AUTODESK UNIVERSITY 2013

Cloud-Based, Real-Time, Round-Trip, 2D Revit Model Editing on Any Mobile Device

Jeremy Tammik – Autodesk

DV1736 This presentation demonstrates real-time, round-trip editing of a simplified 2D rendering of an Revit BIM on any mobile device with no need to install any additional software whatsoever beyond a web browser. How can this be achieved? A Revit add-in exports polygon renderings of room boundaries and other elements such as furniture and equipment to a cloud-based repository that is implemented using an Apache CouchDB NoSQL database. On the mobile device, the repository is queried and the data rendered in a standard browser using server-side generated JavaScript and SVG. The rendering supports graphical editing, specifically translation and rotation of the furniture and equipment. Modified transformations are saved back to the cloud database. The Revit add-in picks up these changes and updates the BIM in real-time. All of the components used are completely open source, except for Revit itself. This is an advanced class for experienced programmers.

Learning Objectives

At the end of this class, you will be able to:

- Architect a cloud-based data repository using NoSQL and Apache CouchDB
- Implement server-side scripting to display and edit 2D graphical data in the browser on a mobile device
- Understand the JavaScript implementation using jquery, db and Raphaël to generate and drive the HTML and SVG room editor
- Use the Revit API to determine room and family instance 2D boundary polygons and the Idling event for real-time BIM updates

About the Speaker

Jeremy is a member of the AEC workgroup of the Autodesk Developer Network ADN team, providing developer support, training, conference presentations, and blogging on the Revit API.

He joined Autodesk in 1988 as the technology evangelist responsible for European developer support to lecture, consult, and support AutoCAD application developers in Europe, the United States, Australia, and Africa. He was a co-founder of ADGE, the AutoCAD Developer Group Europe, and a prolific author on AutoCAD application development. He left Autodesk in 1994 to work as an HVAC application developer, and then rejoined the company in 2005.

Jeremy graduated in mathematics and physics in Germany, worked as a teacher and translator, then as a C++ programmer on early GUI and multitasking projects. He is fluent in six European languages, vegetarian, has four kids, plays the flute, likes reading, travelling, theatre improvisation, carpentry, and loves mountains, oceans, sports, and especially climbing.

jeremy.tammik@eur.autodesk.com

Table of Contents

Overview

Before we dive into the nitty-gritty details, let us mention some governing principles and provide an architectural overview of our topic.

Quotes on Three Fundamental Aspects

Here are some quotes supporting three of the governing principles followed in the implementation of the 2D model editor, and indeed in all my really serious work:

- Lazy
  - ... develop the three great virtues of a programmer: laziness, impatience, and hubris – Larry Wall
- Simple
  - Simplicity is the ultimate sophistication – Leonardo da Vinci
  - There is no greatness where there is no simplicity – Leo Tolstoy
  - KISS
- Perfect
  - Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away – Antoine de Saint-Exupéry

I assume there is no question about the importance and effectiveness of simplicity and perfection.

The quote from Larry Wall is well know in the developer community and may seem a little more controversial at first sight. In the first edition of Programming Perl, Larry says: "We will encourage you to develop the three great virtues of a programmer: laziness, impatience, and hubris."

The second edition of the book includes a glossary providing pithy definitions for each of these terms:

- Laziness – The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labour-saving programs that other people will find useful, and document what you wrote so you don't have to answer so many questions about it. Hence, the first great virtue of a programmer. Also hence, this book. See also impatience and hubris. (p. 609)
- Impatience – The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to. Hence, the second great virtue of a programmer. See also laziness and hubris. (p. 608)
- Hubris – Excessive pride, the sort of thing Zeus zaps you for. Also the quality that makes you write (and maintain) programs that other people won't want to say bad things about. Hence, the third great virtue of a programmer. See also laziness and impatience. (p. 607)

One of my goals in this presentation is to show you a convincing example of starting off on a project with little prior knowledge, going down numerous wrong alleys, and through diligent application of these virtues ending up with a perfect, simple and minimal solution.

Architectural Basics

Data Source, Repository and Consumer Client

The first aspect of this project is the visualisation of a simplified 2D representation of a Revit BIM on any mobile device, involving the following three main components:

- BIM – Building Information Model
- Cloud-based data repository
- 2D rendering on mobile device

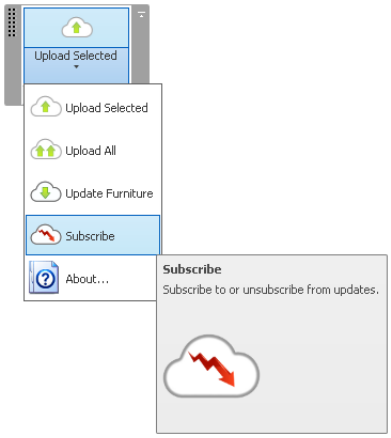
Real-time Editing Triggers Database and BIM Update

We want much more than just that one-way communication, though: the simplified 2D model on the mobile device can be edited, the changes are reflected back via the cloud database to the 3D BIM model, and this is completed in real time, so that updates are immediately obvious:

- Graphical room editor on mobile device
- Update cloud database
- Reflect real-time changes in BIM

Base Technologies

- BIM – Revit
- Data repository – NoSQL
- Rendering and editing – HTML and SVG



Implementation Environment

- BIM – Revit .NET add-in
- Database – Apache CouchDB
- JavaScript – jquery, db, Raphaël, SVG, HTML

Simpler Still

- Same origin policy
- Server-side scripting
- Two components instead of three

The Two and Only Projects

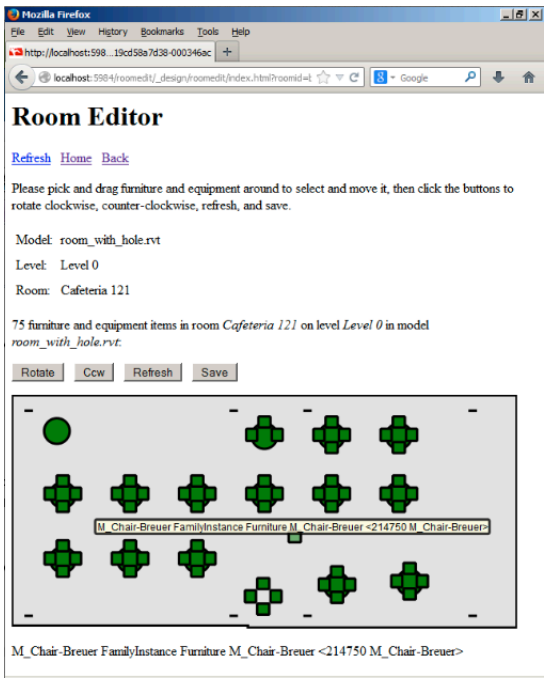
- Revit add-in
- CouchDB database

Revit Add-in

The functionality provided by the Revit add-in is pretty interesting, including:

- Determine 2D boundary polygon loops
- Upload model to cloud database
- Download changes from cloud database
- Subscribe to real-time updates using Idling event

We'll start off by looking first at the exciting cloud stuff and return to the add-in afterwards, understanding the details of what it is used for.



Demonstration

Note the simple navigation

- Home page: list all models and select one
- Model selected: list all levels and select one
- Level selected: list all rooms and select one

Once a room is selected, we can display the room boundary and the furniture it contains using the symbol graphics and enable

the editor, which supports click and drag to modify the family instance transformations. Clicking the save button updates the cloud database.

Note that all interaction with the application is driven by a REST API. You can see the URLs being used in the address bar. The database interaction is RESTful as well

Free and Simple

This entire application is 100% based on free open source components.

It took me as a total newbie about 200 hours to research and implement, starting from scratch with zero previous cloud database knowledge.

Now that I understand the basics and know which tools I can use and how, it would take less than 8 hours to install all required components and rebuild it from scratch.

In fact, it is so small and compact that the entire application is contained in one single file, index.html:

https://github.com/jeremytammik/roomedit/blob/master/index.html

The CouchDB Database

The cloud database is implemented using CouchDB, a NoSQL database.

NoSQL

For a while, it seemed as if all databases were SQL and used the ACID transaction paradigm. However, massive global systems forced the insight that they were not scalable enough.

- Next generation database paradigm
- Address some of the points: non-relational, distributed, open-source, horizontally scalable
- Began 2009 and growing fast
- Frequent other characteristics: schema-free, easy replication support, simple API, eventually consistent / BASE (not ACID), huge amounts of data
- "Not only SQL"
- http://nosql-database.org, https://en.wikipedia.org/wiki/NoSQL

Acid versus Base

https://en.wikipedia.org/wiki/ACID

In traditional computer science, ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee that database transactions are processed reliably.

http://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base

An alternative to ACID is BASE:

- Basic Availability
- Soft-state
- Eventual consistency

The database is not guaranteed to be in a consistent state at any given moment. Amazon may end up selling the same book twice. However, consistency is guaranteed, eventually, so just hang on for a moment while we sort this out.

CouchDB Database Implementation

In a CouchDB database, everything is a document, and all documents are in JSON format. Every document has built-in id and revision.

We can let the database generate the ids automatically as the documents are added, or assign them ourselves. In our case, we use the Revit database unique ids, making it very easy to maintain the bidirectional links back and forth.

The database design itself is also a document, making it easy to replicate. The design defines views and attachments.

CouchDB Database Interaction

Relax!

The CouchDB motto is Relax! All interactions are REST. A management console is provided, named Futon. Here are some examples of the RESTful database access, appending various paths to the base URL. To see the management console, append _utils like this:

```
http://127.0.0.1:5984/_utils
```

Access all database documents by appending the database name and _all_docs to the base URL:

```
http://127.0.0.1:5984/roomedit/_all_docs
```

To retrieve the full document data, not just id and revision, specify the property include_docs=true:

```
http://127.0.0.1:5984/roomedit/_all_docs?include_docs=true
```

BIM Model

How do we represent the simplified BIM in the CouchDB database? Well, here are the BIM objects of interest, to start with:

- Model – a Revit project file
- Level
- Room
- FamilyInstance – represents furniture or equipment
- FamilySymbol – defines geometry

BIM Object Relationships and Graphics

How do they relate to each other, and how are the graphics, geometry and location defined?

We support rooms with multiple boundary loops, so they can contain holes. Furniture and equipment is represented by family instances, which refer to family symbols to define their geometry. For those, we just determine one single exterior boundary loop, to keep things simple.

The furniture and equipment location that can be edited by dragging on the mobile device is a 2D placement defined by a translation and rotation.

The model defines the following containment relationships:

Family instance → room → level → model

Each room defines its own geometry. The furniture and equipment reuses geometry defined by a family symbol:

Family instance → symbol

NoSQL Database Structure

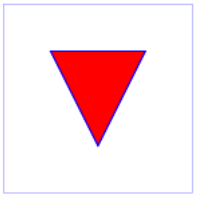
The BIM objects listed above a represented by the following database classes, or document types:

- DbObj base class
- DbModel
- DbLevel
- DbRoom
- DbFurniture
- DbSymbol

Database Object Relationships

To support the containment and geometrical relationships discussed above, the database objects need to maintain the following links to each other:

- DbFurniture.symbolId → DbSymbol
- DbFurniture.roomId → DbRoom
- DbRoom.levelId → DbLevel
- DbLevel.modelId → DbModel



Database Object Graphics and Placement

So far, we discussed how to represent the required information in the cloud database. For displaying and editing the simplified 2D model on a mobile device, we need to render it somehow. All graphics used here are represented by SVG path element data. As an example of that, the figure on the right can be defined using a rectangle and a path element like this:

```
<svg width="4cm" height="4cm" viewBox="0 0 400 400"
    xmlns="http://www.w3.org/2000/svg" version="1.1">
  <rect x="1" y="1" width="398" height="398"
      fill="none" stroke="blue" />
  <path d="M 100 100 L 300 100 L 200 300 z"
      fill="red" stroke="blue" stroke-width="3" />
</svg>
```

The path element geometry is defined by its stroke, the d attribute that specifies a closed 2D polygon. It contains a string listing pairs of X and Y coordinates representing 2D points and the letters M, L and Z that stand for the move, line-to and close commands.

JSON Symbol Database Document

As said, all database documents are defined completely in JSON format. Here is an example of a family symbol document and the data it holds to define its geometry in SVG format, as a path element stroke attribute. As said, the CouchDB document id is populated from the Revit element unique id:

```
{
  "_id": "11cc6e52-519e-49b2-9813-c9561b59a1fd-0005f5fc",
  "_rev": "1-d575ca095533db4ccbed9f7ab2607a12",
  "loop": "M-191 922 L190 922 216 862 -191 859 -216 862 -216 919Z",
  "type": "symbol",
  "description": "FamilySymbol Furniture <390652 Table ronde a chaises>",
  "name": "Table ronde avec chaises - 01"
}
```

JSON Instance Database Document

Here is another example, a furniture document representing a family instance. It defines the relationship to the containing room, to the family symbol specifying its geometry, and its placement, or transform, consisting of the translation and rotation components:

```
{
  "_id": "11cc6e52-519e-49b2-9813-c9561b59a1fd-0005f65b",
  "_rev": "1-c1b4fc969181267b55dab4c6857fc5d7",
  "roomId": "cbe571b0-0593-4350-a8e6-abf3c9239325-00061210",
  "symbolId": "11cc6e52-519e-49b2-9813-c9561b59a1fd-0005fe6d",
  "transform": "R-90T-10429,1020",
  "type": "furniture",
  "description": "FamilyInstance Furniture <390747 Canapé 3 places>",
  "name": "Canapé 3 places"
}
```

CouchDB Views

Once we have populated the database, i.e. put the information into it, how do we get it out again, i.e. create a view?

The primary tool used for querying and reporting on CouchDB databases are views.[1] They can be defined in JavaScript, as we do here, although there are other query servers available.

To create a view, the appropriate function is defined and saved in a *design document*. The IDs of a design document begins with *_design/* and has a special *views* attribute that has a *map* member and an optional *reduce* member to hold the view functions. All views in a design document are indexed whenever any of them is queried.

The map function produces a list of key-value pairs. The reduce produces an accumulation, for which we have no use of in this application.

A design document that defines *all*, *by_lastname*, and *total_purchases* views might look like this:

```
{
  "_id":"_design/company",
  "_rev":"12345",
  "language": "javascript",
  "views":
  {
    "all": {
      "map": "function(doc)
        { if (doc.Type == 'customer') emit(null, doc) }"
    },
    "by_lastname": {
      "map": "function(doc)
        { if (doc.Type == 'customer') emit(doc.LastName, doc) }"
    },
    "total_purchases": {
      "map": "function(doc)
        { if (doc.Type == 'purchase')  emit(doc.Customer, doc.Amount) }",
      "reduce": "function(keys, values)
        { return sum(values) }"
    }
  }
```

[1] http://wiki.apache.org/couchdb/HTTP_view_API - Basics

```
}
```

The *language* property of the design document tells CouchDB the language of the functions inside it, e.g. map, reduce, validate, show, list, etc. Based on this it selects the appropriate view server specified in the couch.ini file. The default is JavaScript, so we can omit this property in our case.

To change a view, you simply alter the design document it is stored in and save it as a new revision. This causes all modified views in that design document to be rebuilt on the next access.

Once a design document has been saved to the database, the *all* view can be retrieved at the URL:

- http://localhost:5984/database/_design/company/_view/all

A number of options can be specified when querying a view: *key, keys, startkey, startkey_docid, endkey, endkey_docid, limit, stale, descending, skip, group, group_level, reduce, include_docs, inclusive_end, update_seq*.[2] In our case, we will only be making use of *key* and *keys*.

Room Editor Views

Since all database object retrieval is implemented using views, we need to define appropriate views to retrieve the objects of interest to the room editor, i.e. the models, levels, rooms, furniture and symbols.

Here is an example of a simple un-keyed view returning all rooms in the database. It retrieves all documents of the specified type and emits key-value pairs containing the entire document as a key with no associated value:

```
rooms = {
  map: function (doc) {
    if( 'room' == doc.type ) {
      emit(doc, null);
    }
  }
};
```

Since the database may contain a large number of models, each containing a large number of levels, each containing a large number of rooms, each containing a large number of family instances, we obviously also need to limit the searches for the lower-level items to retrieve only the ones contained in the next-higher-level container.

This is achieved using the following views taking a key argument to specify the container:

- map_room_to_furniture
- map_level_to_room
- map_model_to_level

The views used to retrieve specific documents belonging to a given container emit key-value pairs specifying the container as the key and the document as the value:

```
map_level_to_room = {
  map: function (doc) {
    if( 'room' == doc.type ) {
      emit(doc.levelId, doc);
    }
  }
};
```

When such a keyed view is queried, the caller can specify the keys of interest, and only the corresponding results will be returned.

[2] http://wiki.apache.org/couchdb/HTTP_view_API - Querying_Options

Accessing CouchDB Documents and Views

Here are some sample URLs to access various room editor documents and views showing the REST API representation:

- Specific document
  http://127.0.0.1:5984/roomedit/11cc6e52-519e-49b2-9813-c9561b59a1fd-0005f5fc
- Specific view
  http://127.0.0.1:5984/roomedit/_design/roomedit/_view/models
- Specific key in view
  http://127.0.0.1:5984/roomedit/_design/roomedit/_view/map_level_to_room?key="933c4a06-93b8-11d3-80f8-00c04f8efc32-0000001e"

Entire CouchDB Application Definition

To recapitulate, CouchDB manages databases, the database contains documents, and the database definition is contained in special design documents.

So, if we want to create a new database, how can we create the required design documents to specify its behaviour?

One way would be to add them manually one by one, e.g. using the interactive Futon interface.

Another method would be to define the design documents using regular files and upload them into database design documents using a dedicated tool.

Numerous such tools exist, and the one I ended up using is Kanso, http://kan.so.

The entire Kanso definition of my room editor database consists of the following files:

- roomedit/data/room_model_9.json -- sample JSON data
- roomedit/index.html -- main database interaction
- roomedit/kanso.json -- database specification
- roomedit/lib/app.js -- application
- roomedit/lib/views.js -- view definitions
- roomedit/raphael-min-jt.js -- SVG library

The top-level database specification is stored in kanso.json:

```
kanso.json
{
  "name": "roomedit",
  "attachments": ["index.html",
  "raphael-min-jt.js"],
  "modules": ["lib"],
  "load": "lib/app",
  "dependencies": {
    "attachments": null,
    "modules": null,
    "properties": null,
    "db": null,
    "jquery": null
  }
}
```

Index.html

The entire room editor user interface defining all user interaction with it is implemented in the one and only index.html, which defines:

- HTML scaffolding
- JavaScript query section
- Raphaël SVG generation and interaction
- RESTful database updates

It is opened in a browser and supports appending the following arguments to the URL:

- No argument: list all models
- Model id: list all levels in model
- Level id: list all rooms on level
- Room id: display graphical editor and enable database updates

Minimal Predefined HTML Scaffolding

The following HTML scaffolding is provided in index.html and populated in various ways from the database depending on the input arguments provided:

```
<h1>Room Editor</h1>

<div id="content"></div>

<ul id="navigatorlist"></ul>

<div id="editor"></div>

<p id="current_furniture"></p>

<script type="text/javascript" src="modules.js"></script>
<script type="text/javascript" src="raphael-min-jt.js"></script>
```

These HTML nodes are populated using JavaScript, adding additional HTML and SVG sub-nodes, with help from the referenced jquery, raphael and db libraries, depending on the results of the CouchDB queries.

List all Models

The first and simplest call to index.html is providing no input parameter. In this case, the top-level navigation listing all models is presented. They are retrieved from the database through the *models* view, and no key is specified. All models are returned and used to populate the HTML *navigator* list node like this:

```
db.getView('roomedit', 'models',
  function (err, data) {
    if (err) {
      return alert(err);
    }
    var n = data.rows.length;
    for (var i = 0; i < n; ++i) {
      var doc = data.rows[i].key;
      var s = url + '?modelid=' + doc._id;
      $('<li/>').append($('<a>')
        .attr('href',s)
        .text(doc.name))
      .appendTo('#navigatorlist');
    }
    var p = $('<p/>').appendTo('#content');
    p.append( $('<a/>').text('Home').attr('href',url) );
    p.append( document.createTextNode( three_spaces ) );
```

```
    p.append( $('<a/>').text('Back').attr('href',url) );

    $('<p/>')
      .text( 'Please select a model in the list below.' )
      .appendTo('#content');

    var prompt = n.toString() + ' model'
      + pluralSuffix( n ) + dotOrColon( n );

    $('<p/>').text( prompt ).appendTo('#content');
  }
);
```

List all Levels in Selected Model

The next navigation level is presented when a specific model has been selected. The model id is added as an input parameter *modelId* to the URL. It is used both to retrieve the model document itself and used to define a key for querying the *map_model_to_level* CouchDB view to retrieve the levels contained in it, again populating the navigator list.

List all Rooms on Selected Level

On selecting a specific level its id is added as an input parameter *levelId* to the URL. The next level of navigation retrieves the level document, uses its model id to retrieve the containing model document data, and queries the *map_level_to_room* view to retrieve and populate the navigator list with all its rooms.

Display and Edit a Selected Room

Once the user has finally navigated to a room, the interesting stuff can begin. Its id is added as an input parameter *roomId* to the URL. Based on the room id, the room document is retrieved, providing the level id. The level document is retrieved, providing the model id. The model document is retrieved, and the relevant data of these nested elements is displayed.

The graphical editor displays the room polygon and the furniture and equipment family instances it contains. These are retrieved by querying the *map_room_to_furniture* view with the room id as a key, returning the family instance documents. Each of those defines the instance placement and references a symbol object defining its geometry.

- Get room document itself → level id
- Get level document → model id
- Get model document
- Display current selection
- View *map_room_to_furniture* with key room id
- Retrieve family instances → symbol ids
- View 'symbols' with list of symbol keys
- Populate and display SVG editor

Database Query Callback Function

As you can see from the db.getView code sample above, none of the JavaScript database query functions return any direct results. They all take the input arguments required to execute the query, plus a callback function that is invoked to process the results once they become available.

All except the first and simplest level of navigation presented by the room editor require more than one database query to retrieve the required data. At the most complex level, to graphically display and edit the room and furniture, three documents need to be retrieved and two views queried: the room, level and

model documents, the view of all family instances in the room, and the family symbols of all the instances encountered.

One way to ensure that all requests have completed before the final result is generated is to nest each additional request inside the callback function invoked on the completion of the previous one.

You could say that we have rerouted the asynchronous database interaction into a synchronous interface.

SVG Room Editor

Once all the data required to display the room and the family instances it contains has been retrieved from the cloud database, the display is constructed graphically using SVG with the help of the jquery and Raphaël JavaScript libraries.

The main work is accomplished by the JavaScript function 'raphael' taking the room and furniture document input arguments, where each furniture object has been populated with the SVG path retrieved from its symbol. It determine the required size and aspect ratio for the device and browser window, sets up a Raphaël canvas, also known as paper, draws the room boundary, attaches the tooltip handling event handlers reacting to mouse-over and mouse-out, places the furniture SVG graphics, attaches identification tags, drag and tooltip event handlers to those as well like this:

```
function raphael( roomdoc, furniture ) {

  // canvas

  // min-x=left min-y=top width height
  var v = get_floats_from_string( roomdoc.viewBox, " " );
  var w = 600;
  var h = Math.round(((v[3] * w) / v[2]) + 0.5);
  paper = Raphael("editor", w, h);
  xmin = v[0];
  ymin = v[1];
  xscale = v[2] / w;
  yscale = v[3] / h;

  paper.setViewBox( xmin, ymin, v[2], v[3], false );

  // tooltip support

  var att = { fill: "white", stroke: "black",
    "stroke-width": 0.1 * xscale, "opacity" : 0.85 };

  tooltip_bg = paper
    .rect( 0, 0, 55 * xscale, 17 * yscale, 2 * xscale )
    .attr(att).hide();

  tooltip = paper
    .text( 0, 0, '' ).attr("font-size", 12 * yscale )
    .hide();

  // room - outer loop anti-clockwise, inner clockwise

  att = { fill: "gray", "fill-opacity": "0.2",
    stroke: "black", "stroke-width": 0.1 * xscale };

  var room = paper
    .path( roomdoc.loops )
    .attr( att )
```

```
    .data( "doc", roomdoc );

  $(room.node)
    //.hover( tooltip_show, tooltip_hide )
    .mousemove( tooltip_show )
    .mouseout( tooltip_hide );

  // furniture

  att = { fill: "green", stroke: "black",
    "stroke-width": 0.1 * xscale, class: "pointer" };

  for( var i=0; i < furniture.length; ++i ) {
    var f = furniture[i];
    console.log(f.transform);
    var trxy = f.transform.slice(1).split(/[,RT]/)
    var angle = parseFloat(trxy[0]);
    console.log(trxy[0]+","+trxy[1]+","+trxy[2]);

    var item = paper.path( f.loop )
      .transform("T"+trxy[1]+","+trxy[2])
      .rotate( angle )
      .click(jonclick2)
      .data("angle", angle )
      .data("jid", f._id)
      .data("doc", f)
      .attr(att)
      .drag(move, dragger, up);

    $(item.node)
      //.hover( tooltip_show, tooltip_hide )
      .mousemove( tooltip_show )
      .mouseout( tooltip_hide );
  }
};
```

Challenges

Many!

This entire topic was entirely new to me when I started.

I was surprised to discover the existence of NoSQL databases, for instance, and it took some reading and experimenting to get my mind around the use of CouchDB, views, CouchApp and Kanso.

I already knew enough about JavaScript and SVG, but getting the event handling to work as expected on all different browser and mobile device types was another big challenge.

Towards the end, I had to figure out the mapping between the Revit and SVG canvas Y axis, which needs flipping for display and restoring to update the BIM to the modified family instance transformation.

It took a while to understand how to handle the nested database callback functions.

Retrieving database changes via DreamSeat was relatively straightforward. An open issue is whether to subscribe to continuous changes. At the moment, due to the design of the Idling event, it will probably add no advantage.

Even though this is a simple learning sample, I can immediately envision a huge number of extremely exciting possibilities and infinite possible real-life applications.

Conclusion

NoSQL is great!

I am extremely impressed with the simplicity and power of the NoSQL database concept and implementations.

The use of open source for this project proved a perfect choice.

REST and JSON are powerful and simple to work with.

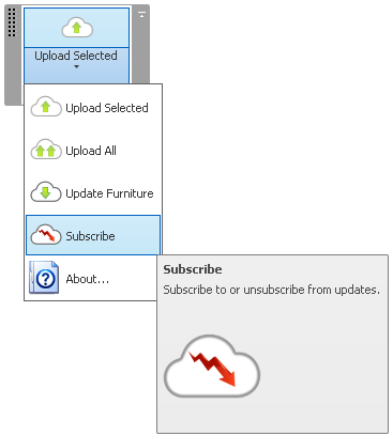
The server-side scripting, which initially seemed like an additional complication, ended up contributing significantly to the simplicity and minimalism of the final solution, and I was extremely happy in the end to be able to avoid all programming on the mobile device itself.

KISS!

The Revit Add-In

The Revit add-in RoomEditorApp implements a number of interesting features that are well documented on The Building Coder blog:

- External application to present the user interface[3]
- Idling challenges and benchmarking timer[4]
- Determine rooms[5], equipment and furniture boundary loops[6]
- Transform Revit graphics to SVG path descriptions
- Temporary graphical display of the boundary loops for debugging purposes[7]
- Represent the database model
- Interact with the cloud database
- Utilities for unit conversion, formatting, messages, folder browser, etc.



Most of these are not directly related to any cloud or mobile issues. Let us first take a look at the one aspect that **is** in fact cloud or mobile related:

How does the Revit add-in access CouchDB?

- Upload model data
- Retrieve database changes
- Subscribe to changes

This is all achieved using the DreamSeat library, available from vdaron's GitHub repository https://github.com/vdaron/DreamSeat. It works well and is easy to use. Many thanks to vdaron for his good job!

[3] http://thebuildingcoder.typepad.com/blog/2013/11/roomeditorapp-architecture-and-external-application.html
[4] http://thebuildingcoder.typepad.com/blog/2013/11/roomeditorapp-idling-and-benchmarking-timer.html
[5] http://thebuildingcoder.typepad.com/blog/2013/03/revit-2014-api-and-room-plan-view-boundary-polygon-loops.html#3
[6] http://thebuildingcoder.typepad.com/blog/2013/04/extrusion-analyser-and-plan-view-boundaries.html
[7] http://thebuildingcoder.typepad.com/blog/2013/04/geosnoop-net-boundary-curve-loop-visualisation.html

Documentation and Materials

The Building Coder

Most of the content of this document has been published on The Building Coder Revit API blog thebuildingcoder.typepad.com. Three dedicated categories there document the following three main aspects of this application: Cloud, Desktop and Mobile.

GitHub Repositories

All of the source code for this application is contained in the roomedit CouchDB database definition and the RoomEditorApp Revit add-in. Both are these are available in their entirety from my GitHub repositories at https://github.com/jeremytammik. A sample cloud installation of the room editor is up and running at jt.iriscouch.com/roomedit/_design/roomedit/index.htm.

Learning More

- Revit Developer Centre: DevTV and My First Plugin Introductions, SDK, API Help, Samples
  http://www.autodesk.com/developrevit
- Developer Guide and Online Help
  http://www.autodesk.com/revitapi-wikihelp
- Revit API Trainings, Webcasts and Archives
  http://www.autodesk.com/apitraining > Revit API
- Discussion Group
  http://discussion.autodesk.com > Revit Architecture > Revit API
- API Training Classes
  http://www.autodesk.com/apitraining
- ADN AEC DevBlog
  http://adndevblog.typepad.com/aec
- The Building Coder, Jeremy Tammik's Revit API Blog
  http://thebuildingcoder.typepad.com
- ADN, The Autodesk Developer Network
  http://www.autodesk.com/joinadn and http://www.autodesk.com/adnopen
- DevHelp Online for ADN members
  http://adn.autodesk.com