half past every hour and is typically available online about 5 minutes later. This translates into a worst case delay of 35 minutes from the measurement being measured until it's available through EYK.

7.2. OAT plugin

The data collected by the database interfaces is displayed by the OAT plugin. The Air Quality (AQ) sensor OAT plugin uses two modules, a properties module and a plot module. The modules and their labels are declared in listing 7.4.

```
class AirQualitySensor(OATBase):
2     TYPE_NAME = 'Air quality sensor'
    BASE_MODULES = ['properties', 'plot']
4     MODULES = ['properties', 'plot']
    MODULE_LABELS = {'properties': TYPE_NAME, 'plot': 'Air quality measurements'}
```

Listing 7.4: The modules of the AQ sensor

7.2.1. Properties module

Implementing the properties module is straight forward. The AQ sensor has no properties other than the ones available from `Asset`, `AssetType` and `Organization` models. The properties module initial view is thus passed: the incoming request, the asset being viewed, the module namespace, and the `None` object is passed where a properties model would otherwise be passed, listing 7.5.

```

1  @staticmethod
   def _get_properties_html(request, asset, namespace):
3      return properties.views.initial_view(request, asset, namespace,
                                           None)

```

Listing 7.5: The response to the module callback request for the properties module

The rendered properties module is shown in figure 7.1.

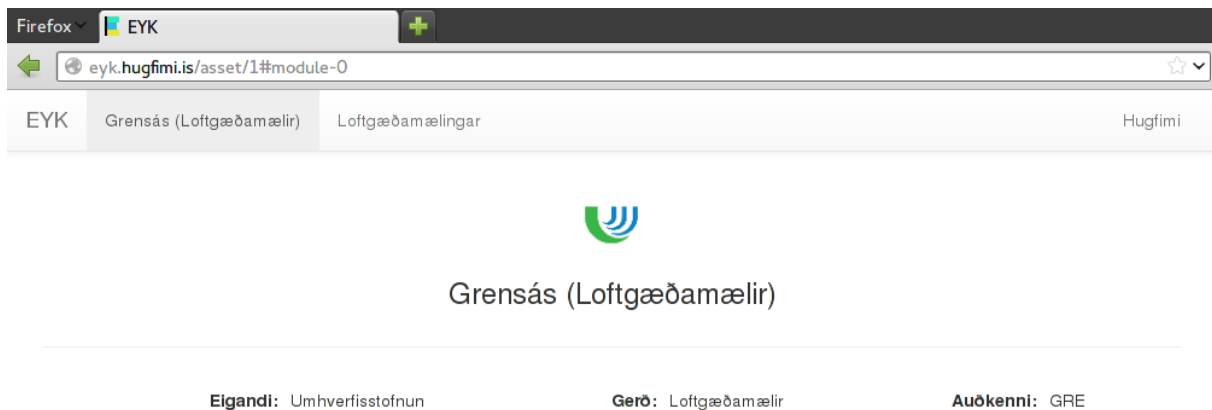


Figure 7.1: The Properties module of the air quality sensor

7.2.2. Plot module

The plot module is labeled “Air quality measurements” and displays data scraped by the database interface described in section 7.1.

Initial view As the frequency of AQ measurements is high, the module initially shows measurements for only the last 7 days. The `timeSeriesPlotOptions.js` settings file is used to configure the graph for displaying timed measurements and the initial view of the plot compiled and returned, listing 7.6.

7. Example system implementation

```
def _get_plot_html(request, asset, namespace):
2     initial_date_range = 7
    options = 'plot/js/timeSeriesPlotOptions.js'
4     return plot.views.initial_view(request, namespace, initial_date_range,
                                    options)
```

Listing 7.6: The response to the module callback request for the plot module

Show data When data is requested by the plot module via the module data callback, the `_get_plot_data` method responds. The dates requested are retrieved from the query string and converted to date time objects as shown in listing 7.7.

```
1     def _get_plot_data(request, asset, namespace):
        date_from = datetime.fromtimestamp(
3             int(request.GET['date—from']))
        )
5     date_to = datetime.fromtimestamp(
        int(request.GET['date—to']))
7     )
```

Listing 7.7: Date range of request retrieved from query string

The `TimeSeries` objects pertaining to the asset being viewed are requested from the database and added to a `TimeSeriesCollection`. The collection is provided with a name, and the range of dates requested, see listing 7.8.

```
    series = models.TimeSeries.objects.filter( asset=asset )
2
    collection = time_series_collection.TimeSeriesCollection(
4        'Air quality at %s' %asset.name.encode('utf-8'),
        series,
6        date_from,
        date_to + timedelta(days=1)
8    )
```

Listing 7.8: Collecting the TimeSeries for the asset being viewed

To include the end-date of the request the `date_to` parameter is incremented one day. At last the `TimeSeriesCollection` is used to generate a `TimeSeriesTable` and the two used to render the `show_data.html` template from the plot module app as shown in listing 7.9.

```

1      collection.x_type,
      collection.series_names,
3      order=table.DECENDING
    )
5      collection_table.compile()

7      return plot.views.show_data(request, namespace, collection,
                                   collection_table)

```

Listing 7.9: Creating a *TimeSeriesTable* from the *TimeSeriesCollection*

The rendered plot module is shown in figure 7.2.

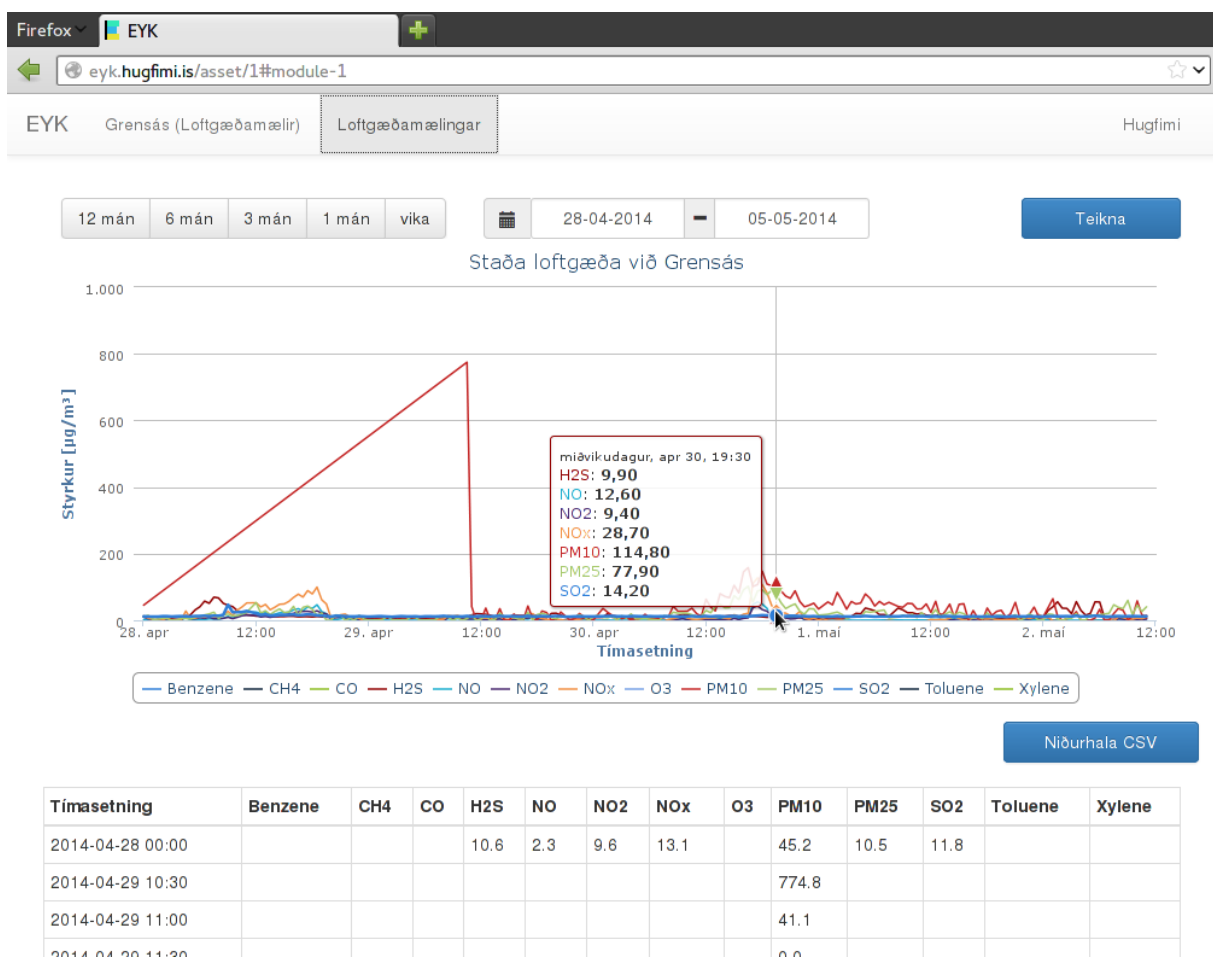


Figure 7.2: The plot module of the air quality sensor

8. Results

In section 2.2 a series of requirements were stated which the system is to fulfill. The status of requirements is shown in table 8.1.

Table 8.1: Fulfillment of requirements made in section 2.2

ID	Requirement description	Requirement met
Customer options		
1	Have data presented in a custom way	True
2	Have custom calculations performed	True
3	Select functionality for each data collection	True
Selectable functionality		
4	Store samples	False
5	Add samples	False
6	Display samples	False
7	Interface with the Watch software	False
8	Allow batch analysis of samples through Watch	False
9	Store time series	True
10	Import time series	False
11	Plot time series	True
12	Export time series as TSV	True
13	Perform calculations on time series	True
14	Store measurements	True
15	Import measurements	False
16	Plot measurements	True
17	Export measurements as TSV	True
18	Perform calculations on measurements	True
Improvements over Gagnor		
19	Be accessible without third party software	True (Browser only)
20	Be accessible to data collectors on site	True
21	Identify assets in an intuitive way	True
Further requirements		
22	Be able to interface to any accessible data source	True

8.1. Customer options

1. Have data presented in a custom way: The customers can have their data presented in custom way by either: requesting a custom setting file for the plot module app or, if the desired data presentation does not fit the plot module app, request a custom module app.

2. Have custom calculations performed: Custom calculations can be performed as data is scraped by the scrapers of the database interfaces. The scrapers are custom made for each data source which allows for custom treatment of data therein. This is demonstrated in the example system where depth values are corrected for borehole measurements.

3. Select functionality for each data collection: The architecture of the EYK framework is focused on assets rather than data collections. Customers can indirectly select functionality for data collections by selecting the functionality each asset offers through its modules.

8.2. Selectable functionality

4-8 Samples: The EYK framework does as of this writing support managing samples. This however will be implemented by adding an abstract model for samples and a sample module app. The timeline of these additions is discussed in the outline section of the next chapter.

9. Store time series: The framework can store time series. Abstract database models were developed to accommodate time series data. These were also used in the example system where air quality measurements are stored as time series.

10. Import time series: This requirement is currently not fulfilled by the framework. This however will be implemented by the development of a data collection module app before the first release of the software.

11. Plot time series: The framework can plot time series. A plot module app was developed which can be configured to plot time series. This is demonstrated in the example system where the air quality measurements are plotted.

12. Export time series as TSV: The framework can export time series as TSV. This is implemented in the plot module app and demonstrated in the example system where the air quality measurements are displayed as a table which is downloadable.

13. Perform calculations on time series: This is supported by the framework in the same way as requirement 2 is supported, through the database interfaces.

14. Store measurements: The framework can store measurements. Abstract database models were developed to accommodate measurement data. These were also used in the example system where borehole measurements are stored as measurements.

15. Import measurements: Much like requirement 10, this has not yet been fulfilled. It will be fulfilled by the same data collection module app which will be developed before the first release of the software.

16. Plot measurements: The framework can plot measurements. A plot module app was developed which can be configured to measurements. This is demonstrated in the example system where the borehole measurements are plotted.

17. Export measurements as TSV: The framework can export measurements as TSV. This is implemented in the plot module app and demonstrated in the example system where the borehole measurements are displayed as a table which is downloadable.

18. Perform calculations on measurements: This is supported by the framework in the same way as requirement 2 is supported, through the database interfaces. This is also demonstrated in the example system where depth values are corrected for borehole measurements.

8.3. Improvements over Gagnor

19. Be accessible without third party software: The framework is not accessible without a third party software. The framework is web based and thus requires the user to use a browser to connect to it. Even so, most operating systems do include a browser and many users are more native to their browsers than any other software. This requirement is therefore considered fulfilled.

20. Be accessible to data collectors on site: The system can be accessed by data collectors on site. This is a question of how the system is installed at the customer and

8. Results

whether there is a internet connection at the data collection site. No user identification is included in the system which for some customers might be unacceptable as their data would be publicly accessible. To remedy this, user identification will be added to the framework before release.

21. Identify assets in an intuitive way: Assets are identified visually by their geographical locations which is considered highly intuitive by the developers of the framework. For those users who are accustomed to using the string based identification, the assets can be searched for by these ids.

8.4. Further requirements

22. Be able to interface to any accessible data source: This is supported by the framework through the database interfaces as a custom interface is created to each data source. The example system demonstrates this as it interfaces with two distinct data sources.

9. Conclusions

This thesis has described the design and development of the EYK framework. A large part of the requirements set for the system were met while others are pending implementation. Several plug-ins were implemented as a demonstration of the functionality of the framework and that example is available online at <http://eyk.hugfimi.is>. The development of these plugins demonstrates how flexible the framework is and how much functionality can be achieved with little code. Some improvements must be made to the system before it is first released, which are discussed in the next section. The outlook of the project is discussed thereafter where the project road map is discussed and future releases outlined.

9.1. Improvements

The framework did not meet all the requirements made in section 2.2. To do so, the functionality of the framework must be extended. Requirements 4-8 can be met by adding a sample module to the module collection and requirement 20 can be further met by adding user authentication.

Furthermore, not all modules described in section 3.2.1 were implemented. The sample module, component module and data collection module were not implemented during this project but need to be implemented for the system to be considered a viable replacement for Gagnor. The method of implementing the subtypes described in section 3.2 has, as of this writing, not been developed but must be added to provide flexibility in the definitions of OAT plugins.

Further enhancements that can be made to the system is adding the user location to the map. This would further improve the use of the map and help data collectors locate the asset they are collecting data from.

9.2. Outlook

The development of the EYK framework continues and it will be offered as a data management solution by Hugfimi ehf. Hugfimi will offer services in creating plugins for the framework and custom modules. A roadmap for the planned feature releases for the EYK framework is displayed in figure 9.1.

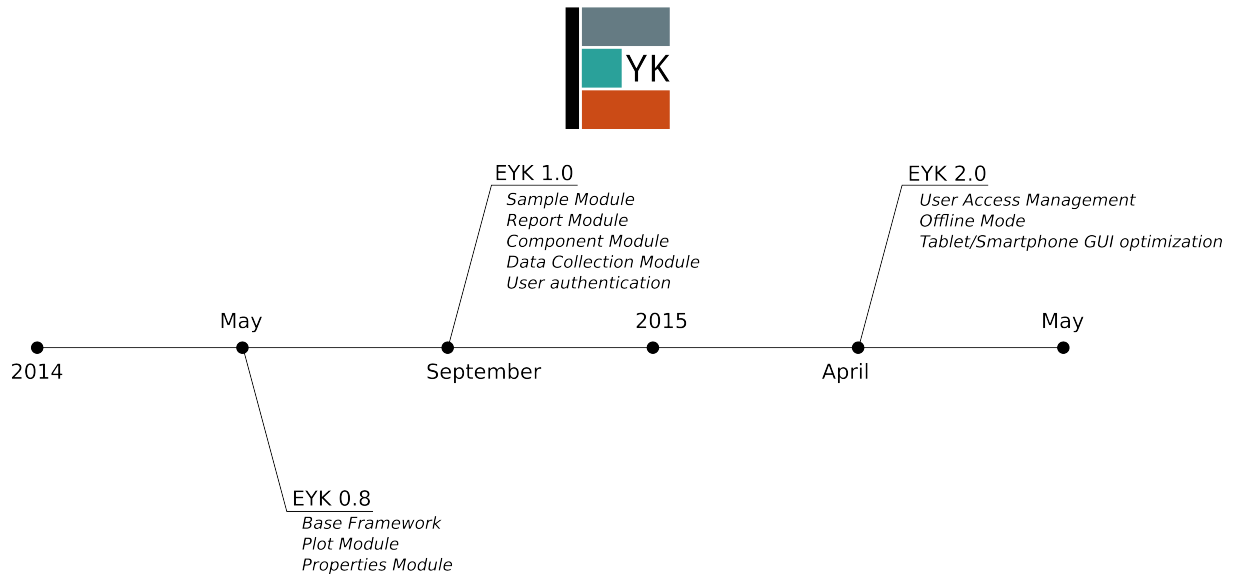


Figure 9.1: Planned feature releases of EYK

9.2.1. Release 1.0

Release 1.0 of EYK is scheduled in September 2014. In addition to the modules described in this thesis the release will include a report module and user authentication. The report module is designed to generate technical report templates. The templates will have their sections, figures, tables and data made and placed so users can focus on analyzing and writing.

9.2.2. Release 2.0

Release 2.0 of EYK is scheduled in April 2015. This release includes user access management, an offline mode and optimizations of user interface in regard to small screens.

User access management: This addition is intended to allow customers to specify which parts of the system are accessible to which users.

Offline mode: The offline mode is designed to allow data collection from areas where no internet connection is available. Data is to be stored on the users device and then synced with EYK's server once connection is made.

Bibliography

- Vladimir Agafonkin. Leaflet, an open-source javascript library for mobile-friendly interactive maps, April 2013. URL <http://www.leafletjs.com>.
- Stefán Arnarsson, Sven Sigurðsson, and Hörður Svavarsson. Watch, a chemical speciation program, April 2010. URL <http://www.geothermal.is/software>.
- Django Software Foundation. Django httpresponse objects, November 2013a. URL <https://docs.djangoproject.com/en/1.6/ref/request-response/#httpresponse-objects>.
- Django Software Foundation. Django abstract model documentation, November 2013b. URL <https://docs.djangoproject.com/en/1.6/topics/db/models/#abstract-base-classes>.
- Django Software Foundation. Django http request documents, November 2013c. URL <https://docs.djangoproject.com/en/1.6/ref/request-response/#httprequest-objects>.
- Django Software Foundation. Django, queries documentation, November 2013d. URL <https://docs.djangoproject.com/en/1.6/topics/db/queries/#following-relationships-backward>.
- Django Software Foundation. Django settings, November 2013e. URL <https://docs.djangoproject.com/en/1.6/topics/settings/>.
- Django Software Foundation. Django url dispatcher, November 2013f. URL <https://docs.djangoproject.com/en/1.6/topics/http/urls/>.
- Django Software Foundation. Geodjango, November 2013g. URL <https://docs.djangoproject.com/en/1.6/ref/contrib/gis/>.
- Colin Harrison. Xming x server, April 2014. URL <http://www.straightrunning.com/XmingNotes/>.
- Highsoft. Highcharts js, November 2013. URL <http://www.highcharts.com/products/highcharts/>.
- Mozilla Developer Network. Function.prototype.call(), February 2008. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call.
- Mozilla Developer Network. var hoisting, April 2014. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var#var_hoisting.

BIBLIOGRAPHY

- Mark Otto and Jacob Thornton. Bootstrap v3.0.0, August 2013a. URL <http://getbootstrap.com>.
- Mark Otto and Jacob Thornton. Bootstrap v3.0.0, August 2013b. URL <http://getbootstrap.com/css/#grid>.
- Python Software Foundation. inspect - inspect live objects, October 2013. URL <https://docs.python.org/2.6/library/inspect.html>.
- Ken Schwaber and Jeff Sutherland. Scrum guide, July 2013. URL <https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf#zoom=100>.
- Pavel Shramov. Leaflet plugins, January 2014. URL <https://github.com/shramov/leaflet-plugins>.
- Ask Solem. Celery: Distributed task queue v3.1, November 2013. URL <http://www.celeryproject.org/>.
- The Object Management Group. Uml 2.0 infrastructure, August 2010. URL <http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/>.
- The PostgreSQL Global Development Group. The postgresql licence, May 2014. URL <http://www.postgresql.org/about/licence/>.

Appendices

A. Diagrams

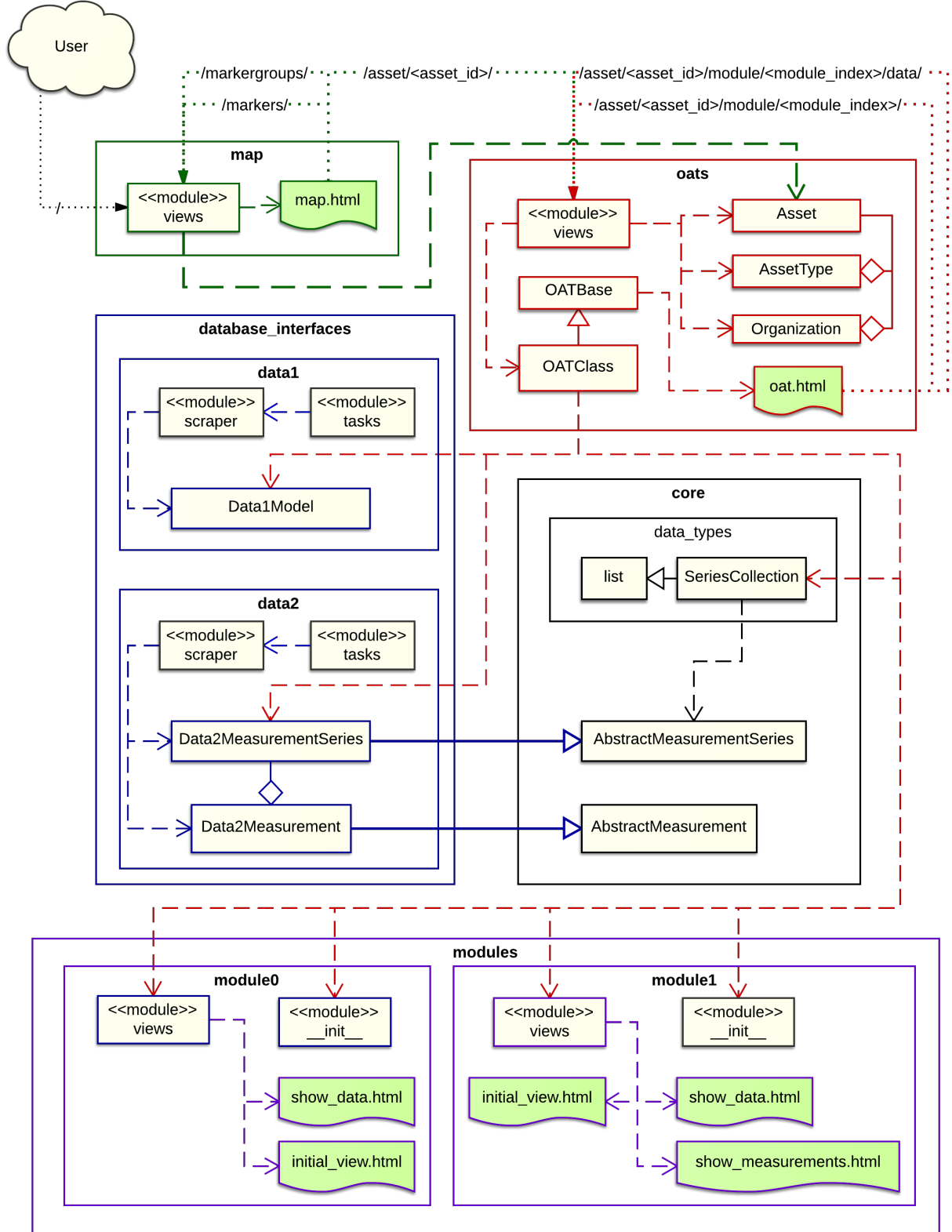


Figure A.1: The complete architecture of EYK

B. Contributions

At the start of development decisions had to be made on tools, methodology and project management. Both developers researched these topics and decided jointly on each. The outline of the architecture of the software was also commonly developed. Some architectural aspects were changed during implementation at which point both developers were involved.

The contribution of each developer to the implementation of the packages discussed in this thesis is as shown in table B.1. The testing of the system is not included in these percentages.

Table B.1: Contribution of each developer to specific packages, all tests excluded

Package	Bjarki	Helgi
core	75%	25%
database_interfaces	100%	0%
eyk	25%	75%
map	25%	75%
modules	75%	25%
oats	100%	0%