



Introduction

iConfigurator2.0 is a project to redesign and revamp the full architecture of the original iConfigurator. In this new redesign, the Customer Systems team members of Brad Ware, Mangesh Gondhalekar, and Yashan Dand decided to explore different technologies that could possibly be implemented in the new iConfigurator. This document describes what the MEAN stack is, how the technologies are used within an application, and why we chose to implement iConfigurator2.0 with this stack. It also displays the final workings of Brad Ware as an intern for the Customer Systems team. But first lets review currently what the iConfigurator technologies and design are providing, and why this limits the growth of AppleCare.

iConfigurator

Overview

iConfigurator is a tool that manages localization workflow for the iLog application. Any informational/error messages, notifications, and labels that get displayed within iLog are stored in iConfigurator ready to be translated into Chinese or Japanese. iConfigurator simply manages the input and output of these strings, and their translation status for the iLog application.

Technologies

The technologies used within iConfigurator on the server-side include Java, Apache Tomcat, and an SQL database. Java Servlet Pages are used to configure the routing from the HTTP requests to the database. The client-side technologies include HTML for content, CSS for styling, and JavaScript/jQuery for dynamic logic.

Shortcomings

These technologies and their implementation do have some issues with scalability and variability as the needs of iConfigurator will change over time. The current user-interface does not back iConfigurator as a multi-tenancy application,



as it only supports iLog and Contact Apple Support. These changes are also true in the back-end, where the SQL database has a very rigid style of organizing the data structure, and cannot be changed from application to application. This will be an issue if iConfigurator was every reorganized to be scalable as the back-end will never be flexible enough to support variability among different applications. There also is a considerable performance issue. iConfigurator fetches all of the data at once every-time the page is reloaded, and takes a considerable amount of time to display all of the data. It also takes time to search, create, and update existing rows within the table as it has to send the new updates back to the database. Finally, the user-interface of iConfigurator does not reflect the current design styles of recent Apple applications. These shortcomings lead overall to an unpleasant experience by the user.

MEAN Stack Technologies

Overview

MEAN is an acronym for Mongo Database, Express, Angular JavaScript, and Node JavaScript and is used to build complete web applications end to end. Each of these technologies plays a different role within the application and will be explained in detail within this document.

MongoDB

MongoDB is a NoSQL database that supports document-oriented storing capabilities. SQL style relational databases powered by technologies such as Oracle store data in tables that contain a set number of columns and each entry add a new row. This database design is usually excellent when you are certain with the size and set designs your database tables. However, when it comes to variability and scalability based on changes in demands, MongoDB is by far the better choice. Instead of using tables with a fixed number of columns and names, MongoDB uses collections. These collections store documents of data instead of rows, where each document can contain any different number of fields. This schema design allows flexibility and variability based on incoming data, where as in relational databases every row in the table contains the same columns. Even though MongoDB is an agile database, it still provides powerful querying through secondary and primary indexing.



SQL to MongoDB Concept Mapping Table

SQL Terms/Concepts	MongoDB Terms/Concepts
database	database
table	collection
row	document
column	field
index	index
table joins	embedded documents and referencing
primary key	_id field

Express

Express is node.js web application framework that helps organize your node.js files into MVC style architecture on the server-side. It includes a multitude of HTTP utility methods and middleware functions that make routing and creating an API simple and efficient. Because it is written on node.js, routes written through the Express module also can easily communicate to a database like MongoDB through an object modeling language (Mongoose, Monk, etc.). Express module can be downloaded and included into your node application through the node package manager.

Angular JavaScript

AngularJS is a structured web application framework that allows the user to extend HTML and express functionality with custom elements, attributes, classes, and much more. AngularJS supports the development style of MVC (Model-View-Controller), and runs only in the browser but provides a complete client-side solution to an application. Angular handles and structures the DOM for your application, and allows AJAX calls from the view to the model to run asynchronously. This is known as two-way data binding, and is continuously updating in the background as the user is submitting changes to the model and



dynamically appears as well in the view. However, AngularJS is not meant for all web applications. Angular excels in single-page applications because it has a lot of compilation to do each time a DOM is loaded into the browser. Therefore, Angular should not be used for round-trip application models, where different DOM pages are constantly to be loaded into the browser. However, on single-page applications based on CRUD operations, Angular is extremely simple and fast due to the asynchronous two-way data binding between the model and the view. Angular is NOT jQuery, they actually take completely different approaches to web application development. jQuery simply manipulates the DOM (view) of an application from the server-side logic dynamically. Angular however, transforms the browser into handling the majority of the logic for the application.

Node JavaScript

Node.js is a platform written in the JavaScript language that is built on Chrome's V8 engine that allows building simple, fast, and scalable network applications. Node.js is a single-threaded framework but is scalable through its asynchronous events and callback to support non-blocking IO. Simply put node.js is a server-side solution for JavaScript, and for receiving and responding to HTTP requests. Incorporating node.js for server-side logic allows your application to be unified across the stack by JavaScript, which incorporates JSON object modeling and resource sharing throughout the app. Node.js also comes with the Node Package Manager (NPM) which includes many modules such as Express, MongoDB, Mongoose, Jasmine and Karma for testing, and many more. These modules allow node.js to take care of all the server-side needs, including routing, API implementation, and communication with a database. And because it is in the JavaScript language, JSON object requests from client-side can be handled by the API and then stored in a database (preferably a document-oriented style DB). Node.js performs optimally when written in a high-traffic, data-intensive, and computationally light application.



iConfigurator2.0

Overview

As Apple continues to expand its product influence all over the world the AppleCare team will continue to advance into different regions as well. With this new increase comes a demand for support by the AppleCare team by a multitude of languages and dialects that currently iConfigurator was not designed to support. The original iConfigurator localizes text strings for iLog and stores them in three different languages however, it is not scalable enough to support the demand increase that Apple will bring in the near future. iConfigurator2.0 is a project that wants to redesign the original iConfigurator so that it can support a multitude of applications in conjunction with many languages to meet the demands of growth in the future by AppleCare and other support teams.

Goal

To explore, discover, and develop a frame of new technologies that will help iConfigurator be scalable enough to be the premiere localization application used within AppleCare and the rest of IS&T.

Scaffolding Structure

When working with new technologies, one is often faced with the challenge of organizing the source files in an efficient and intuitive manner. Every technology stack requires different strategies for integrating server and client side testing, incorporating front-end dynamic web content, and debugging during development. Fortunately, Node comes with its own package manager, NPM, which is a fast way for easy installation of modules. There are several standard directory structures, or archetypes, that are available with the MEAN stack. For our purposes, we chose to use MEAN.io, the most popular scaffolding structure for those implementing applications with MEAN. This structure has modular organization separating server-side from the client, and incorporates live-reloading and testing using Mocha for the server-side and Karma for client-side



code. Another advantage within MEAN.io is that it includes Twitter Bootstrap for easy styling when developing client-side. It also comes ready to use after installation, as all of the node_modules are already installed and included in the package.json file. The other popular scaffolding structure is MEAN.JS, however, we chose MEAN.io because it included more features and had a faster startup process. However, there is not much difference between the two and both of these scaffolding strategies are great for building organized and scalable MEAN applications.

Automated Build

With the development of every application comes the frustrations of repetitive builds, task management of structure, and testing your updated code. However, scaffolding using MEAN.io takes care of all of these tasks. MEAN.io comes with the automated task runner Grunt, which when configured in the Gruntfile.js, supports many features. A few of these luxuries when building your application with Grunt include: live reloading whenever a file is changed, automated test runner, and JSHint to keep your code clean and compiling. Grunt also includes many more plug-ins like jasmine-node for testing (will be covered next) that make rapid development possible. When developing MEAN applications, I strongly recommend using an automated task runner such as Grunt, and if you decide to scaffold a well know MEAN directory structure, many of them include it!

Testing

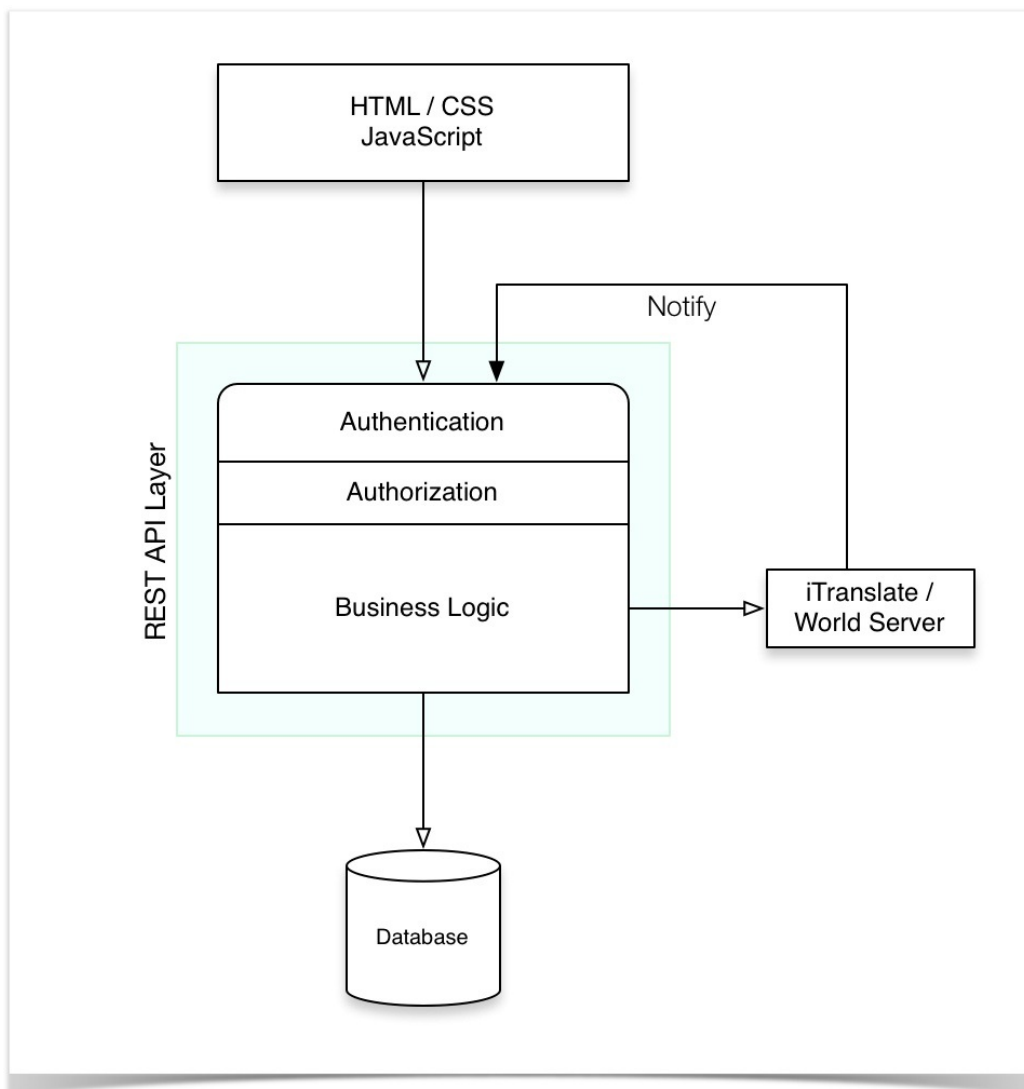
MEAN stack applications have a couple of choices for testing. Traditionally, client-side code would be tested using Karma as a test runner, and server-side testing would be done using Mocha. However, we have tested iConfig2.0 using Jasmine, which uses Karma HTTP servers to be tested on. Jasmine originally was just used for client-side testing on AngularJS code, but since the jasmine-node package was released, it has become increasingly popular to test server-side on node.js files. We chose Jasmine because it's syntax and BDD testing service had a very small learning curve, and integrates our testing to one technology, which can all be run through the task runner Grunt. Testing through Jasmine with the node plug-in is a new and enhanced feature to make BDD easy to test on the server-side as well. For server-side testing, we used Frisby, which is a REST API testing module built on Jasmine and



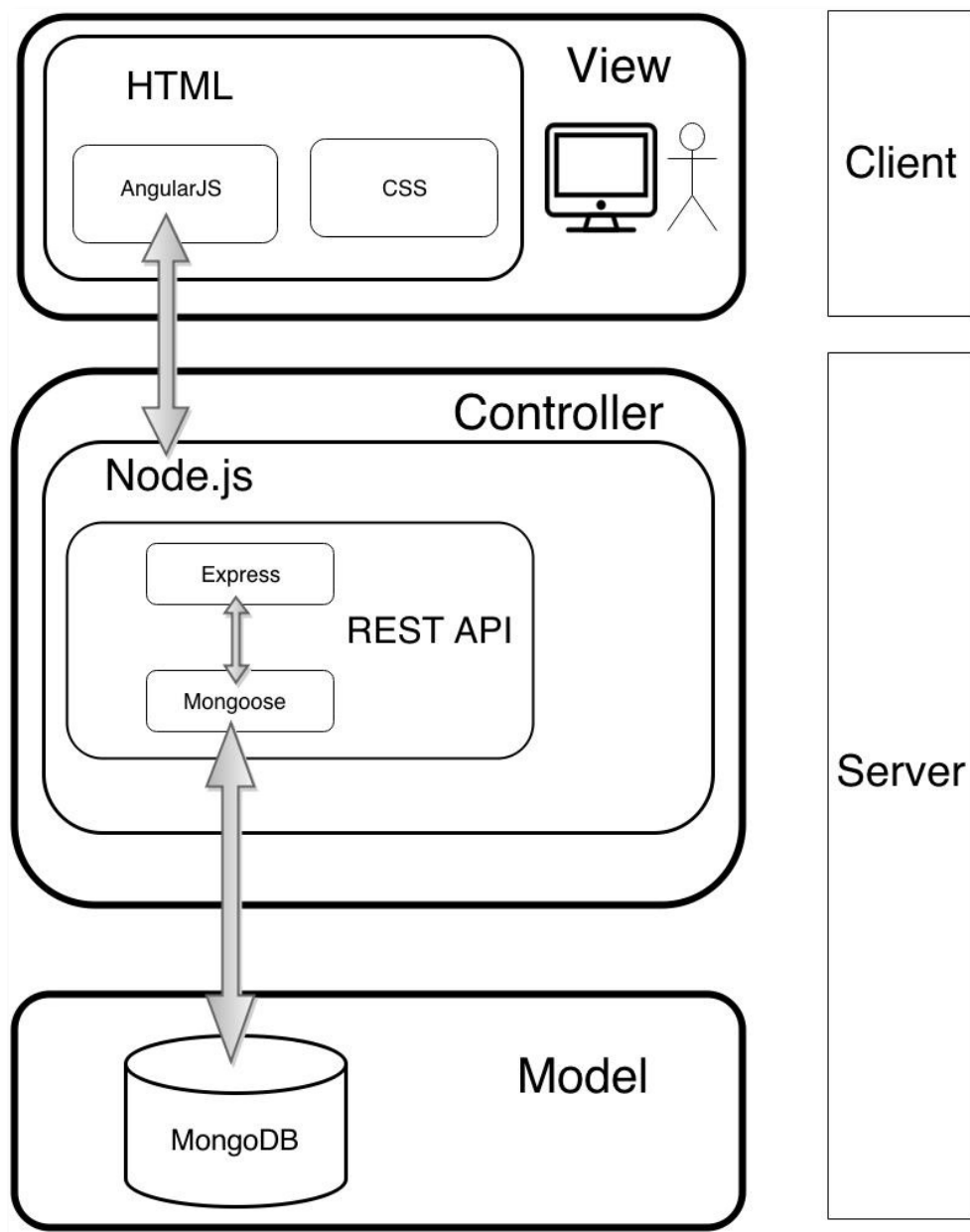
implementable through node.js. Frisby allows you to easily test your routing through Express to make sure your API functions are in sync with the back end database (MongoDB).

iConfigurator2.0 Architecture

Because iConfigurator2.0 used completely new technologies, we had to re-adjust and implement them into a MVC architectural style web application. A traditional basic MVC formatted web application has an overall architecture similar to this one:



The controller layer is typically written in Java or Python, communicating to the View layer which is written in Javascript/HTML/CSS, and then the back-end is implemented using a SQL database with a RESTful API. However, after deciding to use the MEAN stack technologies, we came up with an architecture displaying each technologies purpose within the application. Below is shown the MVC style application implemented with the MEAN stack technologies. Notice that now node.js is the main language 'controlling' the application, and that AngularJS is taking the role of jQuery when implementing dynamic web content on the client-side. The back-end has been completely replaced with MongoDB technologies.



Server-Side

The sections to follow will entail the server-side implementation and details of iConfigurator2.0. The technologies used on the server-side include MongoDB, Node.js, and Express.

Why MongoDB?

iConfigurator2.0 has chosen MongoDB simply because of its flexibility and room for growth. As iConfigurator2.0 is an application expected to grow and provide localization for many apps of AppleCare in the future, it needs a database that can grow and change just as frequently as the applications do. MongoDB provides the most flexibility in the design of its data architecture, as many of the applications supporting iConfigurator is supporting will need varied schemas over time. Here is a glimpse of the data models design for iConfig2.0.

Server-Side - Data Model Designs

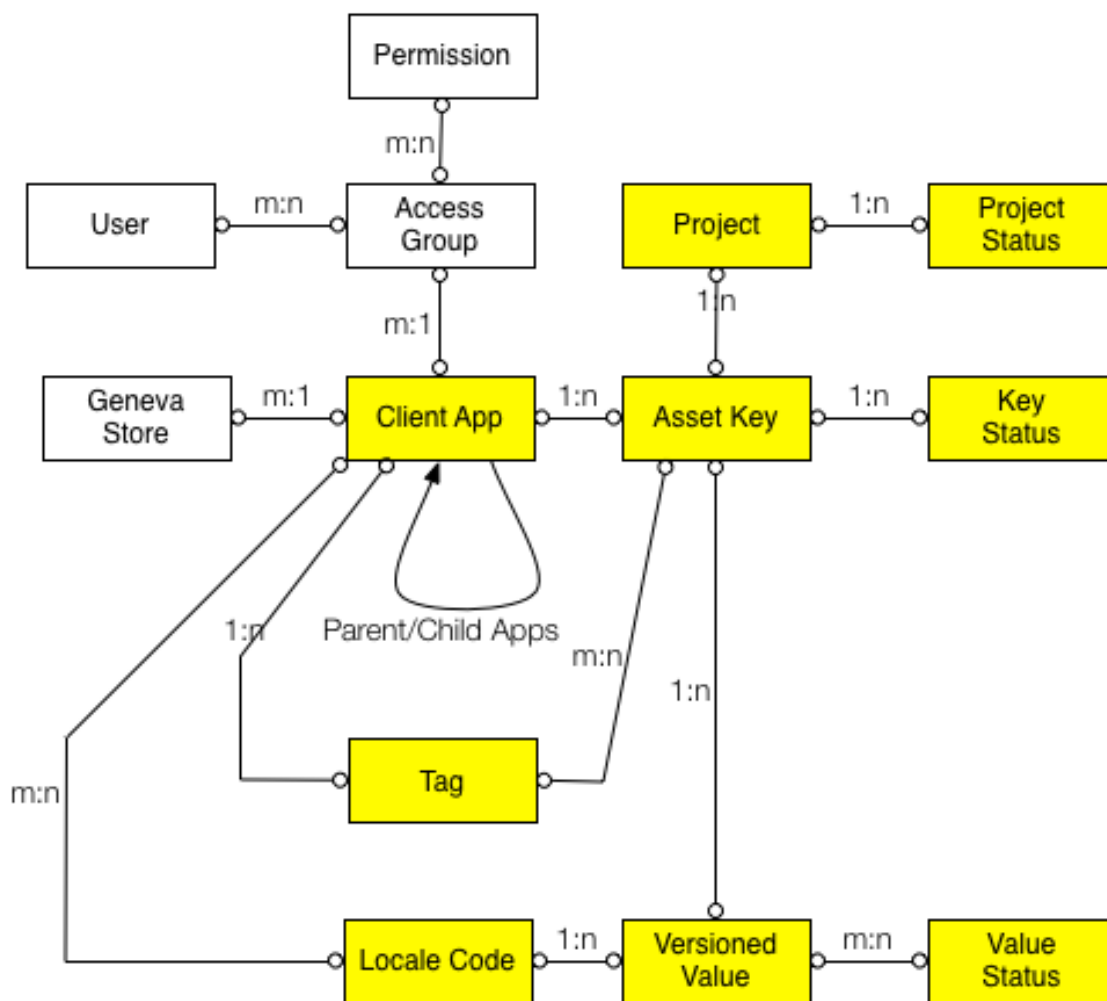
For data object modeling, we are using the node module Mongoose, which makes it simple to map out data objects through Schemas. Mongoose has a library of functions that directly communicates to the MongoDB to find, update, and delete documents. Mongoose can be installed through the node package manager, and is easily configured in node.js to connect to your applications database. The data models for iConfigurator2.0 have been implemented into a RESTful API style and communicated to MongoDB through Mongoose.

Data Model ReDesign - Strategy and Direction

For the design strategy, Mangesh originally drew up a data model structure to handle the needs of iConfigurator2.0 expansion purposes. This original design is shown on the next page (yellow was implemented in Brad Ware's scope). In the original design, we had AssetKey and AssetValue separated into different objects in the database. The status of every object was also its own entity as well. The locale code and Tag models were their own objects as well, taking in id fields and storage as objects in the database.



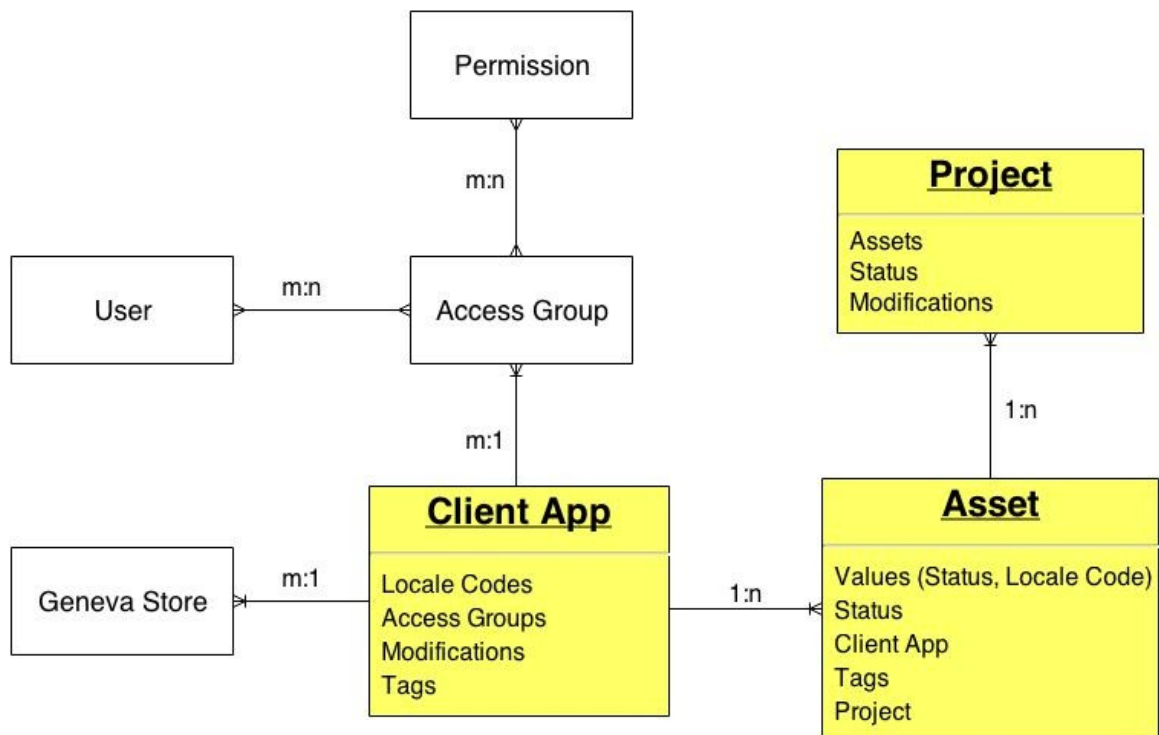
However, after examining the power of Mongo and the situations which allow it to be most efficient, we realized that these fields would be better as virtual objects. A virtual object in Mongo is embedded within the schema of an actual object, and has it's own distinct fields. However, it does not take up "physical" space in a database because it is not a document, and is simply apart of another object. Virtual objects also include an id field so that you can keep every one unique and searchable. The biggest change we made was combining the Asset Key and Asset Value object into one Asset object with it's own schema. Asset essentially took the role of Asset Key, and has a field of values that absorbed the data for Asset Value. Each Asset can have many values, however



every Value is associated with an Asset. The Status object of Asset, Project, and Client App were absorbed as virtual fields, as well as Locale Code being absorbed into Client App and Asset, and Tag into Asset. This condensed data



model makes it much easier for querying, and with Mongo referencing, relationships between documents are simple to retrieve. The new data model design is shown on the next page (yellow was implemented in Brad Ware's scope). There is a fine balance however as embedding too much can overflow the memory size of a Mongo document (16 MB). If the size of a document exceeds this threshold, it causes Mongo to allocate a new record (block of memory) and transfer over the data. This can be costly and inefficient, and should be considered in design strategies of data models. As you can see, the data model has been reduced greatly, balancing embedding and referencing relationships between the different data categories. Each of the models will have a collection in the MongoDB to hold it's documents, and Values will be a virtual object embedded inside the Asset collection. Locale code and status for Value are embedded inside it's virtual object.



Data Model Design - Strategy and Direction

Client App Collection

Client App Document/Object

```
{
  * __id: "_____",
  *name: "iLog",
  access_groups: [ "group_1", "group_2", "group_3" ...],
  *primary_locale_code: "en_US",
  *locale_codes: [ "en_US", "lang_1", "lang_2" ...],
  *created_by: "<user_id>",
  *created_date: "_____",
  last_modified_by: "<user_id>",
  last_modified_date: "_____"
}

* required
```



Asset Collection

Asset Document/Object

```
{
  *__id: _____,
  *name: "com.apple.example",
  *app_id: "<ilog_id>",
  *status: "Translated",
  tags: [ "tag_id_1", "tag_id_2", "tag_id_3" ...],
  description: "_____",
  project_id: "<project_id>",
  *created_by: "<user_id>",
  *created_date: "_____",
  last modified_by: "<user_id>",
  last modified_date: "_____",
  values: [ {
    *string: "Hello World",
    *status: "Not Translated",
    *locale_code: "en_US",
    *created_by: "<user_id>",
    *created_date: "_____"
  }, { ...} ]
}

* required
```



Project Collection

Project Document/Object

```
{  
    *__id: "_____",  
    *name: "InternProject",  
    *description: "Revamping iConfigurator",  
    *created_date: "en_US",  
    *created_by: "<user_id>",  
    statuses: [ {  
        *status: "Not Started",  
        *date: "_____"  
    }, { ... } ],  
    last_modified_by: "<user_id>",  
    last_modified_date: "_____",  
    assets: [ "asset_id_1", "asset_id_2", "asset_id_3", ... ]  
}  
  
* required
```



Routing and RESTful API

Obviously by the title the API used to handle basic CRUD operations from HTTP calls was written in a REST format. As such, depending on the functionality GET, PUT, POST, and DELETE were implemented on different object models. This was written in node.js through the Express module, which as explained earlier made routing to these different requests simple and efficient. All of the requests were transported as JSON objects, which made it extremely convenient for updating and transforming data across front end, middle-tier, and back- end. The API shown below was tested through Jasmine and the NPM module Frisby, which includes many functions for testing REST HTTP requests. The diagram shown below includes the implemented URI's and the associated HTTP method.

****Asset = JSON type Object consisting of one Asset Key, and a list of Asset Values associated with that Asset Key**



Data Model	HTTP Method	URI	Description
	GET	/config	Returns HTML content and resources needed to bootstrap the iConfig application. Essentially, this is the iConfig application's "index.html/index.jsp" file
APPS			
	POST	/apps	Creates a Client App
	PUT	/apps/:app_id	Updates the attributes of a Client App
	GET	/apps/:app_id	Gets the attributes of a single Client App
	GET	/apps	Gets the attributes of all the Client Apps
	DELETE	/apps/:app_id	Deletes a Client App
ASSETS			
	POST	/apps/:app_id/assets	Creates an Asset
	PUT	/apps/:app_id/assets/:asset_id	Updates the attributes of an Asset
	PUT	/apps/:app_id/assets/?name=asset_key_name	Updates attributes of Asset matching search by name
	GET	/apps/app_id/assets/:asset_id	Gets attributes of Asset based on Asset ID



Data Model	HTTP Method	URI	Description
	GET	/apps/:app_id/assets/?name=asset_key_name	Gets attributes of Asset matching search by name
	GET	/apps/:app_id/assets	Gets all of the Assets associated with the App ID
	DELETE	/apps/:app_id/assets/:asset_id	Deletes the Asset with the ID
	DELETE	/apps/app_id/assets/?name=asset_key_name	Deletes Asset with name matching search
ASSET TAGS			
	POST	/apps/:app_id/assets/:asset_id/tags	Adds tags to an Asset
	PUT	/apps/:app_id/assets/:asset_id/tags	Replace the existing tags of an Asset with a new set of tags
	GET	/apps/:app_id/assets/:asset_id/tags	Gets all tags associated with an Asset
	DELETE	/apps/:app_id/assets/:asset_id/tags/tag_name	Remove the specified tag from an Asset
	GET	/apps/:app_id/tags	Get all tags used by an App
LOCALES			
	POST	/apps/:app_id/locales	Adds new locales to an App
	PUT	/apps/:app_id/locales	Replace existing locales with an App with a new set of locales



Data Model	HTTP Method	URI	Description
ASSET VALUES	GET	/apps/:app_id/locales	Get all of the locales associated with an App
	POST	/apps/:app_id/assets/:asset_id/values	Add a Value to an Asset
	GET	/apps/:app_id/assets/:asset_id/values/current	Gets the current Values of an Asset
	GET	/apps/:app_id/assets/:asset_id/values/current/?locale=locale_code	Gets the current Values of an Asset based off a locale code
PROJECTS	GET	/apps/:app_id/assets/:asset_id/values	Get all Values associated with an Asset
	POST	/projects	Creates a Project.
	GET	/projects	Gets all of the Projects
	PUT	/projects/:project_id	Updates the attributes of a Project
PROJECT STATUS	GET	/projects/:project_id	Gets the attributes of a Project
	DELETE	/projects/:project_id	Delete's a Project



Data Model	HTTP Method	URI	Description
	POST	/projects/:project_id/statuses	Adds a status to a Project
	GET	/projects/:project_id/statuses	Gets all of the statuses for a Project
	GET	/projects/:project_id/statuses/?date=status.date	Retrieve statuses created on that date
	GET	/projects/:project_id/statuses/?info=status.info	Gets all of the statuses for Project by searching through info of each Project
	GET	/projects/:project_id/statuses/current	Get details about current status for a Project
	GET	/projects/:project_id/statuses/:status_id	Gets a status for a Project by the status_id
PROJECT ASSETS			
	POST	/projects/:project_id/assets	Add Assets to existing Project
	DELETE	/projects/:project_id/assets/:asset_id	Remove Assets from an existing Project
	GET	/projects/:project_id/assets	Get all Assets for Project



Client-Side

For the front-end (view) portion of iConfigurator2.0, we decided to use the technologies of HTML, CSS, and AngularJS. HTML was used to define the content in a DOM format, CSS files were included to style the pages, and AngularJS served to dynamically update the content as the customer was using the application. AngularJS receives the JSON objects from the RESTful API that was implemented and routed through Mongoose and Express, and performs the business logic on the data to display to the client.

Why Angular?

If your wondering why AngularJS was chosen as a client-side framework for iConfigurator2.0, then don't worry, your about to find out. iConfigurator is a simple single-page application where the user is mostly interacting with a giant table of strings that allows editing, deleting, adding and sending new assets to the world server. As administration and developers will be most likely the only users allowed access to multiple applications, iConfigurator is an indisputable traditional single-page application for most of the customers. That means the DOM page will not be loaded often into the browser, taking full advantage of the asynchronous dynamic two-way binding Angular has to offer with the data transfer between the model and view. This simplified binding between the model and view is extremely fast, and will allow the user to see updates to keys they have made, and to live reload more results of keys as they scroll down the page. As an issue with the original iConfigurator was the loading of data being too large per page, Angular will allow users to display only the current results that will fit on screen, and then quickly update as the user scrolls down. Angular also supports a RESTful data model, which is the strategy we wanted to interact with the MongoDB.

Client-Side Routing

API routing on the client-side using AngularJS typically involves the choice of 2 options: \$http and ngResource. Both of these support CRUD options, and are valid for interacting with a Mongo database. Below details the exact differences between the two.

1. \$http - an Angular service that makes AJAX requests to a server. \$http can manually be used to make POST, PUT, GET, and DELETE requests to a server. \$http is a general purpose, and can be configured for any API style.



2. `ngResource` - performed through the `$resource` factory in Angular, `ngResource` is a higher-level implementation of `$http` configured specifically to interact with a RESTful API. `ngResource` has action methods which provide high-level behaviors without the need to interact with the low level module, `$http`. Supports GET, POST, and DELETE requests to a server through high level functions such as `save()`, `query()`, and `remove()`.

`ngResource` was chosen as the Angular factory for client-side routing in `iConfigurator2.0` simply because it is much straightforward to use. `ngResource` is tailored towards RESTful API's, which is what `iConfig2.0` uses, therefore many of the built-in functions like `remove()`, `get()`, and `query()`. However, it does not come with an inherent PUT request, and has to be inserted manually at the controller level. Also, the POST/CREATE functionality of the `$resource` factory defaults to the function `$save` in code, which can be confusing to the author because the name doesn't accurately represent the action. Therefore, I overwrote the `$save` method to be called `$create()`, which seemed much more appropriate. `ngResource` is simply configured at the top of the `ngController` body, and just needs an URI address. This address can take in parameters, such as `asset_id`, which Angular will responsively configure based on the `hg-model` calling the routing functions. The `ngResource` implementation is extremely simple and quick to get basic API calls up and running.

UI-Router vs. ngRoute

Two views were implemented for `iConfigurator2.0` (while Brad Ware was here). The first is the landing page called Home, and it simply allows the user to navigate to different areas of the web application. The second page called Workspace, is where the customer will be spending the majority of the time while using `iConfigurator2.0`. Routing in Angular is centered around two main libraries, `ui-router` and `ngRoute`. `ngRoute` is the default framework, and organizes each view by a different URI, which is separated in the `index.html` file under the `<ng-view>`. For routing in `iConfig2.0`, I used the UI-Router library which is a framework that organizes the interface by a state machine, rather than a URL route. UI-Router allows the user to nest views, and instead of using `<ng-view>` in the index file, the tag `<ui-view>` must be applied. With UI-Router, Angular is now concerned which state the user is in, and then load the URL associated with that state. The `ui.router` module has to be injected as a dependency in your application-level module to be used, and the states can be declared in the `.configure()` method of each controller. The states are declared with the `$stateProvider` method (as opposed to `$routeProvider`), and have to have a template file (html) and url declared with each state as well. With this setup, one can declare nested states that are associated with hierarchy relationships of views, and then append more



definition to each url route as you go deeper in the parent:child relationship. UI-Router is a more intuitive way to provide routing for your application, and makes it much easier to organize your logic by states of the application.

iConfigurator2.0 Routing

For iConfigurator2.0, since the user will be interacting with 2 views (due to Brad Ware's limited time), two states were used as well with UI-Router. Both of these states are inside of the <ui-view> tag of the driver template, index.html. Index.html loads all of the CSS and JavaScript file dependencies that iConfigurator2.0 uses. As the user switches between states, the ui-view will load the associated template file. The two states with their associated url and views are shown below.

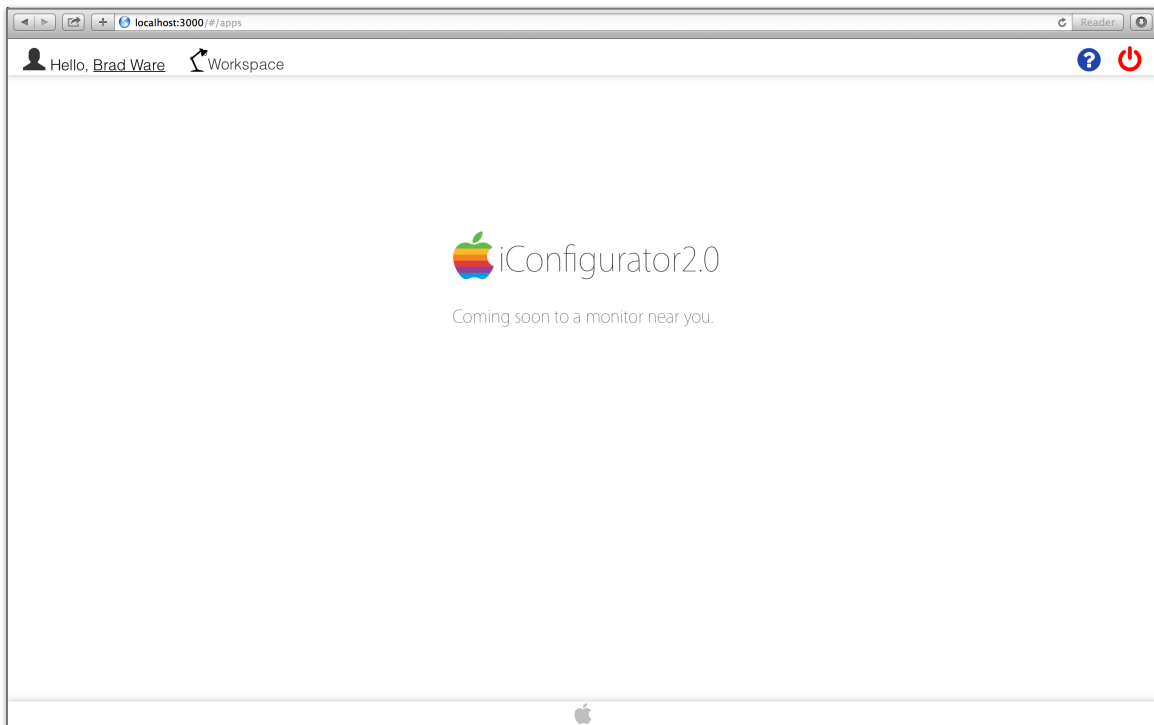
State	URL	Template	Module	Controller
home	http://localhost:3000/#/apps	apps.html	appsMod	appsCtrl
workspace	http://localhost:3000/#/assets	assets.html	assetsMod	assetsCtrl

Home View

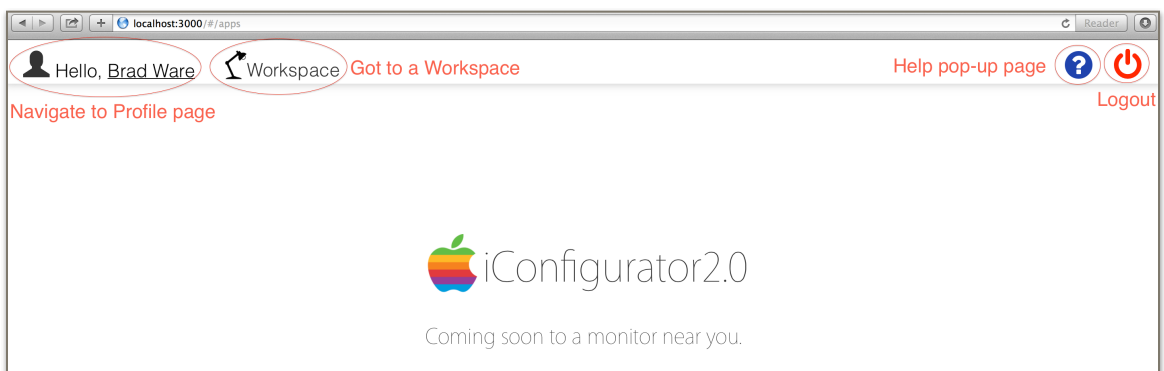
The Home state was used to show the landing page of iConfigurator2.0, inside of apps.html. Apps.html template is for the user to navigate to different parts of the application. From this view, the user can open Workspace with different Client Applications supported by iConfigurator2.0, transition to the Profile page to edit their settings (new feature), open a pop-up window that answers frequent questions and includes a description of iConfigurator2.0, and lastly, logout. A series of screenshots on the next page show the functionality behind the Home view.



Home View Overall

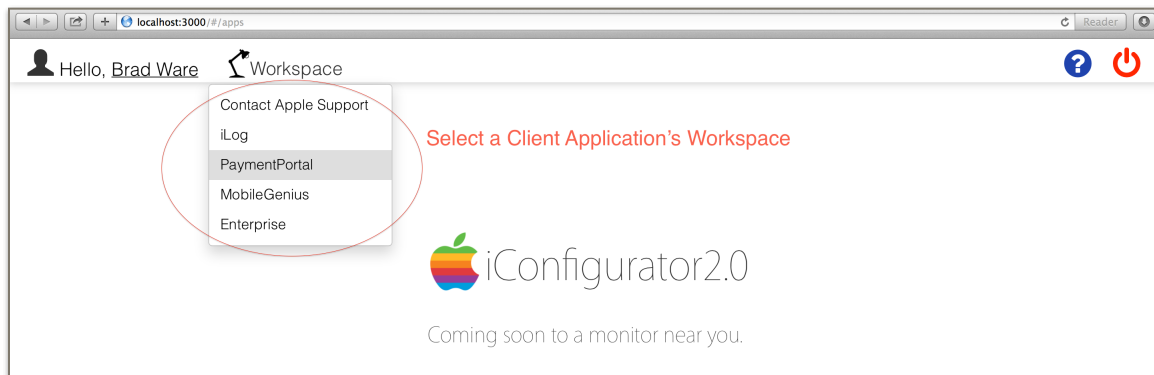


Home View Navigation Bar



The navigation bar on the home screen allows the user to quickly navigate to different parts of the application. Each of the icons on the bar serves a different purpose, and each is described in the image above.

Home View Workspace Drop-Down



The drop down under Workspaces is a new functionality implemented in iConfigurator2.0. This allows the application to be multi-tenancy and provide service to many Client Applications, instead of just iLog. You simply just click on the Workspace text and the drop down appears, and after that choose a Client Application to launch the Workspace view. This will display the Assets only belonging to the Client App the user chose, and can be changed at any time on the toolbar for the Workspace view.

Workspace View

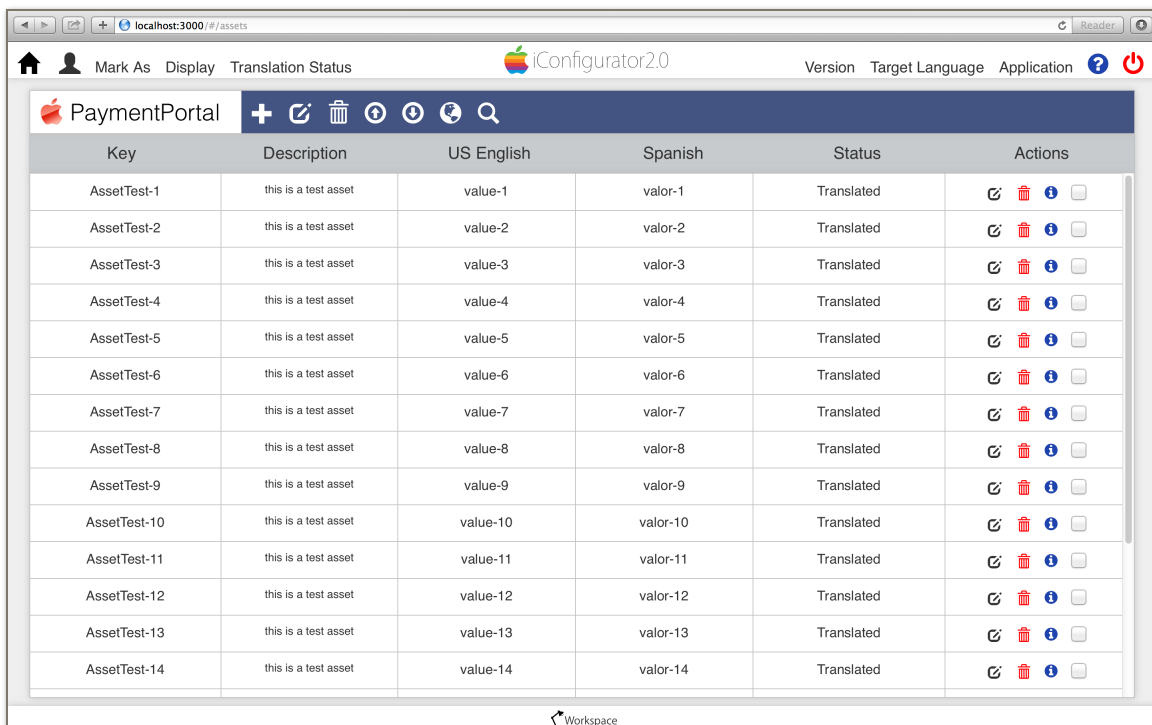
The Workspace view is where the user will be spending the majority of their time while logged into iConfigurator2.0. This view shows all of the Assets associated with a Client Application, and allows the user to add, edit, delete and view more info about individual Assets and the Values associated with each one. It also allows the user to navigate to the Home View, Profile page, Help pop-up window, and logout. A series of screenshots on the next page show the functionality behind the Workspace view.



























































ngTable

ngTable is a module library framework within Angular that allows easy production of tables with two-way binding to the model. It includes example code of numerous tables with different styles and features. ngTable is known for being extremely fast at displaying and updating table information as the user edits the view. An alternative to ngTable is ngGrid, however ngGrid is known performance wise to be quite a bit slower than ngTable, and therefore was not chosen to be implemented in iConfigurator2.0. Instead ngTable displays all of the assets that belong to a Client Application, and is editable by the user. ngTable was used in iConfigurator2.0 only within the Workspace view.

Workspace View Overall



The screenshot shows the iConfigurator2.0 application interface. At the top, there is a navigation bar with a home icon, a user profile icon, and the text "Mark As Display Translation Status". To the right of this bar is the application title "iConfigurator2.0" and a status bar with "Version", "Target Language", "Application", and a power icon. Below the navigation bar is a dark blue header for the "PaymentPortal" section, which includes a plus icon, a trash icon, and a search icon. The main content area is a table with the following columns: "Key", "Description", "US English", "Spanish", "Status", and "Actions". The table contains 14 rows of test assets, each with a unique key (AssetTest-1 to AssetTest-14), a description ("this is a test asset"), and values for the two languages. The status for all assets is "Translated". The "Actions" column contains four icons: a pencil (edit), a trash can (delete), an information icon (details), and a checkbox (toggle).

Key	Description	US English	Spanish	Status	Actions
AssetTest-1	this is a test asset	value-1	valor-1	Translated	   
AssetTest-2	this is a test asset	value-2	valor-2	Translated	   
AssetTest-3	this is a test asset	value-3	valor-3	Translated	   
AssetTest-4	this is a test asset	value-4	valor-4	Translated	   
AssetTest-5	this is a test asset	value-5	valor-5	Translated	   
AssetTest-6	this is a test asset	value-6	valor-6	Translated	   
AssetTest-7	this is a test asset	value-7	valor-7	Translated	   
AssetTest-8	this is a test asset	value-8	valor-8	Translated	   
AssetTest-9	this is a test asset	value-9	valor-9	Translated	   
AssetTest-10	this is a test asset	value-10	valor-10	Translated	   
AssetTest-11	this is a test asset	value-11	valor-11	Translated	   
AssetTest-12	this is a test asset	value-12	valor-12	Translated	   
AssetTest-13	this is a test asset	value-13	valor-13	Translated	   
AssetTest-14	this is a test asset	value-14	valor-14	Translated	   



Workspace View Navigation Bar



The Navigation bar is where the user can edit the table data as a whole, as well as navigate from the page. The Home icon takes you back to the Home page, and the Profile link will take the user to the Profile settings page. Logout and the Question pop-up page on the right side provide the same functionality as on the Home page. Mark As, Display, Translation Status, Version, Target Language, and Application all provide overview functionality for the data set as a whole. The Application drop-down lets the user choose which Client Application they are choosing to view the Asset data from, and allows the user to quickly transition between apps.

Workspace View Search Table

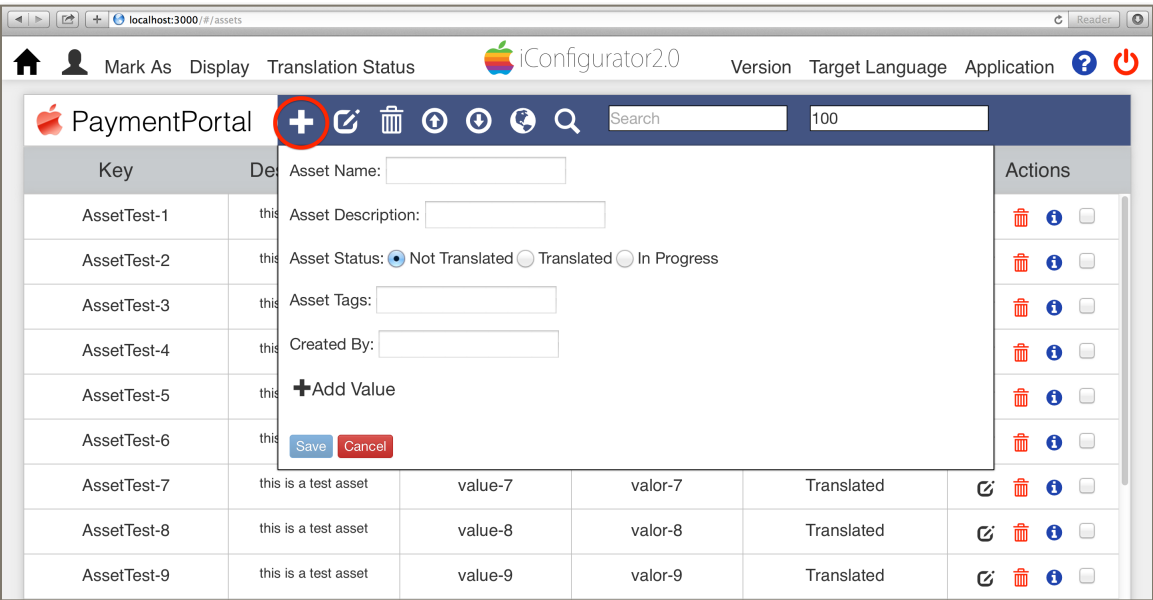


The search icon allows the user to quickly find any row of data from the table. The search field algorithm is based on every row of data associated with an Asset, even the fields that are not displayed in the main table. The other input



bar lets the user choose how many rows they want to display at a time. Both of these fields pipe all the data and the result is the table view.

Workspace View Create Asset



The table toolbar allows the user many functions including adding a new Asset to a Client Application. This pop-up window is accessed through the plus icon, and the user can simply enter the fields for the Asset. The functionality is protected to only allow the user to enter valid data for certain fields, and to enter data for the required fields. After saving the newly created Asset, it will instantaneously appear at the top of the table for review by the user.

Workspace View Table Header Tools



The rest of the header tools also provide key functionality for usage of the table. The icon left of the plus allows the user to edit multiple fields of data at



once. The trash icon allows users to delete many Assets at once. The up and down arrow allow the user to upload and download files associated with an Asset. Finally the world icon lets the user send the Assets to the World Server.

Workspace View Asset Row Tools

PaymentPortal					
+ [edit] [trash] [upload] [download] [world] [search]					
Key	Description	US English	Spanish	Status	Actions
AssetTest-1	this is a test asset	value-1	valor-1	Translated	<div> <div>Edit</div> <div> <div>[edit]</div> <div>[trash]</div> <div>[info]</div> <div>[checkbox]</div> </div> </div>
AssetTest-2	this is a test asset	value-2	valor-2	Translated	<div> <div>[edit]</div> <div>[trash]</div> <div>[info]</div> <div>[checkbox]</div> </div>
AssetTest-3	this is a test asset	value-3	valor-3	Translated	<div> <div>Delete</div> <div> <div>[edit]</div> <div>[trash]</div> <div>[info]</div> <div>[checkbox]</div> </div> </div>
AssetTest-4	this is a test asset	value-4	valor-4	Translated	<div> <div>[edit]</div> <div>[trash]</div> <div>[info]</div> <div>[checkbox]</div> </div>

The tools on each row provide the user easy accessibility to update information on an individual Asset. The edit and trash icons provide the same functionality as the table header clone, except just on the individual Asset. The blue info icon provides a window when clicked that shows all of the metadata for a specific Asset. This metadata can include who created the Asset, the last time it was edited, and what user edited the Asset.

One More thing...

Below are a series of design ideas by Brad Ware to improve the scalability, functionality, and interface of iConfigurator2.0. Unfortunately my internship scope cut my time short to implement these designs, but they quickly can be with the functionality of Angular. They include designs for the new Profile page, where the user can update their settings and preferences. It also includes design wire frames for the slide-in animation of the Details page, which includes metadata of each Asset, and the Send to World Server pop-up window.



Profile Page Wire Frames

Settings.

My Info


iLog

App_2

App_3

App_4

...



John Doe

Email Address [Edit](#)

Phone Number [Edit](#)

Dept [Edit](#)

Location [Edit](#)

Default Application [?](#)

Rows loaded per iteration [?](#)

☒ Apply to All Applications

☒ Redirect to WorkSpace on Login [?](#)

[Save Changes](#)

Settings.

My Info

iLog

App_2

App_3

App_4

...

Layout & History

Language Preferences

Rows loaded per iteration

Default Target Language

Column Preferences

Column Name	Column Displayed (Up to 5)	Optional
Key	<input checked="" type="checkbox"/>	No
Source	<input checked="" type="checkbox"/>	No
Target	<input checked="" type="checkbox"/>	No
Category	<input checked="" type="checkbox"/>	No
Description	<input checked="" type="checkbox"/>	No
State	<input type="checkbox"/>	No
Reference File	<input type="checkbox"/>	No
AnotherCol1	<input type="checkbox"/>	Yes
AnotherCol2	<input type="checkbox"/>	Yes
AnotherCol3	<input type="checkbox"/>	Yes

Translations History

Request	Date	Status
1132	May 2nd 2014 6:45 pm	In Progress
1090	April 4th 2014 8:42 am	Completed
1111	Jan 7th 2014 7:32 am	Cancelled
3244	Dec 2nd 2013 4:44 pm	Cancelled
1212	Nov 7th 2013 2:22 pm	Completed
7431	Oct 12th 2013 1:35 pm	Completed


[Save Changes](#)





Settings.

	Layout & History	Language Preferences
My Info		
iLog		
App_2		
App_3		
App_4		
...		

▼ All

 United States ☒

 Chinese ☒

 Japanese ☒

► Asia-Pac
► Americas
► EMEA


Save Changes


Settings.


	Layout & History	Language Preferences
My Info		
iLog		
App_2		
App_3		
App_4		
...		


► All


▼ Asia-Pac

 Yemen ☒

 Chinese ☒

 Japanese ☒

 Georgia ☐

 Vietnam ☐

► Americas
► EMEA

Save Changes



Details Page Wire Frame

Details

Key

Source

Target

Description

Ware.Brad

Doe.John

Cook.Tim

James.LeBron

Details.

Row Data

Column Name	Column Data	Edit
Key	Ware.Brad	Edit
Source	Hello World	Edit
Target	ハローワールド	Edit
Category	Greeting	Edit
Description	Example1	Edit
State	Translated	Edit
Reference File	None	Edit

Edit All

Save Changes

Row History

Key

Ware.Brad

Application

iLog

Number Languages Supported

3

Last Edited

06/01/14 3:27 PM

Last Edited By

Brad Ware

Entered Originally

Brad Ware

Entered Originally By

06/01/14 3:27 PM

Translation Request Status

Request 100 in Progress

Product Version

02-May-2014 6:45 am

Send to World Server Wire Frame

Send to World Server

Key

Source

Description

Ware.Brad

Hello World

What's Up Work

Apple is Awesom

Basketball is Av

Translation Request

Project Name

newName

Specific Instructions

Instructions

Other Instructions

newCategory

Group Email

newEmail

Due Date

newDate

Workflow

AppleCare Workflow...

Reference File

Attach

Rush

No

Send All Not Translated

Ware.Brad

Delete

Key.Ex1

Delete

Submit



Resources

- <http://confluence.corp.apple.com/display/ACCCS/iConfigurator+Phase+2+Design>
- <http://mean.io/#!/>
- <http://mongoosejs.com>
- <http://www.mongodb.org>
- <http://expressjs.com>
- <http://nodejs.org>
- <https://angularjs.org>
- <http://ux.apple.com>
- <https://iconfigurator.corp.apple.com/login>

