REDESIGN OF FREENEST WEB INTERFACE

Riku Hokkanen

Thesis

November 2012

Degree Programme in Information Technology

Technology, communication and transport

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN

KUVAILULEHTI

Tekijä(t) HOKKANEN, Riku	Julkaisun laji Opinnäytetyö	Päivämäärä 30.11.2012
		Sivumäärä 33
		Verkkojulkaisulupa myönnetty (X)
Työn nimi Redesign of FreeNest Web Interface		
Koulutusohjelma Ohjelmistotekniikka		
Työn ohjaaja(t) PELTOMÄKI, Juha		
Toimeksiantaja(t) RINTAMÄKI, Marko		

Tiivistelmä

FreeNest on selaimella verkossa toimive ohjelmistonkehitysympäristö joka on kehitetty SkyNest – projektissa Jyväskylän Ammattikorkeakoulussa. FreeNestiin kuuluu useita selainkäyttöisiä avoimen lähdekoodin ohjelmistoja jotka ovat yhdistetty yhteen kokonaisuuteen sivuihin lisätyllä yhtenäisellä käyttöliittymällä, automatisoidulla asennuksella ja yhdistetetyillä käyttäjätunuksilla.

FreeNestin lisäämässä käyttöliittymässä on muokattava navigointivalikko ja muita ominaisuuksia kuten yksinkertainen viestintäjärjestelmä. Opinnäytetyön tarkoituksena on suunnitella eri sovelluksia yhdistävä käyttöliittymä uudestaan niin että sitä on helpompi ylläpitää ja että siihen olisi helpompi lisätä uusia ominaisuuksia. Lisäksi opinnäytetyö tarjoaa muita parannusehdotuksia.

Opinnäytetyössä ei toteuteta uutta versiota käyttöliittymästä eikä suunnitella sitä yksityiskohtaisella tasolla, vaan tarjotaan yleisemmän tason ohjeita uuden käyttöliittymän suunnitteluun. Opinnäytetyö perustuu kirjoittajan kokemuksiin SkyNest – projektissa ja erilaisten JavaScript – ratkaisuiden tutkimiseen. Opinnäytetyössä käydään läpi käytettyjä teknologioita ja ehdotetaan MVC – rakennetta joka perustuu tiettyjen JavaScript – kirjastojen käyttöön.

Avainsanat (asiasanat)

FreeNest, SkyNest, JavaScript, AJAX, HTML

Muut tiedot

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖN

KUVAILULEHTI

Author(s)	Type of publication	Date
HOKKANEN, RIku	Bachelor´s / Master's Thesis	30.11.2012
		Pages 33
		Permission for web publication (X)

Title Redesign of FreeNest Web Interface
Degree Programme Software Engineering
Tutor(s) PELTOMÄKI, Juha
Assigned by RINTAMÄKI, Marko

Abstract

FreeNest is a web-based software development environment developed by SkyNest project at Jyväskylä University of Applied Sciences. It consists of open source tools combined with a web interface that is attached to most pages on FreeNest. The interface's main feature is an editable navigation menu, and it has other features such as simple chat system. The aim of the thesis is to redesign the interface to make it easier to maintain, add new features to, and to offer other possible improvement suggestions.

The thesis does not implement the redesign nor go very deeply into architecture, but gives general guidelines on how the interface could be improved. It is based on the author's experiences on SkyNest project writing some parts of the current version of the web interface, and research on JavaScript web solutions. The thesis covers the technologies used, and proposes a MVC framework based on a set of JavaScript libraries. It also contains other suggestions on new features and generic fixes.

Keywords FreeNest, SkyNest, JavaScript, AJAX, HTML
Miscellaneous

Contents

Figures

Terminology

AJAX

Asynchronous JavaScript and XML. A technique used in web applications that makes it possible for clients (browsers) to communicate with servers asynchronously, i.e without reloading the page. (Mozilla Developer Network)

ALM

Application Lifecycle Management is a way to manage software development process from the initial idea to post-release maintenance and publishing using ALM tool collections. (Chappell, 2008)

API

Application Programming Interface. API is an interface that specifies how software components communicate with each other on source code level. (Wikipedia)

DOM Core

Document Object Model. DOM Core is an API used to navigate and manipulate mainly HTML and XML – documents. It is extended by HTML and XML DOM that add more specific descriptions of the corresponding language structures. (W3C)

FreeNest

FreeNest is a product development platform that consists of many open-source products integrated together via an interface overlay and underlying solutions such as shared user information.

HTML

HyperText Markup Language is a language used to describe structure of a document, such as links to other documents, headers and tables. It is the language used on web pages. (W3C)

JavaScript

A scripting language commonly used to improve web page functionality – especially with AJAX.

MVC

Model View Controller. A common programming architecture that describes a separation of program into the namesake parts to separate data, data display and user interaction to achieve better code reusability, easier maintenance and other benefits.

Python

A programming language that is commonly used both for independent programs and as a scripting language.

Server

A platform used to share resources and services, such as a computer running server software to host web pages.

SkyNest

SkyNest is a team that develops FreeNest and cloud software solutions.

URI

Uniform Resource Identifiers are strings that identify resources. URIs can be Uniform Resource Locators (URL) that specify the location and method of accessing the resource, or Uniform Resource Names (URN) that identify the resource without telling how it can be accessed. (W3C)

Web client

An application that accesses a server, such as web browser.

1. INTRODUCTION

1.1 FreeNEST

FreeNEST Portable Project Platform is a product of JAMK SkyNEST Project, and it is essentially a web-based project environment. It combines multiple open-source tools under a single package that is easy to install and can be customized for the needs of different project groups. While currently FreeNEST has mostly tools for IT projects, its purpose is to cater for a wider variety of fields. A great deal of the work is aimed towards ease of use, so that users can focus on their own projects and and learn to use the FreeNEST tool set naturally.

While the tools come from different developers, FreeNEST core package combines them under a single user management and inserts a top bar menu which eases navigation between tools and makes the project work flow more coherent. FreeNEST also adds web tools of its own, such as project dash board, control panel and Git administrator tools.

1.2 FreeNEST web interface

The web interface, which is the focus of this thesis, refers to the top bar that is present on most FreeNEST's tools. At the very basics it is a navigation menu containing links to all installed FreeNEST features. It does contain other features, however. At the time of writing it also contains a basic messaging system, team mood gallup and a script that turns keywords to links to FreeNEST wiki's corresponding topic. There is also an editing feature for simple changes to the menu.

This thesis aims on improving the core web features of FreeNEST - the top bar and other basic functionalities included in every installation. The focus is not on the visible interface, but on easing the upkeep and continued development by making the underlying code more modular and clearer. This thesis will not create a final product, but the framework and plans for continued development.

The current FreeNEST web interface is focused on the visible top bar, with the other features heavily depending on it and tied on a code level – most of the JavaScript is in

a single file, and would be hard to separate. This thesis aims for a more flexible approach where it is possible to easily add and remove features from the interface.

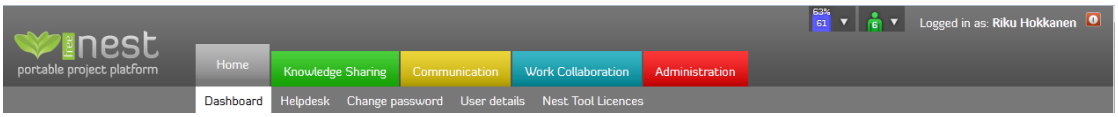

Figure 1: Part of the current FreeNest interface (old theme)

1.2. History

The basic idea behind FreeNEST has been the same since it first started, originally under the name Nest – a collection of open source tools that cover a software project's management needs. The original version was mostly created by one man, Marko Rintamäki, with help from friends and colleagues, while working at Ixonos. During that time Nest was developed to a point where it had most of the open source tools that the current one uses, only with less ease of use and integration. (Rintamäki, 2012)

Later on JAMK started a project called SkyNest, which at the time was mainly focused on improving Nest. At first the project consisted of just a few students, which grew later on. A significant improvement happened early on, as one student created a top bar for Nest, which made it easier to grasp the project environment and navigate between tools. Many bug fixes and integration improvements followed, and by the end of a busy summer Nest 1.3 was released under the name of FreeNest. Among other new features FreeNest also had an installer script and user management.

Another significant improvement from the original Nest has been a packaging system. Originally Nest was used from a virtual image, which made development difficult as all the changes had to be moved to a single image file, which acted as a bottleneck between testing and release cycles and was liable to corruption. With packaging Nest could be installed and updated piece by piece without having to rely on virtual machines. This greatly helped both end users for more convenient usage and installation, and development for more flexibility, speed and convenience for changes.

The author of this thesis was involved in FreeNest development mostly during the first SkyNest summer project. Much of the early work involved fixing various bugs and browser compatibility issues. As the author received multiple tasks involving improvements and changes to top bar, it became clear that the top bar had fundamental problems related to code maintenance as even the smallest changes could be hard to make because of the way the code was constructed. As a solution, the author took on the task of writing the top bar again.

While the new version of top bar fixed some of the problems of the old one, and even added a new feature for easy menu editing, in the view of the current state of the project it does not seem enough. The design of the new top bar was flawed in the idea that the purpose was to create a simple navigation menu. As it first became apparent when the editing feature was added, the design was not flexible enough, in addition to having other code-related problems. This leads to the topic of this thesis, which is a kind of third incarnation of the top bar, under a slightly different focus.

2 CONCEPTS

2.1 REST

REST - short for Representational State Transfer – is an architectural style that defines how client and server should communicate. As the name implies, REST deals with representation of resources – resources being any information or service a server wishes to provide. The client does not interact directly with a resource, but rather with its representation, and REST defines constraints for such interactions. A representation contains the actual data and metadata such as media type, control data like cache control and so on. (Fielding, 2000)

The resources that a service wishes to provide are given an ID that can be used to access the resource's REST representation – naturally with the full URL. A resource may have multiple representations, so it can be accessed in different formats and by different clients. Representations should also be accessed using standard methods, which means that at least within a particular REST implementation all resources should support similar methods such as HTTP's GET, POST, DELETE and so on. Another

important REST constraint is statelessness, which in practise means that the server does not store any information on the previous communications with a client and each request has to contain all relevant information. (Tilkov, 2007)

In practise this means - in the context of this thesis – that the client does not refer to server's script files or other resources directly, but rather to an url address which has functions attached to different HTTP methods. For example, sending a GET request to http://example.x/users/userX, the server returns all available data about userX, while sending a POST request to http://example.x/users could add a new user to a data storage.

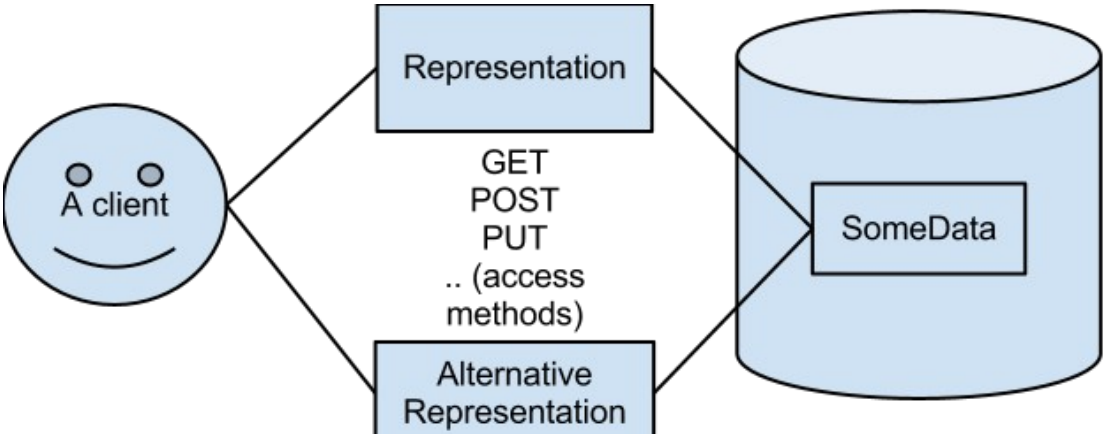

Figure 2: Accessing resources via REST interface

2.2 HTML

HyperText Markup Language is the primary publishing language in WWW. It is meant to be an universally understood language, which is accomplished by an assumption that HTML authors and clients adhere to the same specification. HTML is used to define the structure of data, such as headers, tables and most importantly hypertext links. Originally developed by Tim Berners-Lee at CERN during on early 1990, the most widely used version is currently 4.01(W3C)

While HTML 4.01 is still the currently most used version, the next big thing is HTML5. Unlike HTML 4.01 and earlier versions, HTML5 specification does not originate from W3C, but from a group called Web Hypertext Application Technology Working Group (WHATWG) in response to W3C developing towards XHTML – a HTML specification

that would have strictly followed XML standards and so broken backwards compatibility with older tools and sites, among other faults. Currently HTML5 is being developed by HTML Working Group, with a stable version of specification maintained by W3C. HTML5 is a direct improvement on HTML 4.01, adding semantic elements for defining page structure, support for video and audio and many other powerful features. (Millis, 2011)

2.3 AJAX

AJAX is a way HTML page can communicate with server asynchronously, as the name implies. Normally in order to send or receive data from a server the page has to be reloaded. With the use of JavaScript's XMLHttpRequest – object it is possible to do the same without disruptive page reloads. The object sends a request to server in XML format, which carries various info such as content type and encoding, and any data the client script wishes to send. The server responds to this similarly, content depending on installed web services and scripts. (Mozilla Developer Network)

AJAX is heavily used in the FreeNest web interface, some examples being saving and loading of the navigation menu, checking if the user is administrator and so on. It is especially important because the interface needs to be non-obstructive, and reloading the page for some interface changes would not be feasible. However, if one should look at the current – or future - code, he would hardly find a mention of XMLHttpRequest. This is because the object is only used indirectly via jQuery's jQuery.get(), jQuery.post() and similar shorthand methods. Backbone also uses this method in fetching model contents from server.

2.4 HTML DOM

HTML DOM is an extension of Core DOM, Document Object Model. DOM is a language-neutral interface that describes documents and can be used to update and otherwise interact with them. On the most basic level a DOM structure of a document is a collection of nested nodes such as HTML's , that have contents and attributes. The node tree can be walked to find desired data, and nodes can be added, changed removed. (Quirksmode.org)

HTML DOM adds more specific elements to the Core DOM to describe the contents and how they should be accessed. It also adds restrictions on where certain element types may appear. HTML DOM is used by browsers to display HTML pages, and it is most often manipulated by JavaScript. (Mozilla Developer Network)

2.5 jQuery Core

> *jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development.* ***jQuery is designed to change the way that you write JavaScript.*** *(jQuery.com website)*

jQuery is a commonly used and very powerful JavaScript library. It significantly simplifies common operations such as DOM manipulation and traversing, AJAX and animations. It accomplishes this with a jQuery object, which contains all of jQuery's methods. The object can be used to select a DOM element or elements, and the jQuery methods can then be used to modify or access the element contents. The objects can also be chained:

```
var addButton =
$j(ui.tab).parent().addClass('tabButton').siblings('.ad
dButton').detach();
```

The above sample is from the current top bar's code. It selects a menu tab's parent element, adds 'tabButton' – class to it, then selects all elements with 'addButton' – class that are siblings to the previous selection (meaning they are on the same level), and detaches them from the DOM tree. The detached selection is added to addButton - variable - in this case there can only ever be one button of that type in that location, so not 'var addButtons'.

jQuery is very well suited to creating user interfaces, which is where DOM manipulation is most useful. Another useful facet of jQuery for this purpose are easy to use events, such as .click(), .hover() and .change(), which are called in the context of the invoking element . Also, jQuery handles browser compatibility well, which is often relevant DOM manipulation. (Strahl, 2008)

2.6 MVC with Backbone and Backbone.Marionette

Model View Controller is a common programming architecture where the code is

separated to the namesake parts. Models store data and deal with data manipulation and storage, views are used to display the data, or part of it in the desired manner, and the controller associates user interactions to actions with views and models. The separation of those concepts on a code level enables code that is more flexible and easy to maintain, as they can be relatively independent of each other and thus can be modified separately, and reused. (Osmani, 2012)

Backbone is a JavaScript library that adds a framework for implementing MVC model to JavaScript. It has Models and Collections to store data and the related functions, and Views that can be associated with Models and Collections to display the data. The controller as such does not exist in Backbone's framework, but Views can be considered to fill that purpose also, as they are the part of the code containing user interaction. Backbone also manages attaching client's data to server via REST, which eases saving and loading data. The framework offered by Backbone is just that, a framework, and it is up to the developer to actually implement the model offered. (Backbone.js website)

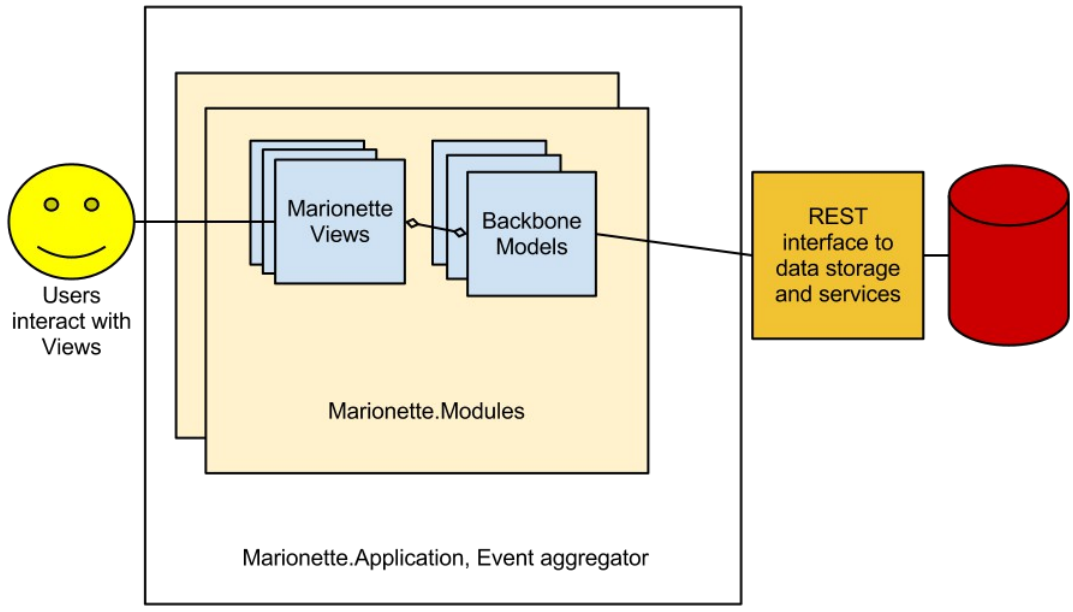

Figure 3: MVC model with Marionette

As Backbone is rather bare bones, it requires much code to work in a more complex application – though the basic usage is very simple. As the author noticed when

testing the libraries, there are many common problems in a Backbone application that have to be solved, and there are also many coders who have created answers for these - just as Backbone is an answer to a common problem, and in fact all libraries are. Marionette adds flesh on Backbone's bones by adding an event aggregator, different types of Views and ways to manage Views and an approach where codes can be separated into Applications and Modules. (Bailey, 2010)

With Backbone.Marionette one can have an MVC application with separation both between Models and Views, but also between different parts of the larger application, Modules.

2.7 Application Lifecycle Management (ALM)

As the name implies, this is a process of managing an application from the early idea to the end of customer support. It covers all aspects of a (software) development project, governance, development and operations as an interconnected process that lasts for the project's lifetime. Governance starts from the idea of the application and consists of all the decision making and the managing of the project as a business. Development is the actual creation and continued maintenance of the application. Operation means the initial and later deployments of the application and its monitoring. (Chappell, 2008)

FreeNest is an ALM solution - as it currently advertises on its web page: *"Application Lifecycle Management for human beings".* It has tools for tasking, testing, source control and generic team collaboration and the tools are combined in such way that it is possible to manage most if not all facets of ALM using it.

3. GOALS OF THE REWORK

This redesign of FreeNEST web interface takes a wider view on the subject. Instead of being another rewrite of top bar it views top bar as a single part of the whole, which includes all functionality that will be added on top of the individual tools.

3.1 REST interface

Most of the communication between server and client should take place via a REST

interface. This way both sides are independent of each other and the implementations can be easily changed as long as the common interface is used. The goal is, naturally, to implement this as fully REST; however this should not be treated as a limitation to prevent implementation of features that contradict REST, as long as it is justified.

3.2 Modularity

Each part of the web interface should be as independent of the others as possible, in order to be updated, removed and added separately. This makes upkeeping of the code easier, as one only needs to know about the particular portion of the code he is working on, instead of having to know if it depends on other parts of the whole.

This is achieved by separating each separate portion of the code in its own logic. For the prototype Marionette.js was chosen to help this, and the separate modules are referred to as applications within Marionette.

Communication between different applications take place by the way of events. A change or action triggers an event, which triggers any calls in applications that care about that particular event. As the services do not directly know about each other, it is significantly easier to maintain them separate of each other.

3.3 Clear file and code structure

The old web interface suffers from somewhat messy code, which is separated into files that care about each other, however there is no clear way to know which files use others. Indeed, there are some files that are not - and could not be - used, because it has not been clear whether they are safe to remove or not. The actual code has DOM manipulation, animations and other visual code alongside actual logic, and there are some dirty solutions without real explanation.

With the rework comes a change to start with a clean slate. All separate files should have a clear purpose, and the actual code ordered into a logical structure. Everything should be as self-documenting as possible, so that future developers do not have to waste time reading irrelevant portions of the code just to change one part.

3.4 Documentation

This ties with the previous chapter – while the work should be self-documenting, it should also be documented. This means code should have comments describing what a function or class does, what kind of arguments it takes and what it returns. Any other non-obvious parts should also be explained, such as any cross-browser compatibility fixes. An actual documenting tool such as JSDoc might also be useful.

There should also be documentation outside of the code, explaining in plain language how the whole interface works, and what the classes do.

4. PLAN FOR NEW INTERFACE ARCHITECTURE

4.1 General idea

The most important part of the interface is implemented using JavaScript, and can be treated as the 'main loop', so to say. The server side scripts are important, but the client side interface can be considered to use those, and their implementation can vary – as long as they support the REST interface.

The 'main loop' itself should be as bare bones as possible and only maintain the event aggregator (or similar) and load/call the modules. The modules themselves should each perform a single function per module as much as reasonably possible.

Much of this uses Marionette as an assumption and example, but it is not necessary to use it. It is simply useful for explanation purposes, as it contains most of the concepts that the new interface should have as classes, and so it reduces the length of examples. This thesis also recommends Marionette be used for the final implementation, as when using Backbone there is much code that has to be created that is already implemented in Marionette. The main argument against using it would be it adding another library dependency, and a possible overhead for a relatively small project.

4.2 Main Application

The main application is the starting point of the interface. It loads all modules and

manages the events or similar methods the modules use to communicate. The main application itself should not contain anything besides the framework for managing the actual features, as it can easily end up bloated otherwise.

An example of main application using Marionette.js:

```
NestApp = new Backbone.Marionette.Application();


//get and append html templates for Marionette to use
$.get('topbarTemplates.html, 0, function() {
      $('#topbar').append();
}


//Add regions, which are Marionette's feature for
managing VIews
NestApp.addRegions({
topRegion: '#topbar',
overlayRegion: '#overlay',
bottomRegion: '#bottom',
..
});
```

This simply creates a Marionette application, uses jQuery to load a template html file and append it to the DOM structure under #topbar – which is a placeholder that tells where the topbar should be placed – and defines some divs as regions for the application. The template file should include templates for Marionette Views to use, or otherwise the basic html structure for all core elements of the interface. Also one should note that any optional modules (addons) may want to add new templates. This could be managed by either providing the template file via REST by returning a single file containing all the required templates from different files, or each module loading their own template. The main templates should cover as much as possible to reduce need for additional ones.

The regions are Marionette's way of managing Views, and even if Marionette is not used it should be useful to identify different regions of page for modules and addons, so that interface element locations are easier to manage. The main identified regions are:

- Top, which is the (default) area for the main navigation menu
- Bottom, which could be used for any smaller and preferably temporary elements. Also if chat feature is implemented, it should primarily reside here
- Overlay, which corresponds to a div that is defined in CSS to cover the rest of the page. Some suggested uses are tutorial/help information and movable elements like a notepad tool.
- Possibly sides, though one should note that these scale less well with different display resolutions etc.

4.3 Modules and add-ons

Modules form the actual meat of the interface. Some modules, primarily main navigation menu, can be considered core parts of the interface, while others are optional addons. Regardless they should function similarly, and the distinction should be a matter of hierarchy – i.e. addons work around core components. Modules in general should be as independent of each other as possible, and any interaction between them should take place via events:

```
NestApp.module('Topbar') {
      Topbar.addInitializer( function() {
            NestApp.vent.trigger('global:newAddon');
            NestApp.vent.on('global:redButton',
selfdestruct());
            ..
      }
      ..
});
```

In the example module called Topbar is created for NestApp main application. The function addInitializer contains everything that is run once the module is started. In this case the module triggers an event on NestApp's event aggregator, and starts listening to redButton. If redButton – event happens, it will trigger selfDestruct() - function of Topbar. By using events modules do not depend on each other and can be removed without breaking other modules, yet they can communicate when needed.

Modules may also need to store data. While it is possible to leave it up to individual modules to take care of their own data storage needs, this may make module installation more inconvenient – it would be best if modules could be installed by simply copying the module folder and enabling the module via a configuration utility. For this purpose the server's REST interface should support creation of tables for modules, so that if a module attempts to POST data via REST and the corresponding tables do not exist they are created. It is very possible that this is not the best way to handle all data storage, but it is a good option to have.

4.4 Addon management

For a system with addon support, there is also a need for convenient add-on management. This can be made remarkably simple with some server side support. There should be a separate folder for addons, and each addon should fulfill certain conditions: they should be contained in a folder with the name of the addon, and each addon should implement an addon interface, such as Marionette.Module.

With these conditions, a simple Python script can read all the addon files and then return them as a single .js. This is important so one can be sure all the relevant files are loaded before their contents get called. There are JavaScript libraries and functions for this same purpose, but that dependency (and additional work) can be avoided with few drawbacks, when the addons are assumed to only work in the Nest context in any case.

It would be also good to have a tool for managing addons, so that it is possible to manage addons – toggle on and off for example - without having to physically move files around. This same tool could also be used to manage addon settings – an addon could provide a set of settings available for configuration, which would automatically be shown in this tool. As the core parts of the web interface should be similar to addons, this same tool doubles as a generic configuration utility.

4.5 MVC model

As there is an assumption that Backbone.js is used to implement the interface, MVC separation should happen naturally. It is important to keep all DOM manipulation – which would be the view – separate from the rest, so the code is easier to read and maintain.

As mentioned, Backbone.js already covers this with its Backbone.Model and Backbone.View classes. Models contain all data manipulation relevant functions, while views are responsible for how the data is actually displayed. Perhaps the most important functionality granted by Backbone.Model is how it can be used to fetch and update data via REST interface. This can greatly simplify all client-server data storage interactions as long as the server supplies proper REST interface – which is part of this thesis.

Views describe how the user interacts and sees the data. Backbone views have a render() - function that is called to display the DOM, and an event support for user interactions such as click, doubleclick etc. A good rule is, if one considers using jQuery for something, the code probably belongs to a view. An example use of Backbone.Model:

```
var Addon = Backbone.Model.extend({
      defaults: {
            name:'test',
            enabled:'false',
            description: 'test'
      },
      url: function() {
            return restAddress + '/' +
this.get('name');
      }
});
```

In the above example a model of an addon (which could be used for an addon control panel, for example) is created. It has some default values, and the url – variable is path that is used to populate this model via REST. The models often belong to a

Collection. To actually display (render) the data to user, the Model or Collection needs to have a View:

```
var AddonsView = Backbone.View.extend({
      tagname: 'div',
      classname: 'addonsViewTest'
      initialize: function() {},
      render: function() {
            $.each(this.collection.models,
function(key, value){
                  console.log(value);
            });
            return this;
      }
});
```

The AddonsView describes how Addons – collection should be rendered. Tagname tells it should be inside a div, classname tells the div's class. In this case it outputs all of the models' data to browser's console, but in practise render should contain code that describes what information is shown and how. It is possible to chain the render()s - which is why the function should return *this* - so that a single model's render describes how that model is rendered, and collection then renders those while creating the container. This way it is possible to separate the visual code of each object neatly, and the code should be independent of what other visual elements do – and even more so of models. Once model and view are created, the model has to be populated and coupled to a view so the user can view the data:

```
var allAddons = new Addons();
allAddons.reset( {{GetAddons()}} ); //or
allAddons.fetch();


var addons_view = new AddonsView({
      collection: allAddons,
      el: $('#addonsList')
});
addons_view.render();
```

Here allAddons is an instance of the Addons – collection. Reset() is used to populate the collection on page load, and this assumes the data is available by time reset() is called. In this case the JavaScript file is returned via a python script that has a GetAddons() function - which in turn gets the addons from database and returns a table. This way when the JavaScript file is loaded, reset() already has all the data as an argument. Alternatively one could use fetch(), which loads the data via the REST url, but if the data is viewable immediately on page load it should also be available right away. As recommended by Backbone.js documentation:

> *"Note that **fetch** should not be used to populate collections on page load — all models needed at load time should already be bootstrapped in to place. **fetch** is intended for lazily-loading models for interfaces that are not needed immediately: for example, documents with collections of notes that may be toggled open and closed."* (Backbone.js website, 2012)

After the model is populated, it is associated with a view, which can then be rendered.

While the above code is a greatly simplified example of a single case, it should show the basic idea behind Backbone.js Models and Views. It is recommended to use these, or another similar solution to when implementing the interface. This is another area where Marionette offers good benefits by offering extended Views for different purposes and the aforementioned Regions and Layouts for further organization. Actual usage of Backbone Views would require similar code in any case, and it makes sense to use an already tried and tested solution.

4.6 Themes

Theme support is another issue that the current instance of the Nest web interface lacks. This should be implemented so that theme configuration and creation of new themes is as simple as possible. A question here is how much a theme should do. It should be taken as granted that a theme does cover fonts, colours and images, but does it also include menu animations, or even how the menus are displayed?

Since the views will be separate, it is possible to have them in different files from the models, and so they are easily replaceable by a theme, and this does make it possible

to alter basically anything the user sees with a theme. A function could, for example, change the names of the views used. Or even simpler, load different view files.

Related to themes, it would be a good idea to have a set of re-usable templates, views and other visual elements – that is, it would be good to create the elements generic enough that they can be re-used. There are naturally libraries for this too, such as jQueryUI, but they should be flexible and light. jQueryUI for example should not be used, as it is not, in fact, very flexible as it depends on its own CSS files and images.

4.7 File structures and interfaces

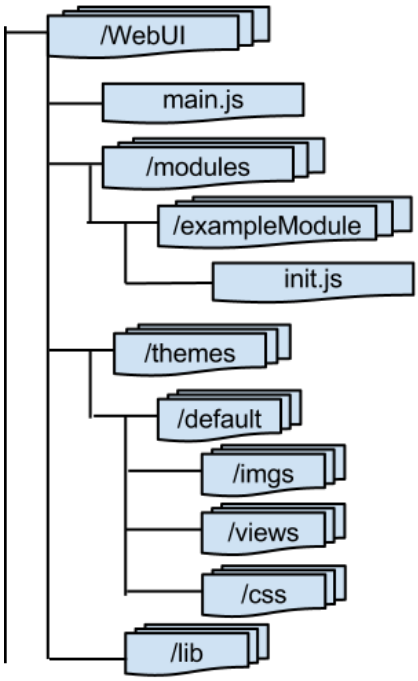

Figure 4: Example file structure

It should be clear where different types of files are located, and what the files contain. Below is an example, though naturally the final structure may not be similar:

That example is rather basic, but it is worth noting few things. All modules should be in their own folders, with a single file (init.js in the example) that is always similarly named so it can be easily found by either a server or client side script, depending how the modules are loaded. Similarly for themes, it should be easier to install themes if all the relevant files are in a single folder. The lib folder of the example is for

any third party libraries, such as jQuery and Backbone – of course it may make more sense to have separate library folders for FreeNest code and for third party libraries, the example is made with the assumption that all of the web interface code is found either in the main.js or in the addons.

The REST interface should be similarly planned, to make clear what services are available. The interface should be as generic as possible, so that any available data can be retrieved easily not only for the current purposes, but any future needs also - which of course simply means the interface should follow REST guidelines. How the exact interfaces are implemented is again left open, but basically whenever the client web interface needs to interact with the FreeNest server it should be via REST if reasonably possible – that is, all direct references to certain files or ports should be avoided. Backbone assumes by default that collections and individual models are stored in certain URLs, but it is very flexible in this regard. The default is a very good place to start with, however.

An important matter to note is that this thesis is written with an assumption that certain libraries are used. It is naturally possible to use different libraries for implementing this, but the model offered by Backbone (and Marionette) seems to fit well with the aims of this thesis, and they are certainly convenient for the sake of examples.

4.8 Libraries

Some of the JavaScript libraries used are very obvious, and basically mandatory choices. j**Query** is already commonly used inside FreeNEST, and not picking it would require rather good reasons. It is a well-known and powerful tool for managing DOM structure, and the other libraries used here have a light dependence on it – they do not require jQuery, but many other libraries assume it is used in their documentation.

Underscore is an equally obvious choice, also being already being used in some FreeNEST solutions. It offers lots of utility tools that simplify common coding structures and add to JavaScript functionalities already available in many common coding languages. It is also not very intrusive, as its functions are stored neatly behind its namesake _ symbol. (Siddhart, 2012)

Backbone is a library that depends on Underscore. This choice too is based on tools already used in FreeNEST. It is a rather lightweight library that adds MVC functionality to JavaScript. It offers Models for storing data, and Views as a template for displaying those models – keeping all DOM manipulation and interface separate from actual logic. There are also Collections for managing and sorting multiple items, and Router for address handling, but on the whole it is a barebones library that offers a framework for cleaner JavaScript code. It is also worth mentioning that it uses REST as a default method of storing objects on server, which is very convenient for this work. (Joreteg, 2010)

Marionette is more of an optional library. This choice comes from realization that there is much boilerplate code that would have to be placed on top of Backbone, which in itself offers only a framework for the actual code. Another reason for choosing this was a desire to have a framework for managing the web interface as a whole application, rather than as a collection of separate functions and calls.

Marionette is also useful for modeling the add-on feature, as one of the keystones of Marionette is a composite structure. This way not only can the default features be implemented as independent applications, but any addons can be managed as such too. (Bailey, 2010)

Here again are choices that stem from pre-existing FreeNEST solutions. Python itself was chosen because of this, as well as **SQLAlchemy**. Beyond that there were choices to make, however the backend could be implemented with many languages and frameworks, as all that matters is that the end product implements the provided REST interface.

Bottle is a web framework for Python. It is used mainly for the routing functionality to implement a REST interface, and to manage returned data types. This could easily be replaced with Flask or many other Python web frameworks, Bottle is again mainly chosen because it is a very lightweight and focused library. Lack of multi-line templates may be a drawback that may necessitate choosing another framework.

SQLAlchemy is simply a commonly used ORM library for Python. A SQLAlchemy plugin for Bottle is also used for a bit simpler session managing.

5 SUGGESTED MODULES

5.1 Navigation menu

This is the most important module of the web interface, though arguably also one that by itself would not need particularly much coding nor planning, and should be the first one to be implemented. This is basically the original FreeNEST Topbar, but the distinction here is that navigation menu only refers to the menu and the relevant visual elements, and excludes all other features.

At its very basics the navigation menu is a collection of links organised under categories. The challenge is that this menu has to be easily editable, look the same on all browsers - though this goes for all of the web interface, have certain animations and the theme should also be easy to change.

The current navigation menu is edited by clicking a button that puts the menu into the edit mode, where the administrator can drag and drop links and categories to desired places, delete those and create new ones. There is no real need to change how this works. What is suggested however is that the edit mode should also open a configuration page with additional customisation options:

- Change theme
- Change fonts, colours and other visuals on top of the theme
- Add more pages to a category. As the navigation menu category tab should always be opened on the category the current page belongs to on default. This association of categories and pages could also be used for other things.

The current version saves the menu structure as an XML file and it may be worth considering the advantages of using JSON or a database instead. The author recommends switching to JSON, as the menu will be manipulated using JavaScript in any case. The menu file was saved as XML originally mainly because there was no editor and XML files were somewhat easier for manual editing compared to JSON. It is probably simpler and cleaner to use JSON in JavaScript in any case. The saving and

loading of the navigation menu structure should be done via REST, so how the server saves the data depends on which is most convenient way, as long as it can send and receive the menus in JSON format – using a database is also a good option.

5.2 Webchat

Webchat is meant to replace the current interface's ShoutBox. The most important features - that are currently lacking - are support for multi-user chat, chat bots (for functionalities such as easily referring to bugs), chat log and more convenient user interface so one can have actual conversations instead of sending single messages.

The proposed way to implement this is to use XMPP (Extensible Messaging and Presence Protocol). It is a very flexible and well-supported protocol with many existing client and server solutions for different platforms. It offers support for all features the webchat needs and more with its great number of extensions and its XML-based nature makes it convenient for web applications.

The server choice is important, as while the protocol itself is very flexible it is in no way guaranteed that a server implements all the extensions. One should also pay attention to the license, as there are commercial or otherwise limited servers which should naturally be ruled out. The author recommends either OpenFire or ejabberd, as both of them should cover the needed features. OpenFire is more user friendly with easily usable web interface that covers most settings, while configuring ejabberd's configuration is based on text files. While this should not heavily weight the choice in either direction, an easy user interface is much preferable if the intention is that the end user group's administrator can also manage users, chat rooms and other features easily. In ejabberd's favour, at the time the author tested the different servers it supported message delivery to all of user's clients at the same time, while OpenFire needed an user-made modification.

There are also many possible choices of library for the client implementation, and it is again recommended that a ready library is used. The library should preferably depend on languages already used in FreeNEST, which basically means that if it has a server-side component it should preferably be in Python. Also any Flash-based clients should be excluded. A good recommended library would be Strophe, written in

JavaScript with optional - though in case of this webchat, quite needed - server-side components in Python.

The suggested chat features are:

- A private chat (between two persons)
- Group chat rooms
- Online user and user status list
- User can be logged in from multiple locations
- Messages delivered to every user's client where user is online – both received and sent preferably.
- Chat history / log
- Default chat rooms for a project
- Chat bot + integration with other tools
- Web based chat management (configuration) – administrators should also be able to disable features such as user rooms etc.

5.3 WikiWord

Wikiword is another feature of the old web interface, this could be re-designed as a module. The basic functionality of this feature is simple: it finds any words that exist on that FreeNEST instance's wiki as topics, and changes those words to links to the relevant topics. While this feature has been problematic due to it working on third party sites (the FreeNEST tools), its idea is still solid and if the new version has a solid configuration tool it is worth implementing. Suggested features:

- Find words written in WikiWord format, replace with links
- Blacklist pages and page elements – areas that should be ignored when searching
- Web based configuration that can:
    - Change the used regexp, both with a wizard or a pool of different options,

and a custom regexp

- Edit the page/element blacklist
- Add options for external link creation

5.4 Other modules

There are also other already existing features that have to be, or should be turned into modules, and possible new features that should be implemented as such. Log in screen could be a module, or it could stay as it is, a separate screen. Since the page has to be reloaded in any case, and the screen is relatively simple, there are not any huge benefits to be gained, except the usage of (possible) themes can be extended to log in screen. The same goes for user configuration and other tools that currently use their own pages in any case. Team mood meter is a feature that is included on the current top bar, so it has to be reimplemented as a module.

A possibly useful feature that has been talked about would be a help and tutorial tool, that could use the overlay region to display tips on how to use the current page. Similarly it could be possible to implement an improvement on WikiWord – feature that could show short descriptions of the topics when moving mouse cursor on them.

6. DOCUMENTATION

This is another important part that is mostly missing from the current version of the web interface. The documentation should be comprehensive and clear so that new developers can easily continue developing instead of wasting time delving through the files and code – and it helps the old developers too, when dealing with older code.

6.1 Comments in code

There are tools for automatically generating documentation for JavaScript, such as JsDoc Toolkit, but it is likely that they will not work well with the different libraries used in the project, and it is better to just comment the code normally. The commenting style used in Backbone.js annotated source is a good example to follow,

with each code block described on a general level – it is not necessary to mention routine variables, but anything that is not immediately obvious or that refers to another function should be described. (backbonejs.org)

While automated document generation using JsDoc or similar tools that rely on code syntax may not be practical, Docco is a viable alternative. It is a document generator that creates a HTML document with syntax highlighted code and the relevant comments separated alongside the code in a different paragraph. It does not rely on comment annotations nor does it try to recognize objects and other code patterns, and instead takes both as is and outputs them in a easy to read format. Docco does support Markdown for advanced comment styling such as emphasis and code block tags, but the user can also just use normal single-line JavaScript comments and get good results. (Bailey, 2011)

While the above concerns JavaScript code, the server side code should be similarly commented. Unlike with JavaScript, it is possible to use automated document generator with Python, and there is a multitude of options to choose from. For sake of consistency it might be a good idea to use Pocco, which is a Python port of Docco with the same features – and limitations. For more heavy-weight document generation a good option would be Sphinx, which is a generator used by Python itself, and many other projects.

6.2 Wiki pages

Besides code comments a good way to document a feature is to create a Wiki page. The page should contain a description of the feature how it works on a more generic level. It should also act as a guide on how to use and configure the feature, what the different options do. There are already existing guidelines on Wiki documentation inside Nest project groups, which will be more up to date, so it is only really worth saying here that the Wiki pages should used in addition to other documentation.

7. SUMMARY

This thesis describes the desired future version of the web interface on a very generic level, and does not go into specifics of architecture or implementation. This is because it is meant to be more of a description of how the current state of the interface could be improved, and not design whole of the architecture. How – and of course, if – the actual next version of the web interface will be implemented depends heavily on the resources available at the time, and how much of the suggested changes are actually needed.

While this thesis on some level stays rather generic, it does lean rather heavily on certain libraries being used. This is because especially Backbone is very useful for most kinds of JavaScript solutions. Marionette offers a more specific model, but it was chosen because the author feels that the same model is a very good one for the redesigned interface. Marionette is not at the moment that commonly used, and is a smaller project, so there is a risk that it will not be updated regularly. Nevertheless, it is good for example's sake even if it is not used otherwise.

An issue that rises from the proposed model of using a single framework to cover all of the web interface means that pretty much everything will have to be rewritten. In the author's opinion this would be needed in any case, as especially the older parts of the current web interface can not be fixed without more or less full rewrite: The old topbar certainly has to be done over again, WikiWord's core logic can be used but it needs configuration and refactoring, and ShoutBox cannot be used in any form. There are other features that will need to be refactored to use the new interface also.

The question that naturally follows is: *is it worth it?* In the authors opinion, yes, in the long run it is worth it. The current web interface is in well functioning state at the moment, and does not really need much resources to upkeep – by and large it works. Any further improvements on the old interface would be built on a rather fragile platform, and it is instead better to spend resources to build a more stable platform before adding new features. It is a lot of work, but in the long run it should save more work. A huge benefit of a modular framework is also that it will be much easier for

third party developers to create modules.

REFERENCES

Backbone.js. n.d. Javascript library. Referred to on September 13, 2012. http://backbonejs.org

Backbone.js, 2012. Backbone annotated source code. Referred to on November 13, 2012. http://backbonejs.org/docs/backbone.html

Bailey, Derick. 2010. Composite Javascript Applications with Backbone and Backbone.Marionette. Referred to on September 13, 2012. http://lostechies.com/derickbailey/2011/12/16/composite-javascript-applications-with-backbone-and-backbone-marionette/

Bailey, Derick. 2011. Annotated Source Code as Documentation with Docco. Referred to on November 13, 2012. http://lostechies.com/derickbailey/2011/12/14/annotated-source-code-as-documentation-with-docco/

Chappell, David. 2008. What is Application Lifecycle Management? Referred to on November 5, 2012. *http://www.davidchappell.com/WhatIsALM--Chappell.pdf*

Fielding, Roy T. 2000. Representational State Transfer (REST). Referred to on September 18, 2012. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Joreteg , Henrik. 2010. Building a single page app with Backbone.js, underscore.js and jQuery. Referred to on September 13, 2012. http://andyet.net/blog/2010/oct/29/building-a-single-page-app-with-backbonejs-undersc/

Mozilla Developer Network, n.d. Ajax: Getting Started. Referred to on November 13, 2012. *https://developer.mozilla.org/en-US/docs/AJAX/Getting_Started*

Mozilla Developer Network, n.d. Javascript Technologies Overview. Referred to on November 13, 2012. https://developer.mozilla.org/en-US/docs/JavaScript_technologies_overview

Osmani, A. Journey Through the JavaScript MVC Jungle, 2012. Referred to on November 13, 2012. *http://coding.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/*

Quirksmode, n.d. W3C DOM – Introduction. Referred to on November 13, 2012. *http://www.quirksmode.org/dom/intro.html*

Rintamäki, M. FreeNest Concept Owner. A Presentation to on FreeNest 9.11.2012.

Mills, C. 2011. A Developer's Introduction to HTML5. Referred to on November 13,

2012. *http://www.developerfusion.com/article/123608/a-developers-introduction-to-html5/*

Siddhart. 2012. Getting Cozy with Underscore.js. Referred to on Septermber 13*, 2012. http://net.tutsplus.com/tutorials/javascript-ajax/getting-cozy-with-underscore-js*

Strahl, R. An Introduction to jQuery, 2008. Referred to on November 13, 2012. *http://www.west-wind.com/presentations/jquery/*

Tilkov, Stefan. 2007. A Brief Introduction to REST. Referred to on September 13, 2012. http://www.infoq.com/articles/rest-introduction

Underscore.js, n.d. Javascript library. Referred to on September 13, 2012. http://underscorejs.org

w3.org. n.d. Introduction to HTML 4. Referred to on November 8, 2012. http://www.w3.org/TR/REC-html40/intro/intro.html

w3.org, 2000. Document Object Model Core. Referred to on November 30, 2012. http://www.w3.org/TR/DOM-Level-2-Core/core.html

w3.org, 2001. URIs, URLs and URNs: Clarifications and Recommendations 1.0. Referred to on November 27, 2012. http://www.w3.org/TR/uri-clarification/

Wikipedia, n.d. Application Programming Interface. Referred to on November 30, 2012. http://en.wikipedia.org/wiki/Application_programming_interface