SmartScape™ DSS

Document Overview

This document is targeted at the developer viewpoint and covers each of the key component areas that fit together to create the SmartScape™ DSS software. This document is roughly ordered from high-level overview, and then in a series of discrete sections, works its way down to cover the mid- and lower-level details. It is, however, assumed that software developers will ultimately have to defer to the actual source code to understand the lowest levels of how the software works. Experienced developers may indeed prefer to skim the first few pages of the high-level software overview and then proceed to dig into the source itself to begin the process of understanding the software, perhaps referring to the detailed sections later as-needed. As such, it is not expected nor is it intended that this document be read sequentially.

Additionally, this document makes no attempt to duplicate documentation that is already available for any of the technologies we rely on. One example would be that the Play! Framework is one of the key components that drive the web server and provide computational infrastructure; it would be impractical to duplicate any of that documentation here as there is already a wealth of detailed documentation and examples already provided on the Play! Framework homesite. We have however tried to provide links to external sources where the reader can find more detailed information about some of the technologies we are using.

Software Overview / Goals

SmartScape™ DSS allows users to build and evaluate a land-use change scenario. The SDSS integrates a variety of open-source tools (ExtJS, OpenLayers, GeoExt, Geoserver, Play! Framework) to build a user-friendly web application. Users use a browser-based client to select different places on the landscape using a simple query tool, and enact transformations on selected land, such as converting row-crop agriculture into grasslands. Then users launch a battery of pre-programmed models on the computation server that calculate economic and environmental outcomes of that land use change. Outcomes are returned to the user's browser to visualize the environmental and economic outcomes of the land-use transformations. Scenarios can be evaluated within seconds, greatly enhancing the stakeholder's decision-making processes.

This software is designed to be expandable and customizable to any region in the world for which data inputs are available, though it is currently functioning for the landscape immediately surrounding Dane County, WI.

High Level Component Overview

Web / Computation Server

The SmartScape™ DSS back-end server manages the web asset hosting as well as providing the computational resources for computing model results. The server infrastructure is provided by the Play! Framework, written in Java, and is a highly scaleable and easily configurable web hosting platform. The PNGJ Java library provides support for fast generation of heatmap images in .png format which are then rendered on the client as OpenLayers layer overlays.

Web Client

The SmartScape™ DSS web client is written primarily in Sencha ExtJS, a Javascript library that provides a suite of windows-like widgets with which one can build a robust interface. The remaining Javascript libraries leveraged are OpenLayers, which provides the map rendering and navigation functionality, and GeoExt, which is a small utility library which makes it easy to use OpenLayers inside of the ExtJS framework.

Google Map Server

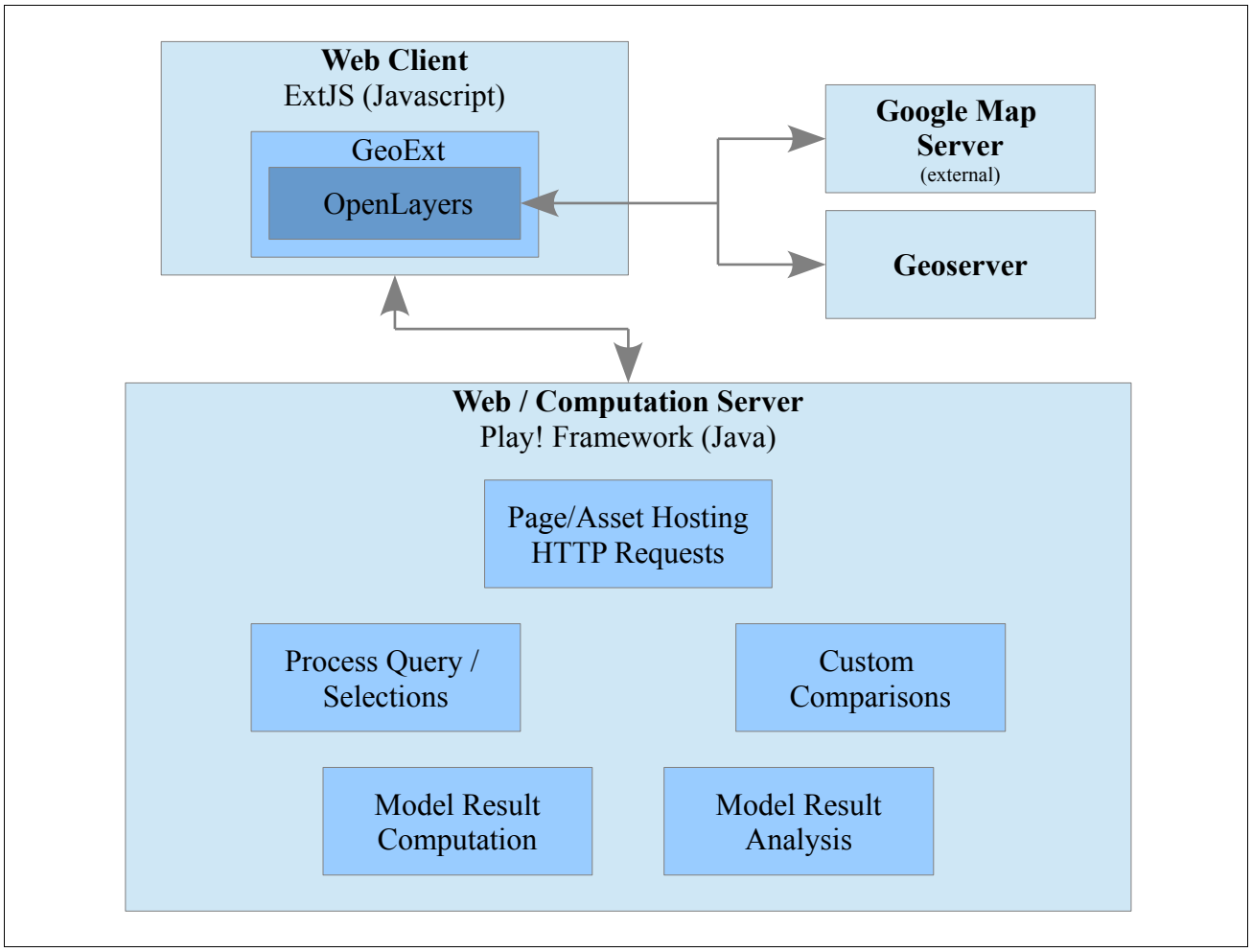
The main map window in the DSS software leverages support built into OpenLayers for Google-based map data visualization. The DSS software allows switching between satellite imagery and hybrid map views provided by Google map servers. These data layers provide the foundation on which custom map visualizations can be displayed by the DSS software.

GeoServer

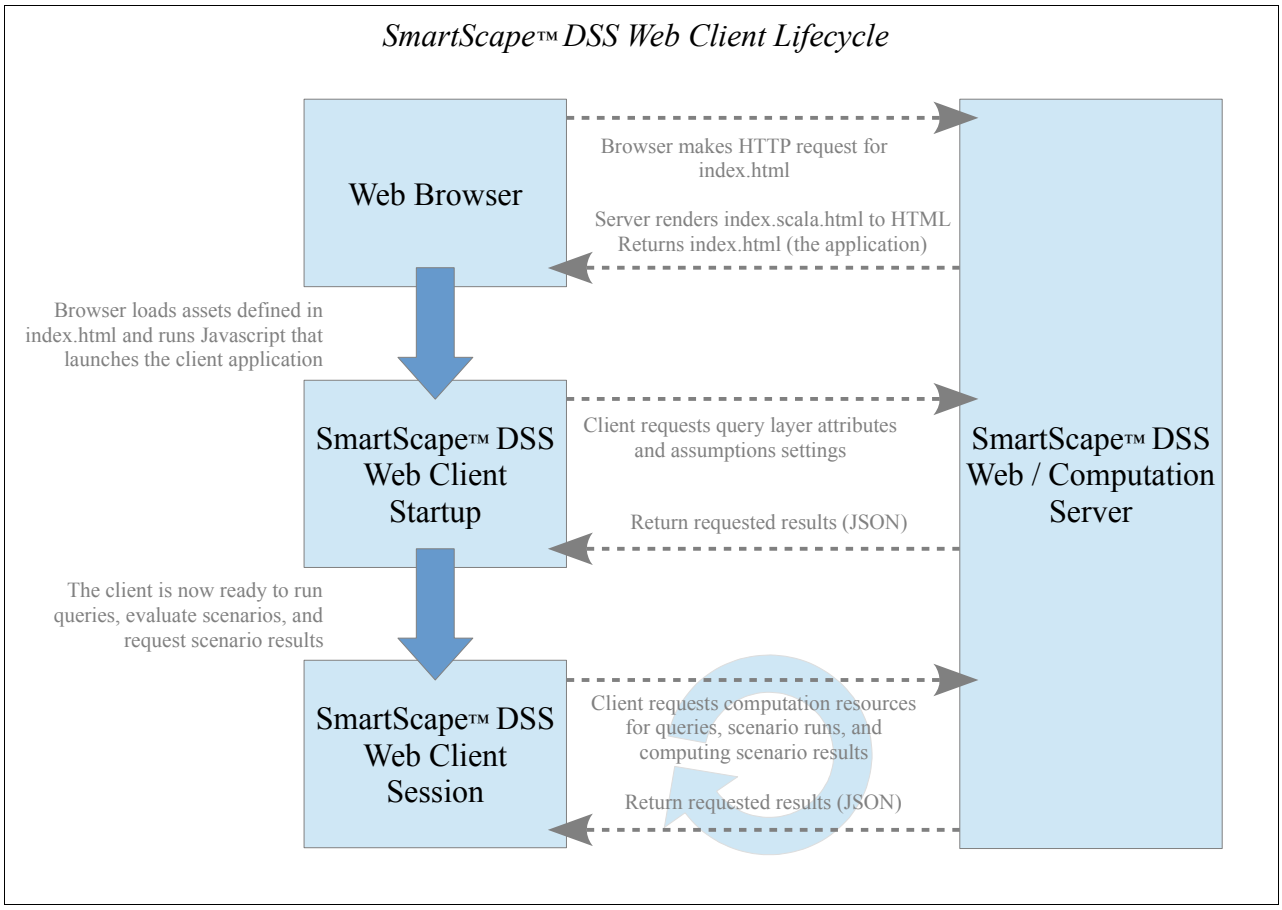
An early iteration of the SmartScape™ DSS allowed for visualizing any of the queryable data layers in their entirety in addition to being able to explore the queryable layers via the selection interface. This visualization feature was eventually disabled because users generally did not feel the feature added anything that couldn't be explored with the selection interface. , the widgets required to support the feature added clutter to the interface, and ma generally was not useful, it required additional widgetsas a result of the sentiment that this was not a valuable feature, which in turn, made it not a feature that we wanted to support and test every time data layers were changed. This functionality is currently hidden in the software and can easily be re-enabled if desired.

A functioning GeoServer may still be needed depending on user requirements. The current DSS does leverage GeoServer for interactive selections of spatial structures, such as watersheds. This data is hosted on the GeoServer in a vector format and the DSS web client interacts with this data set through functionality built into OpenLayers. Thus, the user can intuitively click on a spatial boundary to add watershed features to their selection/query.

High Level Overview - Component Diagram



Server and Web Client Overview - Interactions Diagrams



Application Folder Structure

- (application root) — (root)
  - app — server core implementation
    - analysis — histogram and model result processing
    - controllers — URL entry points from client requests (application.java)
    - fileHandling — file loading and saving routines
    - models — model implementations
    - query — query, selection, and scenario related
    - transformData — data transformation tools
    - views — the HTML page for the application (index.scala.html)
  - conf — server configuration files
    - evolutions — database evolutions (for Client User database table)
  - db — database that tracks optional user accounts
  - layerData — input data for models and queries, model result exports
    - client_***id*** — results folder for a given client, uniquely identified by ***_id***
      - ***{0-9}*** — model result folder, numbered from 0-9, storing a max of 10 runs
  - lib — additional java libraries used
  - logs — server output logs
  - public — web assets
    - help — help files referenced from web client
    - javascripts — client code container
      - app — client application entry point (app.js)
        - view — client application files

SmartScape™ DSS – Detail

Layer Data Requirements

Data layers serve several purposes in the DSS software – they can both inform the query/selection system as well as be used to provide inputs for model calculations. Due to the tightly-knit nature of these data sets with the entire server core, there are some strict requirements for these data that must be met in order for the server to be able to correctly function with a given dataset.

> ***NOTE: We cover the technical requirements here from the point-of-view of the DSS software but there is an assumption that the reader will have familiarity with GIS data preparation tools such as ArcGIS, Qgis, or similar – tools that will be required to prepare new layer datasets that can properly be read by the SmartScape™ core server. Where to acquire suitable datasets is also beyond the current scope of this documentation and will not be covered at this time.***

Layer Anchor Point – All data layers are expected to have been exported from ArcGIS with the same upper-left starting position, otherwise the data cannot line up correctly. The core server does not currently validate a data layer to ensure that it correctly aligns with all other data layers.

Layer Dimensions – All data layers in the SmartScape™ DSS must have identical dimensions (width and height, in rasters or pixels). This is the single most important requirement as a mixture of input dimensions will likely cause memory exception errors. The models are currently coded to use the CDL (crop data layer) as the dimension reference so any data layers should match your CDL dimensions. When coupled with the anchor point, the dimensions can then be thought of as specifying a rectangular raster that spatially overlays a given geographic area. Where the overlay is spatially located is controlled by the particular projection, discussed next.

Layer Projection – The core server is projection agnostic so theoretically any projection system could be used without causing server issues. However, the projection mode is important for the client since the dataset can be used to visualize results. The client is currently coded to use EPSG:3857 since we want our data layers to line up correctly with Google maps (and 3857 is the projection mode used by Google). It is expected that layer data has been reprojected to the same projection system using ArcGIS or similar prior to exporting.

Data Resolution – All data layers in the SmartScape™ DSS have been reprojected to 30 meter resolution. Preparing your layer data with the same 30 meter resolution will be the easiest to get working however there are only a very small number of places in the code where data resolution is explicitly referenced; changing to a different resolution would not require much work. There is no support in the software for working with mixed resolution data and it is generally unlikely that the core server will function in a predictable way as-is with a mixture of data resolutions.

No-Data Value – ArcGIS ASC (ascii) files export a no-data value in the header to denote raster cells that do not have a defined value. In general, the approach of having an arbitrary numerical value represent an undefined value is lacking – model and analysis code must adequately check for this class of input raster cells lest these values propagate into output raster cells. The value -9999 is currently universally used in the software to represent a no-data value. This is treated as a universal rule regardless of which value is specified as the no-data value in a particular ASC header. Also note that even if layer data files are prepared such that there are no no-data cells, it is still important to keep this constraint in mind – without any changes to the core server, any input cell that happens to contain a -9999 will incorrectly be treated as if there is no data in that cell. Similarly, if a model happens to compute a -9999 output raster cell as a result, that too will be incorrectly treated as if it's a no-data cell.

Raster Cell Datatype – The DSS software can work with a variety of raster cell datatypes. The primary types supported are integer/indexed and floating point cells. Internally, the core server raster query system has a few optimizations it can employ given integer/indexed data types. These are generally transparent to the person preparing the data layers but the details become important when the core server attempts to load this data. The following section on Query / Selection System details the points that must be considered.

Layer File Format – The core server preferentially loads data layers in a proprietary binary DSS format. There is currently no exporter tool for ArcGIS for the DSS file format, however, the core server will automatically create a binary DSS version of a layer data file from an ArcGIS ASC (ascii) file. Thus, data layers should be exported from ArcGIS in the ASC (ascii) format for use with the DSS software. The core server will only load the ASC layer file a single time and then the core will export a binary DSS file for subsequent server starts. For this reason, the first server start may take longer than normal. Subsequent restarts will typically be much faster since the binary DSS files load faster than their counterpart ASC files.

Following all of the above requirements, all layer data files in the DSS with our default Dane County, WI data exported in ASC format will appear as follows:

```
ncols             6150
nrows             4557
xllcorner         -10062652.65061
yllcorner         5249032.6922889
cellsize          30
NODATA_value      -9999
```

DSS Detail - Query / Selection System

The web client software has generic support for many types of queries built in. The client software can easily query new datasets on the server core as long as it uses one of these preset query types.

> ***NOTE: We will not cover creating new query types, either on the server core or the client software, however using the existing query features as a how-to template should provide more than enough information for new types to be created.***

The primary types are indexed and continuous. Indexed will be covered first since there are more options to consider. We'll show how the CDL (crop data layer) is set up as a queryable dataset on both the client and the server. Let's start with the server.

The server is instructed to load the cdl_2012 layer in the cacheLayers function in Global.java. This function will automatically be called by the framework when the server starts up – having layers preloaded in memory vs. loaded on demand improves performance at the expense of memory:

```
// In Global.java
private void cacheLayers()  {

        // This dataset is integer and indexed, so we load it as an integer layer.
        // This constructor sets the indexing type to PreShifted, meaning that the data is
        //      preprocessed for fast comparisons against multiple values. This index query type
        //      is limited to indexes that are less than a value of 32.
        Layer_Base layer = new Layer_Integer("cdl_2012");

        // Load the crop data layer from disk. We prefer the DSS format but if that doesn't
        //      exist, we look for it in ASC format. To speed up the server start in the future
        //      we'll write out a binary version of the ASC.
        // Do any additional work to prepare this  dataset for the query system
        //      E.g., load a .key file if we have one (details later in this section)
        layer.init();

        // TODO: load any other data layers that should be queryable
}
```

The core server looks for a key definition in plain text format when it loads a given indexed layer. This definition file must match the base file name but with a .KEY file extension. The .KEY file for the cdl_2012 data layer is presented here:

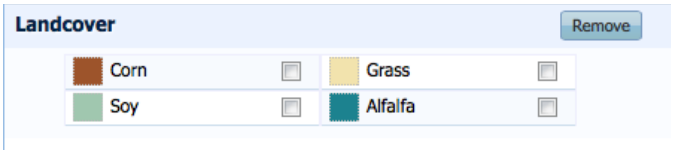
```
;-------------------------------------------------------------------------------
; cdl_2012.key --- Rotation Index, Display String, Client Display Color (in Hex)
; If no color is present, the given key is not ever sent to the client but
;       could still be queried on the java side, e.g., in a model
;-------------------------------------------------------------------------------

; crop types to be sent to the client as part of user transform to a new landcover
;-------------------------------------------------------------------------------
1,Corn,#9c551f
6,Grass,#f2e4a5
16,Soy,#a1c7ac
17,Alfalfa,#218291

; crop types available only for server/model querying of per-cell landcover for models.
;-------------------------------------------------------------------------------
2,Grains
3,Veggies
4,Tree_Crop
7,Woodland
8,Wetland
9,Water
10,Suburban
11,Urban
12,Barren
15,Other_Crop
18,Corn_Grain
19,Soy_Grain
```

This definition accomplishes several things:

- It is the 'glue' that allows the data for that layer to be queried on the server and in model code by name instead of by value. This concept can be illustrated like this:

```
Layer_Integer cdl = (Layer_Integer)Layer_Base.getLayer("cdl_2012");

// Ask the crop data layer to get the corn mask value
int cornMask = cdl.convertStringsToMask("corn");

if ((someRasterCell & cornMask) > 0) {
        // given raster cell contents match the index value for corn
}
```

- It defines which items should be sent to the client to populate the list of checkbox options for that data layer. This is controlled by specifying a web (hex) color for those items to be sent to the client. Thus the preceding definition specifies that the four crop types with web colors should appear on the client interface but the other crop types can still be queried from models running in the core server. Here's how the client options look based on the definition:



Adding additional items to the client is trivially achieved by specifying web colors for the other options that need to be sent to the client.

However, we have not actually created the client query panel widget. That can be accomplished as follows:

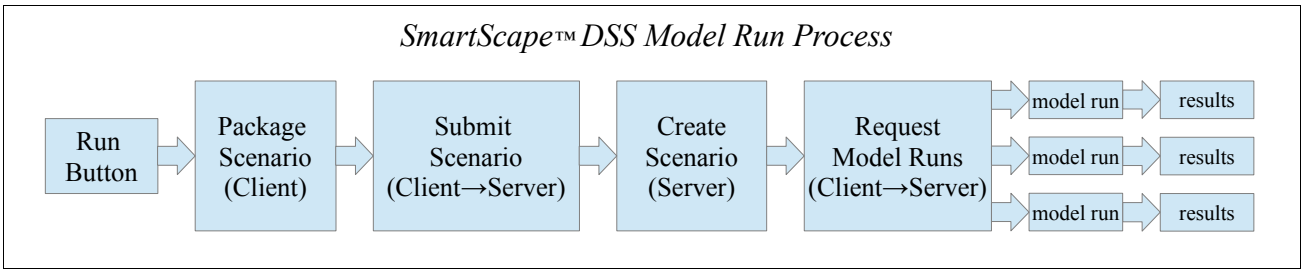
```
// In MainViewport.js
addMapLayers: function(map) {

        // create the query widget. We have indexed data, so create a panel type that matches.
        // The queryTable is the name of a loaded ASC/DSS layer file that exists on the server
        var panelCDL = Ext.create('MyApp.view.LayerPanel_Indexed', {
                title: 'Landcover',
                DSS_Description: 'Match by landcover, example: select rowcrops like corn or soy',
                DSS_QueryTable: 'cdl_2012'
        });

        // Add the created widget to the list of displayed query items
        var dssLeftPanel = Ext.getCmp('DSS_LeftPanel');
        dssLeftPanel.add(panelCDL);

        // Not all panel types are queryable so add this item to the list of queryable panels
        DSS_queryableLayers.push(panelCDL);

        // TODO: add any other widgets
}
```

To clarify, in order for the above client code snippet to correctly work, we expect that a layerData file named cdl_2012.dss (or cdl_2012.asc) was loaded on the core server. It also expects that an index key file, named cdl_2012.key was loaded by the core server (automatically handled by the layer data loading code for indexed files).

If all these requirements are met, you should have a correctly configured query panel on the client that can automatically communicate with the core server.

DSS Models - Overview

A high-level overview of the processing that happens on the client and server upon clicking the 'Run Model' button in the web client is as follows:



In detail, the scenario information is packaged up on the client in JSON format for transmission to the server. This definition contains the entire details for each of the one or more transformation steps in the scenario, data that will primarily include the query/selection definition that defines which crop raster cells should be modified for that particular transformation step, as well as, information about what crop type the selected raster cells should be changed to. The scenario data package also includes all of the assumptions that exist on the client; instead of tracking which assumptions have been changed, if any, the client software currently sends the entire set in bulk. This somewhat streamlines that handling of the set on the server, albeit at the expense of transmitting a much larger data set.

The server handles the request by doing a number of steps. The primary step is that it creates a copy of the CDL (crop data layer) and then runs each requested transformation step, in order, such that we modify the CDL copy to create the new landcover data set the user would like to evaluate. This dataset and a few other key pieces of information are temporarily held in memory on the server, with the entire set being accessible by a unique key that the server has generated and associated with this data. This key is then sent back to the client such that it can request model runs using that unique scenario ID, which will happen next.

If the client receives the scenario ID with a success message from the server, we proceed to the next step which is that the client makes several (nearly) simultaneous AJAX requests to the server, requesting that each model is to be run. The request contains two key pieces of information which is the scenario ID for which we want results and then an identifier which specifies to the server which model is to be evaluated. Because a model may be part of a “cluster”, we request the cluster to be evaluated. Additional details about model clustering will be covered in model sections to follow.

Since the DSS server is multi-threaded and able to handle multiple simultaneous requests, it allocates the required model instances, fetches the correct scenario by ID, and then executes each model instance with the correct scenario information, thereby generating results for the requested models in parallel.

When the model finishes computing its results, it returns a dataset corresponding to the model results to the core server. The core server takes each result and routes them through a suite of analysis tools to generate results that will then be returned to the client for display.

Lastly, the core server passes each result to a spooling process which accepts the model result datasets and saves the results to disk to for future access. This allows for fast comparisons to other already-computed results, as well as, for fast generation of heatmap images; we get very fast response times for the users at the expense of using more disk space.

DSS Models – Server Details

From the point of view of the SmartScape™ DSS software, a model is a Java class with a single, common entry point and a common exit point, which is that the model class returns a Java List of ModelResults. Typically there will be a *single* ModelResult returned, thus maintaining a clean one-to-one association between the class, the model, and the result. On the other hand, the DSS software has related models that rely on identical intermediate computations that may be computationally expensive. One can perhaps imagine several ways a system might be engineered to accommodate this, however, we opted to keep the server core system as simple as possible which ultimately means that the core does not provide any infrastructure for computing and sharing intermediate results. Thus, intermediate computations can currently only be trivially shared *inside* of a model class. This approach led to the concept of a model internally computing as many related results as necessary and returning them as a set for additional handling by the core server. For this reason, it's safest to think of a model as potentially being a "model cluster" and it is model clusters for which the client will request evaluation.

NOTE: an example of this approach can be found in: `app/models/Model_PollinatorPestSuppression.java`

The entry point definition of a model looks like this:

```
public List<ModelResult> run(Scenario scenario)
```

where the Scenario class contains four key things as inputs to a model:

```
public GlobalAssumptions mAssumptions;      // from the client, e.g. custom crop price
public Selection mSelection;                // which pixels were transformed
private JsonNode mConfiguration;            // the transformation steps
public int[][] mNewRotation;                // the resulting transformed crop data layer
```

In more detail, we have a:

- copy of the GlobalAssumptions, which comes from the client. This will contains all parameters regardless of whether they were changed on the client. Models should only fetch assumptions from the client-sent set.
- selection, which are the specific pixels in the crop raster that have been changed. This can be used to restrict intermediate calculations or final analysis to only the pixels in the raster that have been modified.
- configuration object, which contains a list of one or more Transformations encoded as JSON.
- NewRotation data raster, which is the input CDL crop layer with the transformations requested by the client applied. In other words, it is the new landscape with which we'd like to evaluate our set of models.

The exit point of the model is returning one or more ModelResults to the server core. A ModelResult stores key aspects of the model results that are needed by the core server to do additional work on the results.

```
// Model result Class member variables
public String mName;                // unique name
public String mDestinationFolder;   // save routing
public float [][] mRasterData;      // the raster generated by the model
public int mWidth, mHeight;         // dimensions
```

The first step is to pass the ModelResult object to an analysis suite which manages basic statistical analysis. Additionally, the SmartScape™ server currently writes these rasters to disk to facilitate the comparison utilities the tool offers. These results are stored in the layerData folder under a client folder that is named uniquely for each unique client. Additionally, a numbered sub-folder is created, counting from 0-9, which allows us to save up to 10 simulation runs for future compairson.

DSS Models - Custom Server Model Example

What the model computes is clearly highly dependent on the specific model, however there is a lot of flexibility in “how” computation can be done given the server infrastructure. For example, models clearly may need other data rasters to process their computations but it is up to the model code itself to request these other data sets. A trivial example of how some key model programming concepts are accomplished is presented here in a simple example model which accepts a Scenario, loads some additional data layers, and then computes a final result to be passed back to the core server.

```
//---------------------------------------------------------------------------------------------
// Custom model example
//
// An arbitrary model to showcase a few basic concepts.
//
// Our arbitrary model works like this:
//
// Detect a pixel which has been transformed from its default landcover type and assign that model
//      output pixel the distance to the nearest river.
//
// If a given pixel has NOT been transformed to a new landcover type, we'll inspect the original
//      landcover type to see if it's a cell containing corn. If it is, we assign a value of zero to
//      the resulting model output pixel.
//
// Any OTHER model output pixel will acquire the result of an arbitrary formula – here, the
//      product of the current cell x and y.
//
//---------------------------------------------------------------------------------------------
public class Model_TestModel
{
        public List<ModelResult> run(Scenario scenario) {

                // Extract some common values from the Scenario
                int[][] transformedRotationData = scenario.mNewRotation;
                int width = scenario.getWidth();
                int height = scenario.getHeight();

                // Request other data layers as an example
                Layer_Integer cdl = (Layer_Integer)Layer_Base.getLayer("cdl_2012");
                int [][] initialRotationData = cdl.getIntData();
                float[][] distanceToRivers = Layer_Base.getLayer("rivers").getFloatData();

                // Ask the crop data layer to get the corn mask value
                int cornMask = cdl.convertStringsToMask("corn");

                // Our model will compute a new raster, so allocate memory for this raster
                float[][] testData = new float[height][width];

                // process all cells in the new raster
                for (int y = 0; y < height; y++) {
                        for (int x = 0; x < width; x++) {
                                // Arbitrarily looking for pixels which do not match and thus have been
                                //      transformed.
                                if (transformRotationData[y][x] != initialRotationData[y][x]) {
                                        // Arbitrarily return the distance to the nearest river at this
                                        //      modified pixel
                                        testData[y][x]  = distanceToRivers[y][x];
                                }
                                else if ((initialRotationData[y][x] & cornMask) > 0) {
                                        testData[y][x] = 0;
                                }
                                else {
                                        // Compute some other arbitrary result
                                        testData[y][x] = height * width;
                                }
                        }
                }

                // Send the results back to the server core for further processing (analysis)
                List<ModelResult> results = new ArrayList<ModelResult>();
                // Note: “test” is the used to refer to the results for later purposes, such as
                //      comparisons. Specifically, a file called “test.dss” will be written that
                //      contains the entire model results.
                results.add(new ModelResult("test", scenario.mOutputDir, testData, width, height));
                return results;
        }
}
```

DSS Models – Client Details

> ***Note: It is assumed that the reader understands the high-level flow of how the client and server communicate to complete the evaluation of a scenario are return results to the client. Please see the section entitled 'Model Overview' for those details.***

Future versions of the DSS would like to streamline the ease of adding new models and making them available from the client. In the meantime, the current version of the software has a few locations where “hard” links have been embedded that are required as part of the communication process of which models should be run by the server and how to funnel the associated results into the correct portion of the client reporting section. The reporting section is the easiest to visualize since one can refer to the reporting panel in the web client to see which models are available. We'll work backwards and show how to set up a new set of widgets to accommodate new model results. However, until we later show how to physically control running the model, these widgets will not be populated with any results.

Client Requests Model Run

The 'Run Model' button is defined in the Scenario layout:

```
<application root>/public/javascripts/app/view/Scenario_Layout.js
```

and while a handler for this button manages the core process of asking the server to physically run models to compute results when clicked, this class also manages the larger job of editing the scenario that the user would like to run.

DSS Client Details

Client Instantiation

When a browser requests index.html from the SmartScape™ DSS server, the browser renders the associated view contained at:

```
<application root>/app/views/index.scala.html
```

to HTML. This is returned as the result of the browser request.

This HTML file requests the ExtJS and OpenLayers javascript libraries to be imported since they are required by the web client. The HTML file also loads our entry point javascript file which will then trigger the creation of our application. This file exists at:

```
<application root>/public/javascripts/app/app.js
```

Specifically, this definition contained in app.js:

```
Ext.application({
    views: [
        'MainViewport'
    ],
    autoCreateViewport: true
});
```

is telling the ExtJS framework to automatically instantiate our viewport class, which is defined in:

```
<application root>/public/javascripts/app/view/MainViewport.js
```

Inspecting the contents of MainViewport.js, we can see a definition near the top of the file that specifies several other of our custom ExtJS 'classes' that a required for that portion of the layout. Thus the following requirement block will cause the listed files to also be loaded and defined automatically by the ExtJS framework, making them ready for use in our application:

```
requires: [
        …
        'MyApp.view.LogoPanel',
        'MyApp.view.ViewSelectToolbar',
        'MyApp.view.Scenario_Layout',
        'MyApp.view.Report_MasterLayout'
],
```

Note that this is *only* ensuring the required definitions are ready for use, it does not actually integrate any of them into a layout. Placing these items into a layout is done via additional definitions inside of an initComponent function, which will automatically be called by ExtJS framework.

A short snippet is presented here, and in it, we'll add items to our layout. Note that this is done by using a widget alias for the xtype instead of an explicit file or 'class' name:

```
initComponent: function() {
        …
        dockedItems: [{
                xtype: 'view_select_toolbar' // this component is docked top left
        },{
                xtype: 'scenario_layout' // this component is docked to the bottom left
        }]
```

Fully describing the layout in this fashion would be tedious so to facilitate understanding the layout hierachy, we'll next cover the highest level layout and subsections in a layout diagram format.

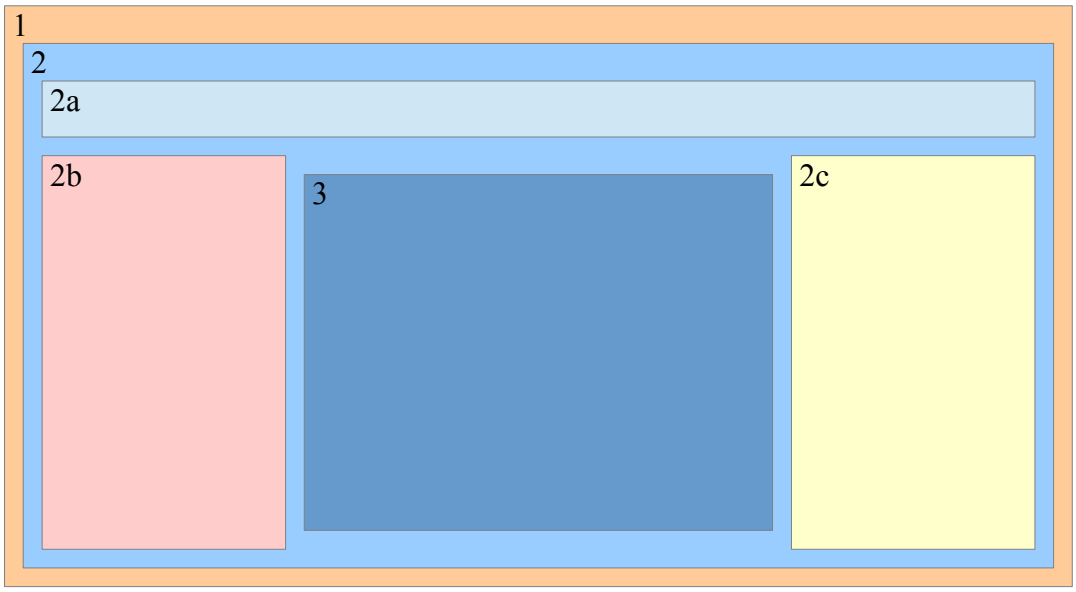
DSS Client Layout Diagrams

Two general approaches are used to create the overall layout: docking and child elements. Both could be considered a form of nesting components but the result achieved differs. We generally use docked components to enforce the core layout and then child elements to fill in the remaining space reserved for/by the parent container. In other words, the application layout structure is achieved almost exclusively through docked components.

Client Layout - High Level

The high-level application structure is defined as follows:

High-Level Client Panel Nesting



1 – Browser Viewport
  2 – ExtJS Viewport Panel, fills the Browser Viewport (1)
    2a – DSS Logo Panel, fixed height, docked to the **top** of the ExtJS Viewport Panel (2)
    2b – DSS Query/Scenario Panel, fixed width, docked to the **left** in the ExtJS Viewport (2)
    2c – DSS Report Panel, fixed width, docked to the **right** in the ExtJS Viewport (2)
      3 – DSS Map Panel, a child element of the ExtJS Viewport (2), using any remaining viewport space

This high-level layout is defined in:

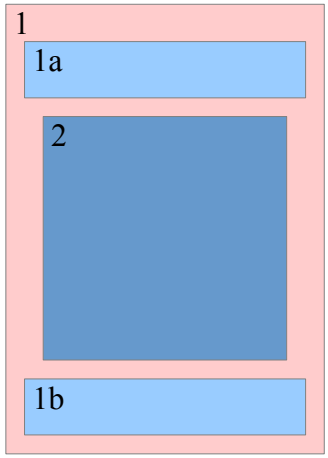
```
<application root>/public/javascripts/app/view/MainViewport.js
```

and provides a recognizable scaffolding upon which the rest of the application is laid out.

The DSS Query/Scenario Panel (2b) and the DSS Report Panel (2c) each have additional items defined to complete the layout for the entire DSS application. The layout diagrams for these sections are covered next.

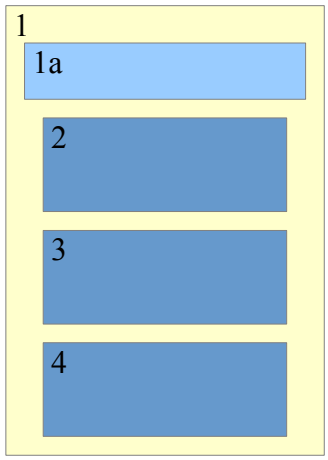
DSS Client Layout – Section Detail

DSS Query/Scenario Nesting



1 – DSS Query/Scenario Container
- 1a – DSS Query Toolbar, docked to the **top** of the Query/Scenario Container (1)
- 1b – DSS Scenario Panel, fixed height, docked to the **bottom** of the Query/Scenario Container (1)
  - 2 – DSS Query Panel, fills remaining height of (1), scrolls to accommodate one-to-many queries

DSS Report Panel Nesting



1 – DSS Report Container
- 1a – DSS Custom Scenario Comparison Toolbar, docked to the **top** of the Report Container (1)
  - 2 – DSS Spider/Radar Chart, fixed height, child element of DSS Report Container (1)
  - 3 – DSS Report Detail, fixed height, child element of DSS Report Container (1)
  - 2 – DSS Save Report, fixed height, child element of DSS Report Container (1)

DSS Software Installation – Required Software

Java 1.8.x – https://www.oracle.com/java/index.html

The server code requires the Java Run-Time Environment, Java version 1.7 or newer is required, with Java 1.8.*x* highly recommended. As of this writing, we are using version 1.8.0_31.

> ***Note: Java 1.6 was used at the start of the DSS the project so if an installation site was restricted to using Java 6 for some unusual reason, a few sections of code relying on more modern versions of Java would have to be rewritten to conform to older syntax (e.g., switch statements using strings would have to be rewritten to use if-else logic).***

Play 2.2.1 – https://www.playframework.com

The Play! Framework provides the core structure for the web hosting needs of the application. Version 2.2.1 is currently in use though newer versions of Play! could likely be used with little issue.

Additional Java Libraries - There are currently 4 additional Java libraries that are required by the DSS software though they should exist in the project explicitly or will be downloaded automatically by the framework. These are detailed below.

Two of the libraries are maintained locally in the Lib folder and only need to be manually downloaded if you have reasons to need a new version (e.g., security updates, performance improvements, etc):

```
<application root>/lib
```

These are:

- PngJ, which is a Java library that offers quick PNG image encoding
- H2, which is a simple open-source database that integrates easily with the Play! Framework

Two libraries are specified in the build.sbt file:

```
<application root>/build.sbt
```

and will be automatically downloaded by the Play! Framework if they cannot be found locally. These are:

- commons-io (v2.4), which is an Apache Java library that offers enhanced file handling tools.
- commons-email (v1.3.3), which is an Apache Java library that makes it easy to send email.

GeoServer 2.4 – http://geoserver.org

A GeoServer installation is required for handling spatial queries, such as selections based on watershed boundaries. It would also be required to visualize individual raster data layers though this feature is currently disabled in the current DSS software so additional work would be needed for the client software to take advantage of this feature.

As of this writing, more recent versions are available. We are using version 2.4 but have used newer versions (including 2.6) for other projects that leverage the GeoServer in nearly identical ways.

DSS Software Installation - Layer Data Setup

Layer data will ultimately need to be prepared for your particular locale – aside from general format requirements presented in other parts of the document – it is beyond the scope of this document to provide detailed information on where exactly appropriate data can be sourced. Step-by-step data conversion in ArcGIS or similar tool is also not covered. Lastly, the startup data layer files are not available in the GIT repo due to the large size of the datasets and the fact that these data do not change so revision tracking is not a requirement for us.

That said, it would be hard to evaluate the software and experiment with making changes to this software without suitable starting data. For this reason, we are making basic startup layer data files for the area surrounding Dane County, Wisconsin available for download.

The uncompressed data size of the input .ASC files is approximately 3.13 GB and is available for download as a single compressed file (approximately 260MB) from the Developer Link on the SmartScape DSS software.

The contents of this archive should be unzipped to:

```
<application root>/layerData/
```

Note: Upon startup, the DSS server will look for DSS versions of each of the layer data input files. For the first server startup, it will not find any so will fallback to ASC loading mode. Each layer data file will them be converted to DSS format for faster server startups in the future. If desired, the input ASC files may be removed at that point since the server will no longer attempt to load them.

DSS Software Installation - Running a Local Server

For development purposes, it's typically easiest to write the code locally and run a local server (e.g., on a desktop computer - I use a 16GB MacBook Pro with no issues). The primary advantage is that once your Play! DSS server is started and running in development mode, any code or project settings changes will be automatically detected by Play! Thus, a simple broswer refresh will cause Play! to recompile your Java changes and then restart the server without any effort on your part. The alternative is to write the code locally, update a server separate from your machine, rebuild/stage the project from the Play! command-line, and then restart the production server.

In order for a development server to be able to run locally on your development machine, you will need the following:

- at least 8 GB of free memory, ideally more.
- Java installed and correctly referenced on your path
- Play! framework downloaded and correctly referenced on your path
- sample LayerData downloaded and decompressed to <application root>/layerData

Once those steps are done:

- open a command line terminal
- change your current directory to <application root>
- type 'play run' on the command line

> ***NOTE: you will have to configure Java, allowing it to allocate more memory if the server errors out during startup. Please see the documentation for your Java installation on how to correctly do this for your operating system. A good starting point is at least 8GB of memory for a DSS server loading the sample layerData we have provided.***

You should now see something similar to this in your terminal window if your installation is ready for use:

```
[info] Loading project definition from /Users/.../Projects/DSS_Server/project
[info] Set current project to Smartscape DSS (in build file:/Users/.../Projects/DSS_Server/)

--- (Running the application from SBT, auto-reloading is enabled) ---
[info] play - Listening for HTTP on /0:0:0:0:0:0:0:0:9000

(Server started, use Ctrl+D to stop and go back to the console...)
```

We can see that the DSS web server is now listening locally on port 9000 for connections. The DSS server is not truly running yet so next, open your web browser and type the following URL to go to your local instance of the web server:

```
localhost:9000
```

Returning to your command line terminal, the server should start emitting logging information related to the startup process, similar to this:

```
[info] application -  ... Server is going to load all layers ...
[info] application - +-----------------------------------------------------+
[info] application - | Binary Read: cdl_2012
[info] application - +-----------------------------------------------------+
[info] application -   Real Path: /Users/.../Projects/DSS_Server/layerData/cdl_2012.dss
[info] application -   Reading header...
[info] application -   - Binary file version: 1
[info] application -   - Width: 6150  Height: 4557
[info] application -   Allocating INT work array
[info] application -   Value range is: 0 to 262144
[info] application -   Attempting to read color and name key file.
[info] application -
[info] application - +-----------------------------------------------------+
[info] application - | Binary Read: slope
[info] application - +-----------------------------------------------------+
[info] application -   Real Path: /Users/.../Projects/DSS_Server/layerData/slope.dss
[info] application -   Reading header...
[info] application -   - Binary file version: 1
[info] application -   - Width: 6150  Height: 4557
[info] application -   Allocating FLOAT work array
[info] application -   Value range is: 0.0 to 70.0
[info] application -   Num Cells with NO_DATA: 0
[info] application -   % Cells with NO_DATA: %0.0
[info] application -
```

NOTE: the first time you run a DSS server, there will likely not be any default scenario computed. This default is the baseline from which any other scenario change will be computed. For this reason, the first time the server is started, the default scenarios will be computed and written to disk for faster comparisons later. There are some differences with this behaviour with production vs. development servers.

<u>Production</u> – production servers always recompute the default model results when they startup. This is done because our developers do not have access to the production servers, and since we wanted the production update process to be as simple as possible for the server administrators, the server startup process always recomputes the default results to ensure that model or data changes are always correctly represented in the default scenario results.

<u>Development</u> – development servers only recompute the default model results if the default folder does not exist. To simplify the server code, this process only checks for the existence of the containing folder, not the individual model results files – deleting individual default model results files is not enough to cause them to be recalculated.

One alternative to prevent this scenario is to modify a server setting in Global.java. Keep in mind that changing this setting will cause development servers to start much slower since they will have to recompute the default scenario results every time the server restarts.

```
private static final boolean FORCE_COMPUTE_DEFAULT_DATA = false;
```

NOTE: the DSS server can be set up to NOT load the default scenario comparison data sets. The production server could be started in this mode, however, it was mostly intended for developers that want the server to start up as quickly as possible. The only downside of using this mode is that scenario comparisons will have to read this data from disk as needed, which will greatly slow down the comparisons in practically all cases, even when using solid-state drives.

```
private static final boolean LOAD_DEFAULT_DATA = false;
```

DSS Software Installation - Production Server

Given the wide variety of server hardware and software used in the real-world, even before considering virtual machine environments and the reality that an installation may need to retooled to work with whatever installations are available, we are opting to not provide detailed hardware recommendations in this documentation. We can however provide some basic insight into what seems to work well for us.

> ***Note: many of these details will similarly impact the performance of a local development server (e.g., a developer DSS server running on a desktop or laptop).***

Server Memory – The DSS server will need at least 8 GB of data to even properly start. A single model run instance by a single user may additionally consume upwards of 2 GB of temporary memory. Considering that, a very conservative memory budget would be 16 GB of memory, and indeed, we have successfully run a DSS server with this being the maximum memory footprint allowed. Practically speaking though, testing with a 32 GB maximum memory footprint showed improved reliability and better throughput.

Server Cores – The DSS server is able to take advantage of multiple cores, improving the ability to simultaneously process multiple model computations, which in turn improves responsiveness of the software and facilitates rapid exploration of different scenario ideas. A minimum of 8 cores is highly recommended though we have successfully run a DSS server with only 4 or fewer, albeit at a significant performance penalty. Extending beyond 8 cores has not been tested though it is conceivable that it would increase the number of simultaneous scenarios that could be computed by different users assuming memory resources are not wholly consumed. Since it hasn't been tried, we cannot comment whether the Play! Framework or Java would need any additional configuration to ensure that it can fully exploit all computational resources available. The respective configuration and tuning documentation is left as an exercise for interested readers to pursue.

Server Disk – The DSS server startup delay is greatly reduced when a high-speed disk is available. Beyond that, server disk speed has little relevance to the performance of the DSS software itself except in the case of custom comparisons where one or more of the scenarios are cached on disk. The only other obvious exception where performance would suffer would be in memory-starved conditions where disk swap comes into play. Given that the DSS can cycle moderately large amounts of memory in a short time span, significant disk swap performance penalties for too little available memory can easily be avoided when more memory could have been allocated for use by the DSS server.

DSS Software Installation - Sample Production Startup Script

Starting the DSS application from the Play! command-line tends to not be a good solution, even in the short term, with many kinds of server setups. The reasons are varied and beyond the scope of this document. Our server administrators wrote the following startup script for our particular hardware and software configuration. It is not meant to be a universal solution but instead to showcase some considerations that a startup script for your own installation may want to consider.

```
#!/bin/bash
# Decision Support System
# chkconfig: 345 90 25

USER=dss
HOME=/home/$USER
APP_HOME=$HOME/DSS_Server
STAGE_DIR=$APP_HOME/target/universal/stage

# Source function library.
. /etc/rc.d/init.d/functions

start() {
      if [ ! -f $APP_HOME/RUNNING_PID ]
      then
            echo "Starting Decision Support System ..."
            su - $USER -c "cd $APP_HOME; $STAGE_DIR/bin/smartscape-dss -Duser.dir=/home/dss/DSS_Server
-mem 16000 &>$APP_HOME/logs/production.log &"
            wait_for_play
            echo "Starting httpd ..."
            /sbin/service httpd start
      else
            echo "Decision Support System is already running (PID file exists at $APP_HOME/RUNNING_PID)"
      fi
}

wait_for_play() {
      NETSTAT_CMD="netstat -anp | egrep '9000.*LISTEN'"
      #NETSTAT_OUT=$(netstat -anp | egrep '9000.*LISTEN')
        NETSTAT_OUT=$(eval $NETSTAT_CMD)
      while [ $? -eq 1 ]; do
                sleep 1
                NETSTAT_OUT=$(eval $NETSTAT_CMD)
      done
}

stop() {
      if [ -f $APP_HOME/RUNNING_PID ]
      then
            echo "Stopping Decision Support System"
            kill -s SIGTERM `cat $APP_HOME/RUNNING_PID`
            # Sleep to allow time for RUNNING_PID to be deleted
            sleep 2
            echo "Stopping httpd ..."
            /sbin/service httpd stop
      else
            echo "Decision Support System is not running (no file at $APP_HOME/RUNNING_PID)"
      fi
}

case $1 in
  start)
    start
    ;;
  stop)
    stop
    ;;
  restart)
    stop
    start
    ;;
esac
```

DSS Software Installation - Google Analytics

Overview

The DSS web application page has basic Google Analytics tracking built into it. The primary goal of this is to track uptake and reach. A secondary goal is to track the rough location of users. Google Analytics provides other services which allow deeper tracking of users' preferences but these features are not enabled.

For more information on Google Analytics, please refer to their website:

http://www.google.com/analytics

What this means to Developers

You should modify the application HTML page for the DSS software in one of two ways if you are running your own installation of the DSS software. The source for this HTML page is located in the source hierarchy at:

```
<application root>/app/views/index.scala.html
```

The two main options to consider are: a) you DO want to track users... or b) you DO NOT want to track users.

a) If you **DO** wish to track users, you will have to subscribe to the Google Analytics service (as of this writing, free), configure which types of information you would like to track (configured on the Google Analytics page), and then modify the line of the DSS HTML source (in blue) such that identifier key is now *your own* identifier key.

```
<script>(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function()
{(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),m=s.getElementsByTagName(o)
[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)})(window,document,'script','//www.google-
analytics.com/analytics.js','ga');
      ga('create', 'UA-60825214-1', 'auto');ga('send', 'pageview');
</script>
```

b) If you **DO NOT** wish to track users, you will have to remove the following lines of source from the application HTML page.

```
<script>(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function()
{(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1*new
Date();a=s.createElement(o),m=s.getElementsByTagName(o)
[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)})(window,document,'script','//www.google-
analytics.com/analytics.js','ga');
      ga('create', 'UA-60825214-1', 'auto');ga('send', 'pageview');
</script>
```

DSS Software Installation – Login Mailing Process

As part of the optional log in process, the DSS software will need to have access to a mail server from which the DSS server can issue emails to users wishing to register an account with your version of the DSS.

The DSS server is currently configured to connect via SMTP on port 25 using SSL. The DSS server is also currently set to authenticate with a fixed user and password. These settings are configured in the following Java file:

```
<application root>/app/SendMail.java
```

Specifially, this section will need to be configured for your installation:

```
email.setHostName("smtpauth.somemail.someschool.edu");
email.setSmtpPort(25);
email.setAuthenticator(new DefaultAuthenticator("dss@someorg.someschool.edu", "dss3m@il!!"));
email.setSSLOnConnect(true);
email.setFrom("dss@someorg.someschool.edu");
```

Note: our IT department configured this mail account to* only *be accessible from within the same network that the DSS server is running on. For this reason, using the mailer configured to defaults will issue Java execption errors upon attempting to connect to a hidden mail server. You will need to reconfigure this to access your own private mail account.

DSS Software Installation – Login Database

The optional user accounts are stored in an H2 database that the Play! Framework will automatically create if it does not exist. Thus this is not a true installation step but instead a process that will happen automatically as part of running the DSS server for the first time. For that reason, it's worthwhile to consider the settings that will be applied on this first startup during the installation process.

This process is controlled by the presence of a set of configuration settings in:

```
<application root>/conf/application.conf
```

Specifically:

```
# Database configuration
# ~~~~~
# You can declare as many datasources as you want.
# By convention, the default datasource is named `default`
#
db.default.driver=org.h2.Driver
db.default.url="jdbc:h2:./db/users"
```

Note: It is beyond this scope of this document to list all of the built-in database options available but it is even possible to use another database if desired (e.g., PostgreSQL), please see the documentation for the Play! Framework to explore how to do this:

https://www.playframework.com/documentation/2.2.x/ScalaDatabase

The DSS software relies on Ebean technology to automatically manage database access to the user account database table. In short, Ebeans are enabled in:

```
<application root>/conf/application.conf
```

Specifically:

```
# Ebean configuration
# ~~~~~
# You can declare as many Ebean servers as you want.
# By convention, the default server is named `default`
#
ebean.default="util.*"
```

And on the Java side, there are a series of options added into the ClientUser Java class which determines how the class data members corresponds to a database table. The ClientUser class is presented here:

```
@Entity
@Table(name="account")
public class ClientUser {
      @Id
      public String email;
      public String password;
      public String organization;
      public String passwordSalt;
      public boolean admin;
      public int accessFlags;
```

Note: It is beyond this scope of this document to explain all of the options available with Ebean technology. Additional resources and information can be found at:

https://www.playframework.com/documentation/2.2.x/JavaEbean
http://www.avaje.org/ebean/introduction.html

DSS Software Installation – Login Starting Accounts

When the H2 database is created for the first time, it will not have any existing users nor will anyone be set up with administrative capabilities by default. The developer has a couple of options for creating initial login accounts for administrators, including issuing SQL statements against the H2 database to create and configure the required accounts.

An alternative, which is automatic, is to specify the starting users and their access rights in a .yaml script responsible for creating the initial desired set of table entries. This script, which lists one or more user accounts to automatically create can be found here:

```
<application root>/conf/initial-data.yml
```

Example contents of that file are as follows:

```
# Users

users:

    - !!util.ClientUser
        email:        someone@gmail.com
        organization: WID
        admin:        true
        password:     secret

    - !!util.ClientUser
        email:        somename@gmail.com
        organization: WEI
        admin:        true
        password:     secret
```

Note: The starting password is irrelevant as there are additional fields stored with each client entry that are required to exist (specifically, a password* salt*) before logging in* can succeed. *In the absence of this additional required field, the users specified in the YAML script will have to utilize the "Forgotten Password" feature in the DSS software to update the password salt database field as well as replacing the above password string with the new password in a hashed format. This process will require the listed email addresses to be valid as the Forgotten Password feature hinges on being able to access an email as part of that process.