

Dev | Blog

First Year



HTTP Servlets

Java

AngularJS

Table of Contents

Intro	0
A Saga of Servlets	1
A Quick Note on JARs	1.1
Servlet Intro and Flavors	1.2
Basic Servlet Implementation	1.3
Servlet Handling of Requests	1.4
Interlude and Announcement	1.5
Servlet Handling Data, A Round House Kick	1.6
Building a Front-End pt.1 Plus a Quick Review	1.7
Building a Front-End pt.2: An App with AngularJS	1.8
Series Review	1.9
Supporting	2
RESTful API Consumption on the Server (Java)	2.1
Server REST Consumption with Authentication	2.2
Custom JSON with Java-ized XAgent	2.3
Application Logic	3
Unraveling the M-V-C Mysteries	3.1
REST is Best	3.2
More on HTTP and AJAX Requests	3.3
What is a Single Page Application(SPA)?	3.4
Related	4
"Replacing" an XAgent	4.1
Alternate Front-End Development: Mocking Your Domino Back-end	4.2
Java	5
When You Need a Comparator	5.1
Building Java Objects from JSON	5.2
JavaScript	6
Consistent Multi-Value Formatting in SSJS for Domino/XPages	6.1
Fixing an Older Version of Dojo (1.6.1)	6.2
An Dojo Implementation of the Calendar Picker Script	6.3
Other	7
A Brief Introduction to Nginx	7.1
Self-Hosting SCM Server	7.2
Domino/XPages Design Element Syntax Highlighting on Redmine	7.3
Glossary	

Dev|Blog: The First Year

build passing

Greenkeeper not found

A book format of the first year of my blog on my shared trials, tribulations, and triumphs in the world of Lotus/IBM Domino/XPages and more.

The blog from which this content was generated can be found at edm00se.io, this book is available as a [pdf](#), [epub](#), [mobi](#), and as a generated static site, hosted on [GitHub Pages](#), at edm00se.github.io/dev-blog-book.



Intro

This is a demo application I created as the result of an interesting set of circumstances surrounding my 2015 [IBM ConnectED](#) session/chalk talk. Its purpose is to provide a place to illustrate the many topics related in application layering, design, and development practices that I've blogged about. This app, my App of Ice and Fire, so named after [George R. R. Martin's A Song of Ice and Fire](#), contains a couple different data sets which are topically pulled from public sources regarding subjects from the book series.

Structure

The application is in two layers, which has numerous advantages.

Back-End

The back-end to the applicaiton is a [Domino/XPages](#) NSF, which has traditional [Notes](#) documents with Forms and Views. These are exposed via some [Java](#) HTTPServlets (specifically *DesignerFacesServlets* which gives us *FacesContext*, ergo authenticated [Notes](#) Session, access), which interact with a proper [Java](#) data model for each resource type. The servlets provide a [RESTful API](#) with *application/json* content, using Google's [GSON](#) library to both generate the [JSON](#) and reflect the received data from the client app (via POST and PUT requests) into their respective data model instances.

Front-End

The front-end to this application is an [AngularJS](#) application, using ui-router. The layout is a fairly standard Bootstrap 3 layout with Font Awesome added. In nearly every way, the front-end application has no direct [Domino](#) dependencies, other than the RESTful APIs, which can be easily mocked for front-end development purposes via freely available (and open source) tools, such as [json-server](#); covered in a blog post called "[alternative front-end development](#)".

- [Preface](#)
- [What?](#)
- [Why?](#)
- [See It In Action](#)
- [Caveat](#)

[Edit] [In the comments](#), [Sven Petri pointed out](#) the need to have the JAR in the same relative path in the Designer environment conducting any build of the NSF. This is absolutely worth noting, though my excitement on this topic was driven by the lack of need to edit the `java.policy` file. Ultimately, everyone ought to communicate with their customers and/or administrators as to the external dependencies, to avoid any build issues by customer admins or non-developers. Basically, make sure people know to drop a copy of the JARs from the server in their local `/jvm/lib/ext/` path. [Edit]

Preface

Either I just didn't know that this was a viable option or we've all been living in the dark for too long. My suspicion is the former, but what follows is a quick run down of my preferred approach for using the `com.google.gson` library (or any JAR), server-wide (without OSGi deployment). TLDR; drop it in `/jvm/lib/ext/` and restart your [Domino](#) server (don't forget to do it with your local/dev environment as well).

What?

While preparing for my impending blog series on servlets, I've been hammering out a couple of details regarding external dependencies (aka- JAR files). The short story is that I assumed things had to be a certain way (including the `java.policy` edit for granting all permissions), but that wasn't the case. If you want to read the full circle of comments, [go check them out](#).

Why?

It seems that setting up what I regard as server elements, even these add-on ones, is something I don't do every day. Any developer can see quickly that re-importing the same JAR file you use across your application instances can become quite tedious, quickly. But it would seem that there is a better way of doing things than just importing your JAR to each NSF and needing to add a line on the server (in `/jvm/lib/security/java.policy`) of

```
grant { permission java.security.AllPermission; }
```

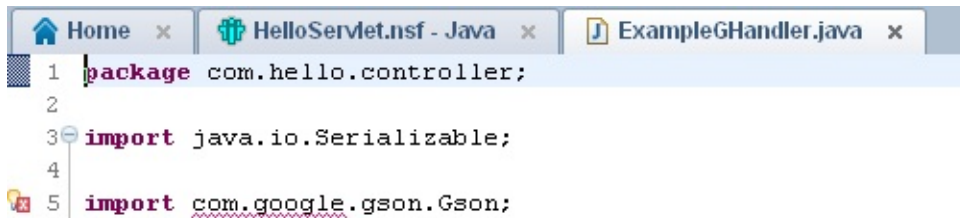
To rule out what I have going in my primarily development environment (something that doesn't come up for me as a staff employee of an [IBM](#) customer, as my environment doesn't change, unless I add a picture of my kid to my desk), I created a fresh install of [Notes/Domino](#) Designer. I took a look at the `/jvm/lib/security/java.policy` file and noticed something that works to our advantage as developers.

```
15 // Standard extensions get all permissions by default
16
17 grant codeBase "file:${java.home}/lib/ext/*" {
18     permission java.security.AllPermission;
19 };
```

So, without the need to edit the `java.policy` file, this makes things a much easier sell to your admins (even though I recommend just buying them their beverage of choice :beer:), as adding an industry accepted library to your server stack has a whole different tone than potentially scaring them with words like "grant" and "`java.security.AllPermission`". One still needs access to the file system, so it may not do some people a lot of good; which is why, going forward with this series, I'll be making the effort to give every [GSON](#) specific task I perform a fair shake with the equivalent using the `com.ibm.commons.util.io.json` package.

See It In Action

Here's my import from my series demo code imported into my fresh DDE install via my [Git](#) repo. As expected, without any JAR to find, it's going to fail.

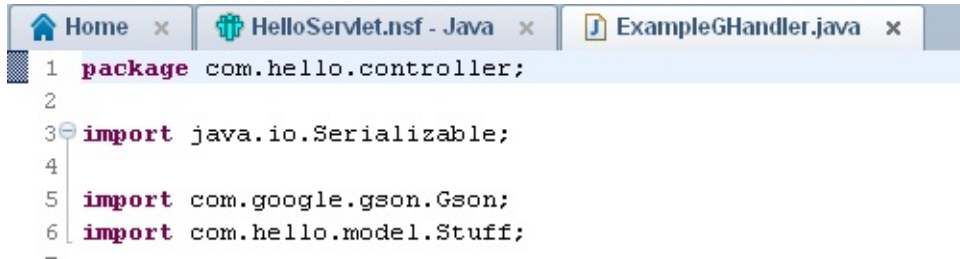


```

1 package com.hello.controller;
2
3 import java.io.Serializable;
4
5 import com.google.gson.Gson;

```

Having shut down Designer and placing the *com.google.gson* JAR into the */jvm/lib/ext/* path and then starting it back up again, you can see that it's now resolved. All without touching the *java.policy* file.



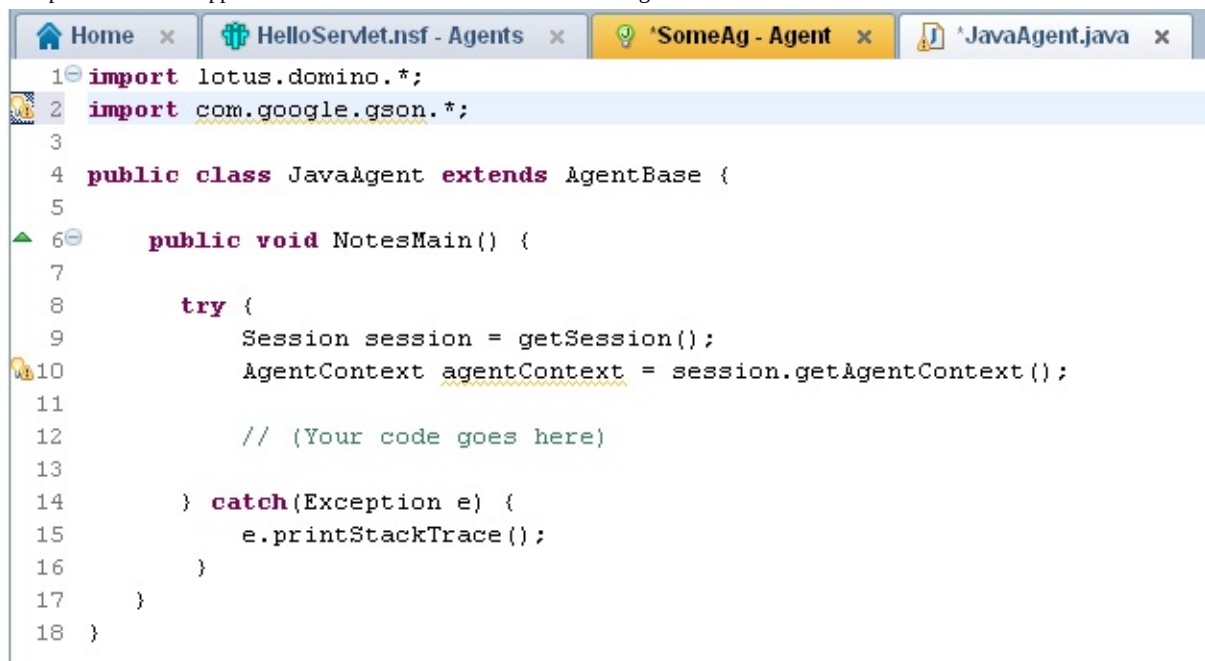
```

1 package com.hello.controller;
2
3 import java.io.Serializable;
4
5 import com.google.gson.Gson;
6 import com.hello.model.Stuff;

```

Added Benefit

The plus side to this approach is that it's now also available in *Java* agents.



```

1 import lotus.domino.*;
2 import com.google.gson.*;
3
4 public class JavaAgent extends AgentBase {
5
6     public void NotesMain() {
7
8         try {
9             Session session = getSession();
10            AgentContext agentContext = session.getAgentContext();
11
12            // (Your code goes here)
13
14        } catch (Exception e) {
15            e.printStackTrace();
16        }
17    }
18 }

```

Caveat

As is inevitable with such things, there is a caveat to the use of the */jvm/lib/ext/* path for JAR inclusion, primarily revolving around any libraries which are already a part of *Domino*.

[@edm00se](#) IIRC ext/lib is in there since 8.0. Do watch out if you put versions of libraries shipping with *domino* in there, like *Abdera*.

— Martin Leyrer (@leyrer) February 9, 2015

Ultimately, I'm aiming to get into OSGi plugins for a first go by including my hit list of usual JAR files, so I can import them on a per-project basis. For example, if I'm building out a RESTful end point with *GSON*, I'm also probably using a couple Apache Commons libraries. It makes sense to package accordingly. One day, I'll have all the cool toys.

- [Preface](#)
- [A Series on Servlets](#)
- [What Is A Servlet?](#)
 - [What Does That Mean for Me?](#)
 - [But I Heard XAgents Are Bad?](#)
- [Flavors of Servlets](#)
 - [HttpServlet](#)
 - [DesignerFacesServlet](#)
 - [AbstractXSPServlet](#)

Preface

This post is essentially the first two parts of the several I've already identified I'll be blogging about. I kept waffling between wanting the first three in one post, which would have been huge, and just one, which would have been short. Here's what I came up with. You can also keep track of this and the other parts of the series on the [servlet series](#) page.

A Series on Servlets

This is the first post in [a short series on servlets](#) I'll be posting over the next month or two. It's going to take some time to go through, but I hope you'll stick with me through to the end. I was originally going to speed through the whole process and give a lengthy one-shot, one-kill post on the topic; but servlets, while simple in nature, can be complex depending on your implementation. I've learned a couple things since I started assembling and hopefully this will be useful to some out there.

What Is A Servlet?

In case you haven't run into the term before, [a servlet is](#), semantically, a portmanteau of the words server and applet. This stems from the origins of [Java](#) dating to the early years of many of the conventions that we take for granted now, in terms of web technology. Functionally speaking, a servlet is a registered Class which connects via URI to provide or consume *something*; this can be data ([XML](#), [JSON](#), others), messages (e.g.- plain text), or more.

What Does That Mean for Me?

Many XPage developers have found great power in the use of [XAgents](#). They can be powerful and flexible to meet our needs. The primary reason for this is the ability to hijack the response output writer, and return our own data, in whatever (content-type) format we specify. This makes things seamless to the user; e.g.- `buildMyContacts.xsp` can yield a `myContacts.csv` file, downloaded as an attachment, with the current/live data. Servlets let us do essentially the same thing; they provide an end point (instead of `buildMyContacts.xsp` we may have `/api/mycontacts`), which lets us return data in the format we want (setting the response content-type, accepting certain types, etc.), in a generally seamless fashion (current data).

But I Heard XAgents Are Bad?

One of the minor themes at ConnectED for those of us in the developer crowd was that XAgents bring along some unnecessary baggage for a simple data response; specifically [the JSF life cycle](#). This isn't exactly "bad" so much as just a set of unnecessary executions performed when the response could just be built and sent after requested. The [JSF life cycle](#) is there for us to assist in building out the tags for an HTML page, provide the data bindings, and set and handle the state of the page and its bound data elements. With a servlet, you choose the response format, making it focus only on hooking in to your code (via [`javax.servlet.http.HttpServlet`](#)).

Flavors of Servlets

So far as I can tell, there are essentially three ways of creating a servlet on a [Domino](#) server. I should mention that I'm focusing on application servlets, with no server-level deployment. The flavors are boiled down to (vanilla) `HttpServlet`, `DesignerFacesServlet`, and an abstracted servlet, which uses a wrapper to handle some of the frequent tedium (why we abstract any code). I'll try to identify why you might use each, with a brief description.

Note: I'm not covering implementation in this post, that will be covered in the next post. Each of the "flavors" outlined below share two steps in the implementation, so I'm attempting to differentiate each now, before cramming them all into an application together.

HttpServlet

Probably the easiest to implement, to write the servlet, one must write a class which extends *HttpServlet*. This class can contain override methods for init and destroy and exposes the methods (VERBs such as GET, POST, PUT, and DELETE) available via *do** methods (*doGet*, *doPost*, *doPut*, and *doDelete*). A servlet needs to provide its response in either a response writer or output stream. Have a look, this is a fully functioning, albeit simple, servlet.

```
package com.hello.servlets;

import java.io.PrintWriter;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Example Servlet, implemented as 'vanilla'
 * HttpServlet. Many non-Domino Java Servlets
 * implement HttpServlet.
 *
 * @author Eric McCormick, @edm00se
 */
public class ExampleHttpServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    @Override
    public void init () {

    }

    @Override
    public void destroy () {

    }

    @Override
    protected void doGet (HttpServletRequest req, HttpServletResponse res) {
        HttpServletRequest _req = req;
        HttpServletResponse _res = res;
        PrintWriter out = null;
        try{
            _res.setContentType("text/plain");
            out = _res.getWriter();
            out.println("Servlet Running");
            out.println("You executed a "+_req.getMethod().toString()+" request.");
        }catch(Exception e){
            if(out!=null){
                out.println("Sorry, an error occurred...");
            }
        }finally{
            out.close();
        }
    }
}
```

Hopefully this seems familiar, even if it's a new format. As you can see, I've only exposed GET as an available method against this servlet. You can provide the others via the *do** methods or, you can specifically lock them down by providing a response code of 405 (method not allowed) with any additional information, error or other descriptive message. It's worth note that the only *do** methods

specifically available are *doGet*, *doPost*, *doPut*, and *doDelete*. To override this and provide, say, *PATCH*, as an available method, you would need to override the behavior offered by the default service method. This comes into play in the next approach, but we'll get there in a second.

An *HttpServlet* is exactly what it claims, but probably isn't the best option for those who want to make use of much of the application, session, or anything which depends on *FacesContext*.

DesignerFacesServlet

So, in order to do anything derived off of *FacesContext*, we'll need a better implementation of our servlet. [Jesse Gallagher has blogged about this very topic](#), big surprise there 😊. Some of the benefits include access to **Scope'd* variables and any managed beans.

```
package com.hello.servlets;

import java.io.IOException;
import java.io.OutputStream;
import java.util.Map;

import javax.faces.context.FacesContext;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.xsp.webapp.DesignerFacesServlet;

/**
 * Example servlet showing the use of access to FacesContext.
 * This was originally blogged about by Jesse Gallagher and
 * is a near duplicate class.
 * src: https://frostillic.us/blog/posts/159496067A27FD3585257A70005E7BC1
 */
public class ExampleServlet extends DesignerFacesServlet {
    private static final long serialVersionUID = 1L;

    @SuppressWarnings("unchecked")
    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
        // Set up handy environment variables
        HttpServletRequest req = (HttpServletRequest)servletRequest;
        HttpServletResponse res = (HttpServletResponse)servletResponse;
        ServletOutputStream out = res.getOutputStream();
        FacesContext facesContext = this.getFacesContext(req, res);

        try {
            res.setContentType("text/plain");

            out.println("start");

            // The sessionScope is available via the ExternalContext. Resolving the variable
            // would work as well
            Map<Object, Object> sessionScope = facesContext.getExternalContext().getSessionMap();
            // ... this is showing how we can get facesContext and *scope variables inside the servlet

            out.println("done");

        } catch (Exception e) {
            // hit an error, dump out whatever is there
            e.printStackTrace(new OutputStream(out));
        } finally {
            out.close();

            // It shouldn't be null if things are going well, but a check never hurt
            if (facesContext != null) {
                //complete the response and release the handle on the FacesContext instance
                facesContext.responseComplete();
            }
        }
    }
}
```

```

        facesContext.release();
    }
}

/**
 * @return FacesContext currentInstance
 */
public static FacesContext getFacesContext() {
    return FacesContext.getCurrentInstance();
}
}

```

You can take note that we're being sure not just to close the output stream, but also to mark the *FacesContext* handle as *responseComplete* and releasing it back into the wild; **do not forget to do this**; this is implied for each and every response operation you provide.

The largest thing to note is, as mentioned above, we're overriding the *service* method. This means that, by default, our accessing of the end point *happens* to be a GET. We need to provide for the response handling based on the request method. It would go something like this:

```

String reqMethod = req.getMethod();

if( reqMethod.equals("GET") ) {
    try {
        out.println("doing something with my GET");
    } catch (Exception e) {
        e.printStackTrace();
        out.println(e.toString());
    } finally {
        facesContext.responseComplete();
        facesContext.release();
        out.close();
    }
} else if( reqMethod.equals("POST") ) {
    try {
        out.println("doing something with my POST");
    } catch (Exception e) {
        e.printStackTrace();
        out.println(e.toString());
    } finally {
        facesContext.responseComplete();
        facesContext.release();
        out.close();
    }
} //...

```

The tedium of always adding a *try/catch* block with *finally* blocks to *close* the output and mark the *FacesContext* as *responseComplete* and performing the *release* is exactly the sort of thing that we as developers like to automate, by abstracting.

AbstractXSPServlet

This is the third flavor; it extends and, ultimately, is a *DesignerFacesServlet*, but by using an abstracted Servlet class, we can automate each of *out.close()*, *facesContext.responseComplete()*, and *facesContext.release()*, with each response, with minimal hassle. Jesse came up with this and I've pulled a copy for my use [directly from his frostillic.us framework](#) for use in my own code base. I recommend you have a read and grab a copy. Essentially, as Jesse shows in his [part 7 of his building an app with the frostillic.us framework](#), all that's needed is to build a class to extend *AbstractXSPServlet* and override the *doService* method, which is wrapper with the necessary *out.close()*, *facesContext.responseComplete()*, and *facesContext.release()*, for each response. This means our servlet class only has to contain what we need it to. Also note that I'm starting to define my response code for each of the non-GET methods.

```
package com.hello.servlets;

import java.util.Enumeration;

import javax.faces.context.FacesContext;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.commons.util.StringUtil;

import frostillicus.xsp.servlet.AbstractXSPServlet;

/**
 * Example Servlet implementing AbstractXSPServlet,
 * from Jesse Gallagher.
 *
 * @author Eric McCormick, @edm00se
 *
 */
public class ExampleAbstractedServlet extends AbstractXSPServlet {

    /* (non-Javadoc)
     * @see frostillicus.xsp.servlet.AbstractXSPServlet#doService(javax.servlet.http.HttpServletRequest, javax.servlet.
     */
    @Override
    protected void doService(HttpServletRequest req, HttpServletResponse res,
        FacesContext facesContext, ServletOutputStream out)
        throws Exception {

        res.setContentType("text/plain");

        String reqMethod = req.getMethod();

        if( reqMethod.equals("GET") ) {

            out.println("doing something with GET");

        } else if( reqMethod.equals("POST") ) {
            res.setStatus(200); // OK
            out.println("doing something with POST");

        } else if( reqMethod.equals("PUT") ) {
            res.setStatus(200); // OK
            out.println("doing something with PUT");

        } else if( reqMethod.equals("DELETE") ) {
            res.setStatus(200); // OK
            out.println("doing something with DELETE");

        } else if( reqMethod.equals("PATCH") ) {
            res.setStatus(200); // OK
            out.println("doing something with PATCH");

        } else {
            res.setStatus(405); // method not allowed
            out.println("what the devil are you trying to do, break the server?");
        }
    }
}
```

Summary

The big take away here is a common base of reference. Going forward, I'll be implementing Jesse's `AbstractXSPServlet`, which looks and acts differently than just a `DesignerFacesServlet` or `HttpServlet`. I recommend you examine what best fits your needs, but I think you should be happy with what it provides.

In the next post, I'll be showing how to implement a servlet via a `ServletFactory` (so we can actually access it) and start framing out some method handling. As always, if anyone has a better way or alternative method, there's the comments section and I welcome response blog posts.

Intro

The [first post](#) covered the first two parts of [this series](#), the basics of what a servlet is and three "flavors" of servlet classes. This post begins with how to implement a servlet so that they're actually accessible via an end point.

ServletFactory

A [factory is, in OOP](#), an object for creating other objects. In order for these servlets to be "registered" with the application to be end point accessible, they need to be provided by a ServletFactory; specifically, one that implements `com.ibm.designer.runtime.domino.adapter.IServletFactory`. This will register an end point via a pair of Maps which match, via a key, the package.class name to the end point name. This makes the servlet accessible via `<your NSF>/xsp/<end-point-name>`.

A Note on IServletFactory

In one of the more counterintuitive things I've run into since starting [Domino/XPages](#) development, the `IServletFactory` package is fully there on the server and usable, but the `lwpd.domino.adapter.jar` needs to be added as an external JAR to the build path in Designer. [Sven Hasselbach](#) has done an excellent job of showing how to do this in [his blog post on the subject](#). Sven's blog is a great read with some very applicable posts on [REST](#) security, including [CORS](#) topics and more; I highly recommend reading his blog, if you don't already.

Marrying the ServletFactory to the Application

Marriage, The Short, Short Version



[Video link](#)

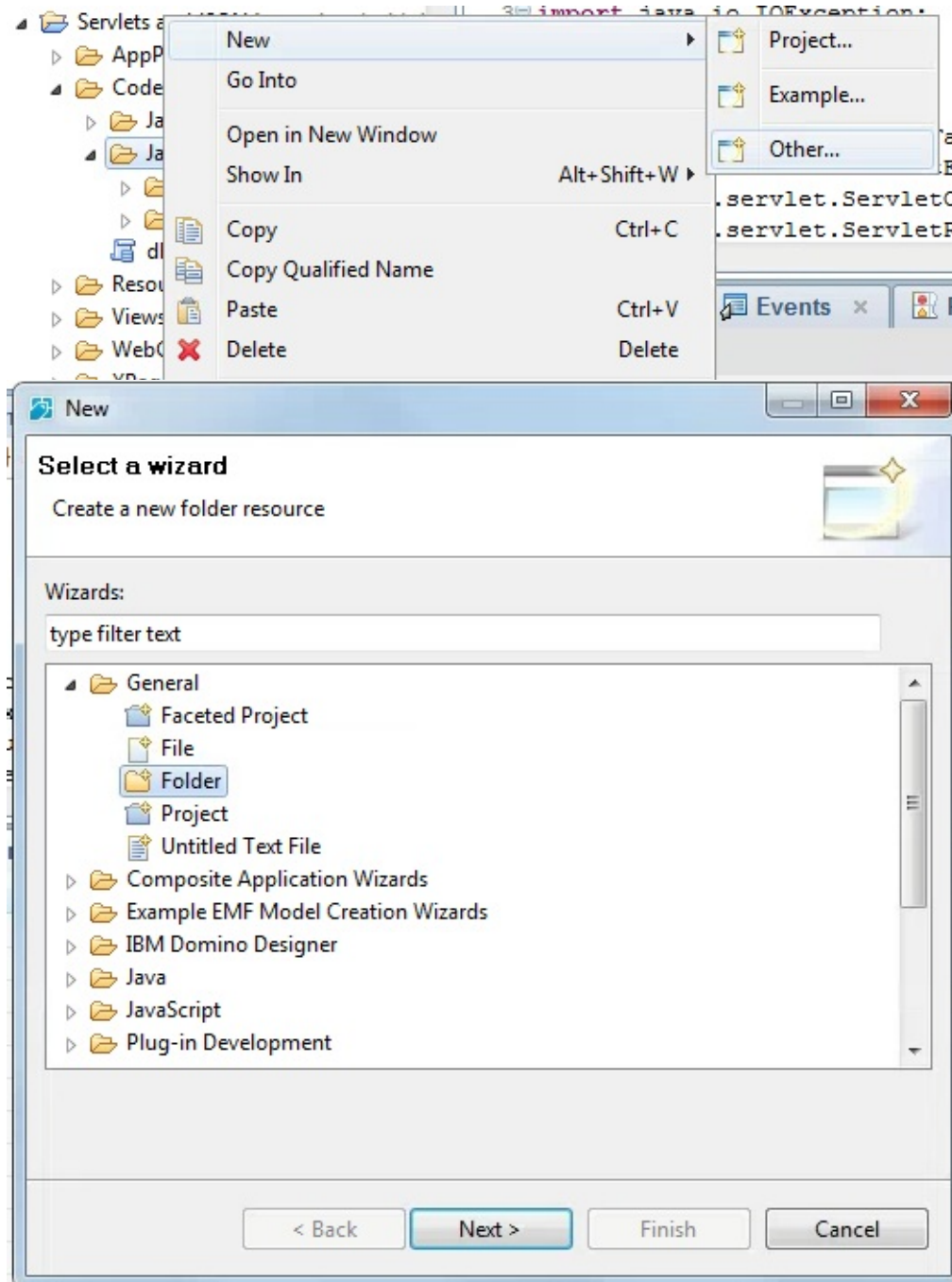
Your `ServletFactory` needs one last step to be registered as usable by your application. Here's the short, short version.

After adding your "external JAR" to your build path, you need to create a file called `com.ibm.xsp.adapter.servletFactory`. Create that file in `/Code/Java/META-INF/services/`; it's easiest if you switch to Package Explorer first. In this file, place the fully qualified package.Class name of your `ServletFactory`. Once your application builds, you're good to go with your keyed servlet names from your `ServletFactory`.

The Less-Short Version

The file we'll create needs to be in the project build path. In [Domino 9](#) (and late 8.5.3 versions, anything in which there came a *Code/Java* and *Code/JARs* design section in the Application perspective), we should focus on the *Code/Java* section. For older versions, the classic location tends to be *WebContent/src*; the bottom line is: **it must be a part of your build path**. To create the file, it's best to switch views to Package Explorer.

You'll need to right-click on your *Code/Java* folder and select New, followed by Other. Select folder and create one called *META-INF/services/* (it'll nest the second one).



Then do the same, selecting file, and call it *com.ibm.xsp.adapter.servletFactory*. In this file, we put a single line for the class which will do the assigning of end points to servlet Classes.

```
com.eric.test.ServletFactory
```

Registering Your Servlet Classes

Now that we finally have our *adapter.servletFactory* file pointing at our *ServletFactory* Class, we can start adding them into the *ServletFactory*. Here's one I prepared earlier.

```
package com.hello.factory;

import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import javax.servlet.ServletException;
import javax.servlet.ServletException;

import com.ibm.designer.runtime.domino.adapter.ComponentModule;
import com.ibm.designer.runtime.domino.adapter.IServletFactory;
import com.ibm.designer.runtime.domino.adapter.ServletMatch;

/**
 * The factory (a provider) implements IServletFactory and creates
 * two maps, for key to package.class and key to servletname matching.
 */
public class ServletFactory implements IServletFactory {
    private static final Map<String, String> servletClasses = new HashMap<String, String>();
    private static final Map<String, String> servletNames = new HashMap<String, String>();
    private ComponentModule module;

    /**
     * init adds the classes and servlet names, mapping to the same key.
     */
    public void init(ComponentModule module) {

        servletClasses.put("exhttpervlet", "com.hello.servlets.ExampleHttpServlet");
        servletNames.put("exhttpervlet", "Example HttpServlet");

        servletClasses.put("exdesignerfaceservlet", "com.hello.servlets.ExampleDesignerFacesServlet");
        servletNames.put("exdesignerfaceservlet", "Example DesignerFaces Servlet");

        servletClasses.put("exabstractervlet", "com.hello.servlets.ExampleAbstractedServlet");
        servletNames.put("exabstractervlet", "Example AbstractXSP Servlet");

        this.module = module;
    }

    /**
     * The ServletMatch matches the path to the correctly identified servlet;
     * by the routed key.
     */
    public ServletMatch getServletMatch(String contextPath, String path)
        throws ServletException {
        try {
            String servletPath = "";
            // iterate the servletNames map
            Iterator<Map.Entry<String, String>> it = servletNames.entrySet().iterator();
            while (it.hasNext()) {
                Map.Entry<String, String> pairs = it.next();
                if (path.contains("/") + pairs.getKey()) {
                    String pathInfo = path;
                    return new ServletMatch(getWidgetServlet(pairs.getKey()),
                        servletPath, pathInfo);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    public Servlet getWidgetServlet(String key) throws ServletException {
        return module.createServlet(servletClasses.get(key), servletNames
            .get(key), null);
    }
}
```

Aside from a bit of voodoo, this should show how we can map our end point names to the class names and proper names, respectively. As you can see, I mapped each of my example servlets (*HttpServlet*, *DesignerFacesServlet*, and *AbstractXSPServlet*) from the last post into respective [endpoint](#) names/keys. The table below shows the resulting mapping of the [endpoint](#) (after the server/path/NSF/).

Servlet Endpoint	Servlet Class	Name
/xsp/exhttpervlet	com.hello.servlets.ExampleHttpServlet	Example HttpServlet
/xsp/exdesignerfaceservlet	com.hello.servlets.ExampleDesignerFacesServlet	Example DesignerFacesServlet
/xsp/exabstractervlet	com.hello.servlets.ExampleAbstractedServlet	Example AbstractXSPServlet

Summary

Now we have a servlet and it's fully registered with the application and accessible via an HTTP [endpoint](#). The next post will get into what we do with these servlets.

Previously, on #ASagaOfServlets

So far in [this series](#) I've covered some [basics](#) on servlets, implementing our methods along with a showing of the "flavors" of servlets, and how to implement these servlets [via a ServletFactory](#). This has been the ground work for everything that comes next.

What to Do With My Servlet?

A servlet can be just about anything. It can receive a payload of data (or just handle a simple network GET request) and process and return almost anything. Ultimately, I want to provide [RESTful API](#) interaction to the front-end side of my application, by:

- abstracting the CRUD operations, in order to
- validate received data changes (not committing changes in case of failure, throwing an error, with messages, to the user)
- and provide a layer of business logic for those interactions, enforcing a set of rules by which all data objects will adhere to (I have previously described this as "loose schema", which is a misnomer, as the entire purpose of a schema is to provide [strict provisioning](#) at the db level; aka- integrity constraints)

Receiving Requests

As I've mentioned above, I've referenced a pattern of `/collection/{:id}` for an [endpoint](#). The basic premise is that you provide the base [endpoint](#) of `.../collection` (usually shown as the plural version, so for a collection of users, it would be `/users`) which at the base level gives the full collection, but when is followed by a route parameter of an ID (for example, a 32-character length hexadecimal value, < like our [Notes Document UNIDs](#)), it will handle requests specific to that document. This effectively makes our servlet at one [endpoint](#) a two-part affair. Here's the approach I'll be using, with strictly [application/json](#) content type.

Formatting and Documentation

Route	Methods Allowed
<code>.../collection</code>	GET
<code>.../collection/{:id}</code>	GET, POST, PUT, DELETE

One major benefit of using a [REST API](#) framework in [Java](#) is the ability to automate your documentation. Documentation is one of the most important aspects of [REST APIs](#) (especially in publicly accessible ones), as if those who will consume them don't know how to interact with them, they won't be worth anything. Usually documentation includes the endpoints, allowed methods, and request and response structure.

Route Matching

We'll be handling multiple routed paths off a single collection [endpoint](#) (the collection and the `collection/{:id}`). The approach I'll be implementing in the route matching will make use of regular expressions. This involves defining a pattern and testing that against the requested path for a match. This will make use of [java.util.regex.Pattern](#) and [java.util.regex.Matcher](#), respectively.

Since we will get a true match with [Matcher.find\(\)](#) from a partial subset, it's important to test in a descending order from the more complex [endpoint](#) down to the simplest; the raw collection. It probably ought to look something like this:

```
// in a method to parse route params

// Accommodate two requests, one for all resources, another for a specific resource
private Pattern regexAllPattern = Pattern.compile("/collection");
// a UNID is 32-char hex, /collection/{:unid}
// UNID ref: http://www-01.ibm.com/support/docview.wss?uid=swg21112556
private Pattern regexIdPattern = Pattern.compile("/collection/([0-9a-fA-F]{32})");

// regex parse pathInfo
Matcher matcher;

// Check for ID case first, since the All pattern would also match
```

```

matcher = regexIdPattern.matcher(pathInfo);
if (matcher.find()) {
    unid = Integer.parseInt(matcher.group(1));
    System.out.println("do something with this document, the id is: "+unid);
}

matcher = regexAllPattern.matcher(pathInfo);
if (matcher.find()) {
    System.out.println("do something with the collection");
}

throw new ServletException("Invalid URI");

```

[EDIT]

It was brought to my attention that route matching is easier via `@` annotations, as one might use via an approach [with Jersey](#). I absolutely agree, but up until now, for this series, I've taken a framework-free approach to generating and implementing servlets. I'll just say that there's a very good reason that such frameworks are out there, and even implementing just the pieces for the `@` annotations could be effort well spent. I fully welcome any response piece on this topic, as I'm not experienced with Jersey (my preference to RegEx matching comes from my [NodeJS/Express API](#) experience).

[/EDIT]

Route Parameters

Now that we've handled the route, it's time to handle any route parameters. Route parameters can be a little confusing, seeing how they look just like another route, but they can also be useful. Strictly speaking, the `/:{id}` is a form of route parameter, but they can also be nested (sequential?) to provide more echelons in a hierarchy. I previously built a single-purpose [NodeJS/Express](#) app that provided an [API](#) to handles requests to our [IBM i](#) for DB2 access; the specifics of that project were to have a three-level deep hierarchy of required information. This is generally a bit deeper than most people will go with route parameters, but it serves to illustrate the concept. My requests look like this:

```
.../api/{:firstLevelParam}/{:secondLevelParam}/{:thirdLevelParam}
```

Route parameters are a way of handling required, hierarchically defining values in a request. They're not the only way and many people don't like them, but I'm a fan (for such hierarchical requirements). To parse them out, we need a handle on the *HttpServletRequest's* *pathInfo* property. We then split it off the `/` character to have a collection, in this case a *List<String>* of all the route path elements. Since the first three are related to the structure of the servlet, we need to start checking at the 4th (3rd position).

```

String reqPath = req.getPathInfo();
out.println("pathInfo: " + reqPath);
List<String> routeParm = Arrays.asList(reqPath.split("/(.*?)"));
if(routeParm.size() > 3) {
    // /nsf/xsp/servletname is the base, so the fourth is the first routeParm
    for( int i=3; i<routeParm.size(); i++ ) {
        out.println( "routeParm: " + routeParm.get(i) );
    }
} else {
    // didn't have any route parameters after the base
    out.println("routeParm: " + "none");
}

```

Query Parameters

Query parameters should be familiar to every [XPages](#) developer. In fact, it's so normal that I'll just mention that you may wish to use a [VariableResolver](#) to populate your `Map<String, String>` as opposed to performing a `split` on the `queryString` of the `HttpServletRequest`.

[Edit]

Thanks to [Jesse Gallagher](#) for catching something here. You can resolve `param`, but it would be better to use something else as it behaves as a `Map<String, String>`, **not** a `Map<String, String[]>`. If you're performing an `HttpServletRequest.getQueryString()`, you will get a `java.lang.String` back, with which contains your results. You can manually pull this apart, but you should really use the [getParameterMap](#) method on your `HttpServletRequest` (the method is inherited from `ServletRequest`) as this *does return* a `Map<String, String[]>`, ensuring you get keyed values for each of multiple values per key. I've used the method elsewhere, I'm not sure what my brain was thinking up above, but I suspect it was a lack of caffeine ☕.

```
Map<String, String[]> param = (Map<String, String[]>) req.getParameterMap();
```

[/Edit]

RESTful APIs?

How is this all [REST](#)? How is it an [API](#)? APIs, for those living under a rock, are an [Application Programming Interface](#); the [Notes/Domino API](#) is how we interact with, reference, and use [Notes/Domino](#) entities. Providing access to invoke calls and operations over a [REST API](#) means that we have logic build into our network calls to our [endpoint](#). [REST](#) is an approach, it has to do with stateless data requests, uses the HTTP VERBs, and is generally descriptive in format. There's not a governing true specification, just some basic rules. If you want to read more on [REST](#) in general, I recommend [this scotch.io post](#).

Next Time

Up next will be a bit more code heavy, as I'll be walking the life cycle of data reception and response handling. It will cover an [endpoint](#) governing a certain data type, provide a collection at the collection level, establish a data model that both our responses will use and the ingested data types will instantiate, and provide CRUD operations against a given document (the data object instance). It will be a fast-paced post, but it should be worth the read.

An Interlude for Servlets

My [series on servlets](#) is in a temporary interlude. Don't worry, I've been working on it, the only problem is the issue I ran into. I was forced to re-evaluate some of the assumptions I had made previously and, to be quite honest, I'm glad I ran into that issue now, as opposed to much later. Suffice it to say, there is much more to come and I am excited to bring my next post, but I won't publish it [again](#) until it's ready.

For those who caught my blog post malfunction while I was ☹️ <http://t.co/qbroT2qpCK>

— Eric McCormick (@edm00se) [February 28, 2015](#)

Issues with Java Security

Over the last week or so, I started to run into an issue I had trouble quantifying. Thankfully, with the help of some intelligent people on twitter (Jesse for one, who seemed to know what I ran into almost immediately) I once again appreciate the fact that our [#XPages](#) developer community is a strong one which is almost always willing to help someone through an issue. It's a credit to this community and one of the reasons that this blog exists. When David Leedy suggested those with anything they could share, ought to, it hit home as I've benefited greatly from the work of others.

The Issue

I shifted my development environment to a new vm and, while doing some actual code work in preparation for the next post in the servlet series, I noticed that my servlets that depended *DesignerFacesServlet* had stopped working. After consulting with Jesse Gallagher, there seems to be some issues with ClassLoader outside of [XPages](#) design elements without this.

So, it seems that while I can keep my JARs in `/jvm/lib/ext/`, it *also* seems that I need to continue to apply the *lady-of-the-evening* approach to my `java.pol(icy)` file to get *FacesContext* access in my servlets; with the following:

```
grant {
    permission java.security.AllPermission;
}
```

My Fix

For now, this means I'm adding the `grant ...AllPermission` line back into my `java.pol(icy)`. I've tried to find ways of keeping that from being necessary, but it seems that if you want *FacesContext* access in your servlet, you should either add that back in, or roll your servlet as an OSGi servlet; an approach I've yet to get into, though [Jason Hook seems to have covered the topic a bit](#). For those concerned about this edit, I would recommend that the requirements of your servlet (should it require *FacesContext*, which is very likely) include having security permissions. If the admin(s) of whoever owns the server can't think of a better way, then they should roll the above line, otherwise they can manage that as-needed. If someone knows of a better way around this, please don't keep it to yourself.

Reminder note: as [Mark Leusink](#) and [John Dalsgaard have pointed out](#), upgrading to 9.0.1 FP3 ~~can~~ will cause loss of custom entries in your `/jvm/lib/security/java.policy` file; this seems to have to do with the [JVM](#) updates. An ideal is to keep your changes in a `/jvm/lib/security/java.pol` file, which gets interpreted with the same syntax as `java.policy`, and is less likely to be overwritten during an upgrade.

An Announcement

Speaking of my blog, I am migrating to a new domain name. Don't worry, all your existing bookmarks and feed links will work, as it will be the same blog hosted on [GitHub Pages](#). My link references will be updating and the Disqus comments migrating. From here on out, I can save myself a few characters here and there:

edm00se.io

A Second Announcement

Recording has begun! I'm hoping that my efforts will be fruitful to people in the ~~years?~~ months to come, but I also know that many people learn better by seeing instead of reading. My blog can get a bit wordy at times, something I try to keep at bay, but in preparation for the end stages of my servlet series, I have recorded the first few pieces of the companion [Notes](#) in 9 episode to-be. It was suggested by some previously and is something I plan on delivering in conjunction with the end of my series.

Until next time, :beers:.

A Fast-Paced, Round-House Kick Tour of Data Interactions

As promised at the end of the last post (in this series), this post will walk through the entire life cycle of data reception and response handling. This is where my [ConnectED demo app-that-never-was](#) comes in, we're going to build part of it. We're going to create an [endpoint](#) which governs the provision of a collection of the houses of note in our fictitious land of Os (it's out west). I'll be providing the `com.westeros.servlets.HouseServlet` class, which is an `AbstractXSPServlet` (previously demonstrated), to my `ServletFactory`.

Note: I'll be sticking to the same, vanilla `Java` approach I've used previously in this series. I'll outright say it though, it'd be great to see how some of the processes involved in the setup can be automated and made easier, be it by `@` annotation or via other frameworks. I fully invite those more experienced in these methods to show us the way.

- [The Endpoint](#)
 - [Request Handling](#)
 - [Collection](#)
 - [Document](#)
- [An Object Model for \(Almost\) Everyone](#)
 - [The HouseModel](#)
- [Receiving Data from POST or PUT](#)
 - [ServletInputStream](#)
 - [FromJson](#)
 - [Using the InputStream Directly](#)
 - [My Class's Interpretation](#)
 - [Provide Response](#)
- [Note: On PUT and DELETE Methods](#)
- [Summary](#)

The Endpoint

The `endpoint` will accept (and return) only `application/json`. Here's the structure it'll take.

Route	Methods Allowed
...NSF/xsp/houses	GET
	POST
-----	-----
...NSF/xsp/houses/{:unid}	GET
	PUT
	DELETE

It's straight forward and follows with the approach I've previously laid out. Do note that to create a new entry, it will be taking a POST against the collection, whereas the individual entry will be accessed via GET to send the existing document, PUT to update partial information, and DELETE to do the obvious.

Request Handling

In order to better process my request data and process for my response, I've segregated my `Collection` and `Record` operations into separate classes; `HouseCollection` and `HouseRecord`, respectively. Here's the down and dirty of my main servlet class:

```
public class HouseServlet extends AbstractXSPServlet {

    @Override
    protected void doService(HttpServletRequest req, HttpServletResponse res,
        FacesContext facesContext,
        ServletOutputStream out) throws Exception {
```

```

// Accommodate two requests, one for all resources, another for a
// specific resource
Pattern regexAllPattern = Pattern.compile("/houses");
Pattern regexIdPattern = Pattern.compile("/houses/([0-9A-Za-z]{32})");

// set content type, cache, and Access-Control headers
res.setContentType("application/json");
res.setHeader("Cache-Control", "no-cache");
res.setHeader("Access-Control-Allow-Origin", "*");
res.setHeader("Access-Control-Allow-Headers", "Origin, X-Requested-With, Content-Type, Accept");

String pathInfo = req.getPathInfo();

// regex parse pathInfo
Matcher matchRecord = regexIdPattern.matcher(pathInfo);
Matcher matchCollection = regexAllPattern.matcher(pathInfo);

// Method invoking the URI
String reqMethod = req.getMethod();

/*
 * Specific Document, by UNID. Allowed are GET,
 * PUT, and DELETE.
 */

if (matchRecord.find()) {
    String unid = matchRecord.group(1); // .group(1);
    if (reqMethod.equals("GET")) {

        // GET the single record
        HouseRecord.doGet(unid, req, res, facesContext, out);

    } else if (reqMethod.equals("PUT")) {

        // PUT to update, in whole or part, a single record
        HouseRecord.doPut(unid, req, res, facesContext, out);

    } else if (reqMethod.equals("DELETE")) {

        // DELETE single record
        HouseRecord.doDelete(unid, req, res, facesContext, out);

    } else {
        // unsupported request method
        HouseRecord.handleUnexpectedVerb(req, res, facesContext, out);
    }
} else if (matchCollection.find()) {
    /*
     * Collection, allows only GET for the View equivalent or POST for
     * creating a new Document
     */

    if (reqMethod.equals("GET")) {

        HouseCollection.doGet(req, res, facesContext, out);

    } else if (reqMethod.equals("POST")) {

        HouseCollection.doPost(req, res, facesContext, out);

    } else {
        // unsupported request method
        HouseCollection.handleUnexpectedVerb(req, res, facesContext, out);
    }
}
}
}

```

Collection

The collection will iterate records and return the **JSON** array of objects representing each house. I'm going to wrap the array as a data element, to give some mild metadata I usually provide, including a simple version of any request parameters and, lastly, an error flag (with an error message, if the boolean *error* property is true); this is consistent with [what I've done before](#).

Below, when I handle the reflection of **JSON** to a **Java** Object (in conjunction with the `toJson`), I will show how to use both. Here's the providing of a collection, pulling entry information from a *ViewNavigator* into the **Java** object that will become the **JSON** string. I'm going to use a *HashMap* as my base object, with an *ArrayList* which will hold the individual data entries.

While it's certainly a lot of lines, I believe it to be fairly straight forward. In the *HouseCollection* class, there are defined three methods; *doGet*, *doPost*, and *handleUnexpectedVerb*. These are invoked by the main *HouseServlet* class, which calls the appropriate Collection or Record method, based on the full request path info and request method. I've included both the *com.google.Gson* and *com.ibm.commons.util.io.json* method, the latter is just commented out.

```
public class HouseCollection {

    private static String colAllowMethods = "GET, POST";

    public static void doGet(HttpServletRequest req, HttpServletResponse res,
        FacesContext facesContext, ServletOutputStream out) throws IOException {

        try {

            // the HashMap will represent the main JSON object
            HashMap<String, Object> myResponse = new HashMap<String, Object>();
            // the ArrayList will contain the JSON Array's data elements
            // which is another HashMap
            ArrayList<HashMap<String, String>> dataAr = new ArrayList<HashMap<String, String>>();

            Database db = Utils.getCurrentDatabase();
            View vw = db.getView("houses");
            ViewNavigator nav = vw.createViewNav();
            ViewEntry ent = nav.getFirstDocument();
            while( ent != null ) {

                Vector<String> colVals = ent.getColumnValues();
                HashMap<String, String> curOb = new HashMap<String, String>();
                curOb.put("name", colVals.get(0));
                curOb.put("description", colVals.get(1));
                curOb.put("words", colVals.get(2));
                curOb.put("unid", colVals.get(3));
                dataAr.add(curOb);

                ViewEntry tmpEnt = nav.getNext(ent);
                ent.recycle();
                ent = tmpEnt;
            }

            myResponse.put("dataAr", dataAr);
            myResponse.put("error", false);

            // IBM commons way of toJson
            /*
            * out.println(JsonGenerator.toJson(JsonJavaFactory.instanceEx,
            * myResponse));
            */

            // GSON way of toJson
            Gson g = new Gson();
            out.print(g.toJson(myResponse));

            res.setStatus(200); // OK
            res.addHeader("Allow", colAllowMethods);
        }
    }
}
```



```

    } catch (Exception e) {
        res.setStatus(500); // something pooped out
        res.addHeader("Allow", colAllowMethods);
        out.print( "{error: true, errorMessage: \""+e.toString()+"\"} " );
    }
}

public static void doPost(HttpServletRequest req, HttpServletResponse res,
    FacesContext facesContext, ServletOutputStream out) throws IOException {
    try {
        String unid;

        ServletInputStream is = req.getInputStream();
        // not that I'm using it, but the ServletRequestWrapper
        // can be quite helpful
        // ServletRequestWrapper srw = new ServletRequestWrapper(req);
        String reqStr = IOUtils.toString(is);

        // com.ibm.commons way
        /*
        JsonJavaFactory factory = JsonJavaFactory.instanceEx;
        JsonJavaObject tmpNwHouse = (JsonJavaObject) JsonParser.fromJson(factory, reqStr);
        Iterator<String> it = tmpNwHouse.getJsonProperties();
        HouseModel nwHouse = new HouseModel();
        nwHouse.setEditMode(true);
        while( it.hasNext() ) {
            String curProp = it.next();
            String curVal = tmpNwHouse.getAsString(curProp);
            nwHouse.setValue(curProp, curVal);
            it.remove();
        }
        */

        // GSON way
        Gson g = new Gson();
        /*
        * Direct reflection to the HouseModel breaks, as it
        * extends AbstractSmartDocumentModel :'-(.
        *
        * To get around that issue, as I know that the House
        * model is really a bunch of String key to String value pairs.
        * The AbstractSmartDocumentModel class basically adds some helper
        * methods to wrap a Map (representing the Notes
        * Document's Field to Value nature) with things like an edit
        * property, load (by unid) method, and save (for the obvious).
        */
        Map<String, Object> tmpNwHouse = (Map) g.fromJson(reqStr, HashMap.class);
        HouseModel nwHouse = new HouseModel();
        nwHouse.setEditMode(true);
        for (Map.Entry<String, Object> pair : tmpNwHouse.entrySet()) {
            String curProp = pair.getKey();
            String curVal = (String) pair.getValue();
            nwHouse.setValue(curProp, curVal);
        }

        nwHouse.save();
        unid = nwHouse.getUnid();
        res.setStatus(201);
        res.addHeader("Allow", colAllowMethods);
        res.addHeader("Location", "/xsp/houses/"+unid);
    } catch (Exception e) {
        HashMap<String, Object> errOb = new HashMap<String, Object>();
        errOb.put("error", true);
        errOb.put("errorMessage", e.toString());

        res.setStatus(500);
        res.addHeader("Allow", colAllowMethods);
        Gson g = new Gson();
        out.print(g.toJson(errOb));
    }
}
}

```

```

    public static void handleUnexpectedVerb(HttpServletRequest req,
        HttpServletResponse res, FacesContext facesContext,
        ServletOutputStream out) {
        res.setStatus(405);
        res.addHeader("Allow", colAllowMethods);
    }
}

```

You can find how I'm able to *POST* a new document in the *doPost* method here, but I'll cover that process in more detail further down.

Document

Handling the individual records, the *NotesDocument_s*, gets more fun. I'm not just stepping through a *_NotesViewNavigator* and for me personally, this is why we should be embracing our *Java* roots on *Domino/XPages*. Say I have myself set up for using a managed bean to represent my documents. Aside from the *Notes/Domino API* specifics, we're dealing with an otherwise plain *Java* object, in memory, to represent our data record, with which we interact. Using that same bean, I'm able to interact with it the same in my servlet as I might through the *XPages* UI. The biggest difference is that it's as a POJO (plain ol' *Java* object), as it's not managed, not defined in my *Faces-config* and has no "scope"; it'll be created/loaded, modified, and saved as fast as the servlet responds.

Here's my *HouseRecord* class, explanation afterwards.

```

public class HouseRecord {

    private static String recAllowedMethods = "GET, PUT, DELETE";

    public static void doGet(String unid, HttpServletRequest req, HttpServletResponse res,
        FacesContext facesContext, ServletOutputStream out) throws IOException {

        try {

            // create a House model object in memory and load its contents
            HouseModel myHouse = new HouseModel();
            myHouse.load(unid);

            //return the contents in JSON to the OutputStream

            // com.ibm.commons way
            /*
             * out.print(JsonGenerator.toJson(JsonJavaFactory.instanceEx,
             * myHouse));
             */

            // GSON way
            Gson g = new Gson();
            out.print( g.toJson(myHouse) );

            res.setStatus(200);
            res.addHeader("Allow", recAllowedMethods);

        } catch(Exception e) {
            res.setStatus(500);
            res.addHeader("Allow", recAllowedMethods);
            Map<String, Object> errOb = new HashMap<String, Object>();
            errOb.put("error", true);
            errOb.put("errorMsg", e.toString());
            Gson g = new Gson();
            out.print(g.toJson(errOb));
        }
    }

    public static void doPost(String unid, HttpServletRequest req, HttpServletResponse res,
        FacesContext facesContext, ServletOutputStream out) throws IOException {

```

```

try {

    // GET existing
    HouseModel exHouse = new HouseModel();
    exHouse.load(unid);

    ServletInputStream is = req.getInputStream();
    String reqStr = IOUtils.toString(is);

    Gson g = new Gson();

    // setting the keys/values into the tmpNwHouse Map
    Map<String, Object> tmpNwHouse = (Map) g.fromJson(reqStr, HashMap.class);
    // suppressing just this warning throws an error on tmpNwHouse
    tmpNwHouse = g.fromJson(reqStr, tmpNwHouse.getClass());
    HouseModel nwHouse = new HouseModel();
    nwHouse.setEditMode(true);
    // compare/update
    for(Map.Entry<String, Object> pair : tmpNwHouse.entrySet() {
        String curProp = pair.getKey();
        String curVal = (String) pair.getValue();
        if( exHouse.getValue(curProp) != curVal ) {
            exHouse.setValue(curProp, curVal);
        }
    }

    // done setting new values back into the existing object
    exHouse.save();

    res.setStatus(200);
    res.addHeader("Allow", recAllowedMethods);

} catch(Exception e) {
    res.setStatus(500);
    res.addHeader("Allow", recAllowedMethods);
    Map<String, Object> errOb = new HashMap<String, Object>();
    errOb.put("error", true);
    errOb.put("errorMsg", e.toString());
    Gson g = new Gson();
    out.print(g.toJson(errOb));
}

}

public static void doDelete(String unid, HttpServletRequest req, HttpServletResponse res,
    FacesContext facesContext, ServletOutputStream out) throws IOException {

    Session s = (Session) facesContext.getApplication().getVariableResolver().resolveVariable(facesContext, "session");
    Document houseDoc;
    try {
        houseDoc = s.getCurrentDatabase().getDocumentByUNID(unid);
        houseDoc.remove(true);
        houseDoc.recycle();
        res.setStatus(200);
        res.addHeader("Allow", recAllowedMethods);
    } catch (NotesException e) {
        res.setStatus(500);
        Gson g = new Gson();
        Map<String, Object> errData = new HashMap<String, Object>();
        errData.put("error", true);
        errData.put("errorMessage", e.toString());
        errData.put("stackTrace", e.getStackTrace());
        out.print(g.toJson(errData));
    }

}

public static void handleUnexpectedVerb(HttpServletRequest req,
    HttpServletResponse res, FacesContext facesContext,
    ServletOutputStream out) {
    res.setStatus(405);
    res.addHeader("Allow", recAllowedMethods);
}

```

```
    }  
  }  
}
```

Obviously a delete operation is just a delete and we've covered *GET*, but the *PUT* is where I had fun with things. The *POST* above assumes an entirely new object, but with the *PUT* as I've implemented it, allowing for full or partial replacement, I need to instantiate the existing record into an object and then pull and compare/update any values. Just as I'm iterating the *HashMap*'s values in the *Collection POST*, instead of just filling the values, I'm comparing the values and replacing as needed, inside a while loop, iterating the *Map.Entry<String, Object>* (pair) values, like so:

```
String curProp = pair.getKey();  
String curVal = (String) pair.getValue();  
if( exHouse.getValue(curProp) != curVal ) {  
    exHouse.setValue(curProp, curVal);  
}
```

Uniqueness of Using an Abstracted Document

This concept relies on having an object model class. The modeling of my house object does what a bean does, has properties which hold values, and interacts via getter and setter methods. For my app, I'm using an (older) implementation of the OpenNTF [Domino API](#); specifically the *AbstractSmartDocumentModel*, as found in [Tim Tripcony's How Ya Bean application](#) and [affiliated Notesin9 videos](#). This is to automate the getter/setter methods (it specifically ditches *get/set PropertyName* in favor of *get/set Value*). It also means that my app is a bit more portable (full project coming to a [GitHub](#) repository near you, soon!).

ToJson

I also create the *JSON* with the *Gson* library, as I've covered both the *Gson* and [com.ibm.commons.util.io.json](#) approaches before, when it comes to creating a *JSON* string, so I won't repeat myself here. The only thing of major difference is to build out your response into a *Java* object, then use a [com.ibm.commons.util.io.json.JsonGenerator's toJson](#) method.

An Object Model for (Almost) Everyone

An object model, for my purposes, is a bean. It provides the definitions for what data to store and in what format. It is my preference to keep any additional business logic, such as notifications (emails, etc) or validation, in a separate class, though this isn't necessary.

If you're ever looking for help in generating a POJO from *JSON*, I recommend checking out [jsonschema2pojo.org](#). As I'm an avid user of *Gson* and Apache Commons, the options I select are *JSON* (not *JSONSchema*), *Gson*, Use double numbers, Use Commons-Lang3, and Include *toString*; like this:

Package

Class name

Source type:

JSON Schema JSON

Annotation style:

Jackson 2.x Jackson 1.x

Gson None

Generate builder methods

Use primitive types

Use long integers

Use double numbers

Use Joda dates

Use Commons-Lang3

Include `hashCode` and `equals`

Include `toString`

Include JSR-303 annotations

Property word delimiters:

As I mentioned in my caveat above, as my *HouseModel* extends *AbstractSmartDocumentModel*, I don't have the usual *get/set Property* methods, but rather *getValue/setValue*; since this is the case, reflecting my received *application/json* content from the *HttpRequest* directly into my *HouseModel* for a new instance, meaning that I have to do some processing of that data to fill a new instance of a *HouseModel*. Since I know the data format I'll be expecting, I'm going to read everything into a *HashMap<String, Object>*, then populate my *HouseModel* from that. I could probably write my own *GsonBuilder* to account for this difference, but I'm not going that far into things.

The HouseModel

To demonstrate why I'm using an abstracted model which doesn't conform exactly to bean conventions (the getter/setter methods being replace by a universal *getValue/setValue*, for instance), have a look at the simplicity of my *HouseModel* class.

```
public class HouseModel extends AbstractSmartDocumentModel {
    private static final long serialVersionUID = 1L;

    private String name;
    private String description;
    private String coatOfArms;
    private String words;
    private String seat;
    private String currentLord;
    private String region;
    private String title;
    private String heir;
    private String overlord;

    @Override
    protected String getFormName() {
        return "house";
    }

    @Override
```

```

public void load(final String unid) {
    super.load(unid);
}

@Override
protected boolean querySave() {
    return true;
}
}

```

That's it, nothing else. This should be the hallmark of why you should [go check out the OpenNTF Domino API](#) right now. As I said already, this keeps me from directly reflecting via [Gson](#) or [IBM commons JSON](#), but I can live with that for this level of simplicity.

Receiving Data from POST or PUT

ServletInputStream

To read in the data contained within the *HttpServletRequest*'s body, we need to get a handle on the *ServletInputStream*. More of that below, in the example.

FromJson

Performing the *fromJson* (reading the [JSON](#) string into an Object) can be done by either *com.google.Gson* or *com.ibm.commons.util.io.json*. Both work well, and I have my preference to [Gson](#), but something I found out in doing it both ways was that I rather like the *com.ibm.commons.util.io.json* approach for a particular reason. In my class, visible in the above Collection *POST* handling method, I'm creating my consumed request data first as a *HashMap<String, String>* so that I can iterate the values and build out my appropriate object; this works, but one nicety of the [IBM JSON](#) package is that it is easily created first as a *JsonObject*, which is similar but provides some convenience methods for property access.

Using the InputStream Directly

Instead of iterating the bytes of the content from the *InputStream*, we can use another [Apache Commons utility, IOUtils](#), to automate this for us. Here's a reflection of a traditional bean (with the usual getter and setter methods) from the *InputStream*.

```

// req is the passed in HttpServletRequest
ServletInputStream is = req.getInputStream();
Gson gson = new Gson();

MyBean myBean = (MyBean) gson.fromJson(IOUtils.toString(is), MyBean.class);

```

My Class's Interpretation

As mentioned above, here's how I'm reading my values into a *HashMap* and then filling my object with the *setValue* methods.

```

String reqStr = IOUtils.toString(is);
Gson g = new Gson();
// create the tmp HashMap
Map<String, Object> tmpNwHouse = new HashMap<String, Object>();
// fill the values via Gson, self-referencing the HashMap class
tmpNwHouse = g.fromJson(reqStr, tmpNwHouse.getClass());
// iterate the values and put them into the proper HouseModel object
HouseModel nwHouse = new HouseModel();
Iterator<Map.Entry<String, Object>> it = tmpNwHouse.entrySet().iterator();
nwHouse.setEditMode(true);
while (it.hasNext()) {
    Map.Entry<String, Object> pair = it.next();
    String curProp = pair.getKey();
    String curVal = (String) pair.getValue();
    nwHouse.setValue(curProp, curVal);
}

```

```
    it.remove();
}

// any additional validations, balances, notifications, etc.
nwHouse.save();
// 201 = "Created", should include "Location" header
res.setStatus(201);
res.addHeader("Location", "/xsp/houses/"+nwHouse.getUnid());
```

Provide Response

You'll see I'm relying on the response code to communicate the success. This is what [jQuery](#) and [AngularJS](#) key off of to determine the success/fail of the network event for their respective callbacks. In my error handling, I respond with a status code of 500, and *application/json* content in the body, to the effect of:

```
{
  error: true,
  errorMessage: "whatever my Exception.toString() is"
}
```

This once again highlights the need to document your [API](#). It's okay to use the status codes for primary information, but definitely *at least* put some error messages in for a failing operation.

Note: On PUT and DELETE Methods

I ran into something with this, which I wasn't expecting. I had to enable PUT and DELETE methods in my [Domino](#) Designer while testing locally. It seems that my PUT and DELETE calls were being hijacked and consistently throwing 405: method not allowed calls. This threw me for a loop, as my development and production servers didn't have this issue. My suspicion is that they were already enabled, via enabling of the [Domino](#) Data Services, previously.

To enable PUT and DELETE (or PATCH, though I've avoided it for simplicity's sake), you should do so by any of:

- enable in Internet Site (if your server uses them)
- enable in [Notes.ini](#) (specifics below)
- work around using X-Http-Method-Override

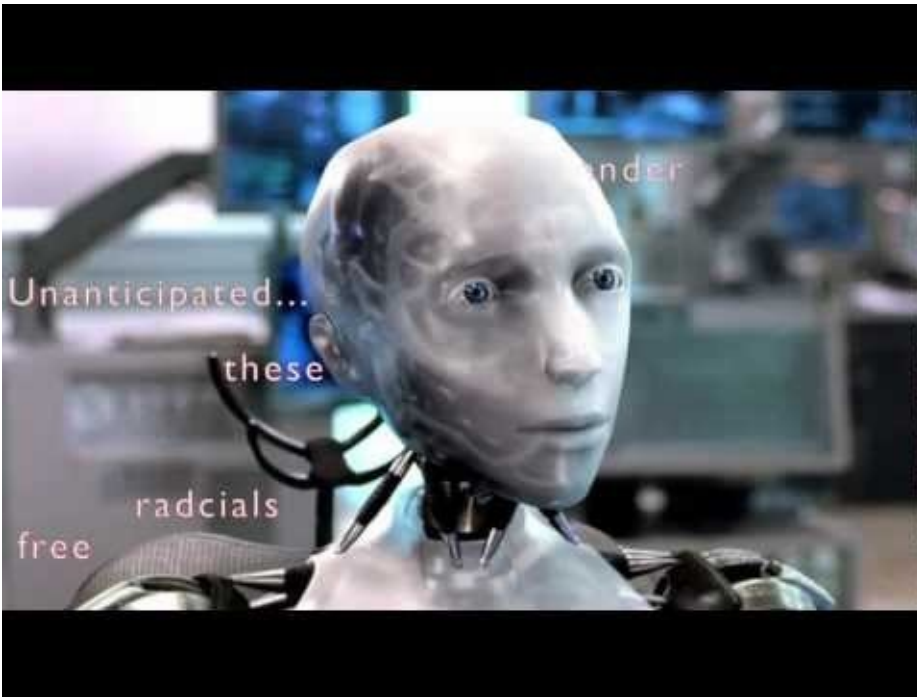
Using the X-Http-Method-Override seems silly, but is pretty easy to use. Here's a [jQuery.ajax](#) example of a *PUT* request being sent as a *POST*, taken from a [StackOverflow answer on the subject](#):

```
$.ajax({
  beforeSend: function(xhr) {
    xhr.setRequestHeader('X-HTTP-Method-Override', 'PUT');
  },
  type: 'POST',
  url: '/someurl',
  success: function(data){
    // do something...
  }
});
```

As for the path I took for my personal development environment, I added the following line to [notes.ini](#):

```
HTTPEnableMethods=PUT,DELETE
```

I initially didn't see it work, as I added it to the end of my file. Once I placed that directly under where I define my local web preview port (further up the file), it started to work without issue. Must be the ghosts in the machine.



[Video link](#)

Summary

I've covered a whole heck of a lot in this post. We split our servlet to handle collection operations (getting the collection and creating a new entry) and the record operations (getting the full content of a single record, updating a record in part or in whole, and deleting records) and worked with a consistent interface via a near-POJO data object, which acts the same as the managed bean use in my code base (see the GH repo, link below).

I also know there are people out there thinking, "but there's this better way to do this part!" Great! Please show us and/or me how. I also welcome all constructive comments below.

To see my application code to this point, by all means check out [my GitHub repository](#) for it. Follow the ReadMe.md instructions to get started. This repository will update once I've completed the next two posts. I still want to cover how to convert XAgent logic to a servlet and creating a basic front-end interface to this servlet with [AngularJS](#). So please stay tuned to this series, as there's more to come!

A Quick Review

I had some trepidation about this post; it revolves around the fact that I'm "completing" my blog series with multiple giant topics, on top of the one primary one I've focused on for the majority of this blog series. So, before we get started, I'm going to summarize a couple things. But first, a ToC:

- [Front-End Consumption](#)
- [Why AngularJS?](#)
- [Tomorrow](#)

Servlets

I've referred to this series as [#ASagaOfServlets](#). While most [Java](#) servlets are intended for use over HTTP (at least from a JEE, web container standpoint), this is not exclusive; I've used [HTTPServlet](#) as analogous to Servlet (for better or for worse).

RESTful API

A [REST API](#) is an architectural style of [API](#). There is no concrete definition of what required for an [API](#) to be RESTful, but it's best if it follows a couple conventions ([previously covered](#)); this generally boils down to:

- a resource based interface, following [HATEOAS](#)
- be stateless (no server session required, the URI request gives all the server needs to know)
- be cachable or not (depending on what sort of data you're providing)
- work entirely independent of any particular client format (while adhering to certain things like authentication and formatted requests)

There are more that a [RESTful API](#) can do or rules that can be applied, but that's the high level stuff. As you can see, this is part of the core of the segregation of data and primary business logic from the client-layer side of the application.

"Stack" Development

Part of my crusade in the realm of segregating application development concerns into the front-end and back-end revolves around the concept of these "ends" to the application. Both play an important role, but work best together. By building your back-end to adhere to certain conventions, you can create your front-end with any front-end technology. This is why I'm such a huge fan. At my company, we have a large number of in-house systems, many of which talk to each other. By segregating the primary business logic (governing how we store the data, events that trigger from the server, and steps in workflow) as being a part of how the server components work, then any client playing by the rules can be a valid interface; whether that's an automated agent which checks for non-interface updates, or the front-end which contains all the user interaction at the UI level. The business logic become much more maintainable and documentable in the process.

Where [XPages](#) fits in as a component in all of this can be a little tricky. Obviously, [XPages](#) design elements encompass the application layer, but deciding how that maps to a front-end as opposed to a back-end is a bit trickier (and [one I've complained about debated before](#)). I don't mean to beat up on [XPages](#), as it offers us quite a lot of tools and components that help assemble a working app, rapidly; I can and will beat up on poorly implemented [XPages](#) application code.

XPages: Full-Stack Development?

Obviously, certain beginner [XPages](#) development approaches (those conducive to [SSJS](#) spaghetti code™) can be quite the antithesis of what the segregated stack approach gives us. This makes our [XPages](#) design elements, containing not just the markup and layout of elements (fields, labels, etc.), but also logic, `if(status.equals("certainStep")){ doSomethingUnique(); }`, and actions (since these X conditions are true, send an email to these 12 people). Combine this with the unique, [NoSQL database](#) handling via the [NotesDominoAPI](#), it's my belief that [XPages](#) development is by default a full-stack application stack; for better or for worse.

Aside (talking crazy for a moment)

Some of these concepts are central to what I've seen previewed of the [XPages \(XSP\) application runtime](#) and [Domino Data Service](#) on Bluemix. That the data container being forced to be separate from the application layer isn't just a good idea with Bluemix (which enforces the segregation of concerns as does almost any other application stack, considering that nearly all out there aren't configured

like an NSF), but means that the [XPages](#) runtime can hook into any database; something it's already capable of, but often not done. In fact, segregating the data NSF from the application NSF isn't a new concept either, but hey, it's my paragraph 😊. I'm also fairly certain that the segregation of the XSP runtime from the other, traditional NSF components may be the gateway for us to get an updated [JVM](#), but maybe I'm just greedy.

Ultimately, the point I'm trying to make, is that we have a lot of options and decisions we can make with [Domino/XPages](#), but with any ambiguity, there are potential pitfalls. One way this is changing, IMO, is [the bringing of the XSP\(XPages\) runtime to Bluemix](#). In case you missed it, [I've posted a couple early thoughts on Bluemix](#), and I'm both impressed and excited for what it can and will bring to the table for my company and I.

Front-End Consumption

Having shaped our back-end [earlier in this series](#) to adhere to a primarily [RESTful API](#) format, we can now consume that [API](#) by *some front-end*. In the [Notes in 9 173: Getting Started With \(HTTP\)Servlets](#) video, I demonstrated this principle via the [Postman REST client](#) (a Chrome extension). There are others out there and you could even [test from your command line via cURL](#), if you're so inclined. What this demonstrates is that virtually *any* front-end can consume the [API](#), it just comes down to how you expose/provision that [API](#) and what you point to it.

It also shows the method of data transfer. In order for a front-end to work with my [RESTful API](#), it will need to:

- provide/receive all data in application/[json](#)
- stick to the available resources (*houses*)
- create a new entry, one-at-a-time, against the collection [endpoint](#) (*/houses*)
- read, update, delete against the (UN)ID keyed URI (*/houses/[0-9a-zA-Z]{32}*)
- collection data is accessible via */houses*

JavaScript Consumption

Front-end development in this day and age focuses on [JavaScript](#) usage. Most people use a framework of some flavor, to automate the things they'd rather not spend too much time on. Some of these things include standardizing how you interact with an HTTP [RESTful API endpoint](#), or automate the updating of data between bound components. The fact of the matter is, there are plenty of frameworks out there, many which can help you in your quest.

JavaScript Frameworks

Choosing a [JavaScript](#) framework can be a little daunting, if you're doing so for the first time. There's a long history of many web applications making use of [jQuery](#) or [Dojo](#), both of which are libraries(/frameworks) that automate quite a bit, they're not of the MVC/MV* flavor I'm looking for. The fact remains, one can make a fully-formed application out of either, I just won't focus on them.

[Aside] There are [jQuery UI \(and mobile\)](#) and [Dojo MVC](#), but I'm moving on for simplicity's sake. [/Aside]

MVC/MV* Architecture

There are a lot of acronyms in framework architecture that get thrown around. Here are a couple to get you started:

- MVC - Model-View-Controller
- MVVM - Model-View-ViewModel
- MVW / MV* - Model-View-Whatever

This list is far from all-inclusive, and is a bit of a side-topic to what I want to focus on here. Just [remember how a model, view, and controller](#) represent different pieces of the application pie, and all will be good.

If you want to read up more on the theory of why you would want an MVC/* framework, I recommend checking out [this answer on Quora](#) on the subject. It's a good read which espouses the need for the a framework but as they point out, no one solution (e.g.- Backbone in their example) is an end-all, be-all.

FWIW

[AngularJS](#) (as you can probably have guessed is the front-end framework I'm using) considers itself to be an [MV*/MVW framework](#)

HTML enhanced for web apps!

and has ditched the MV-something classification almost entirely.

No matter your decision on frameworks, the bottom line is that you should use one that plays to your strengths, and you should play to the strengths of the framework you choose.

Why **AngularJS**?

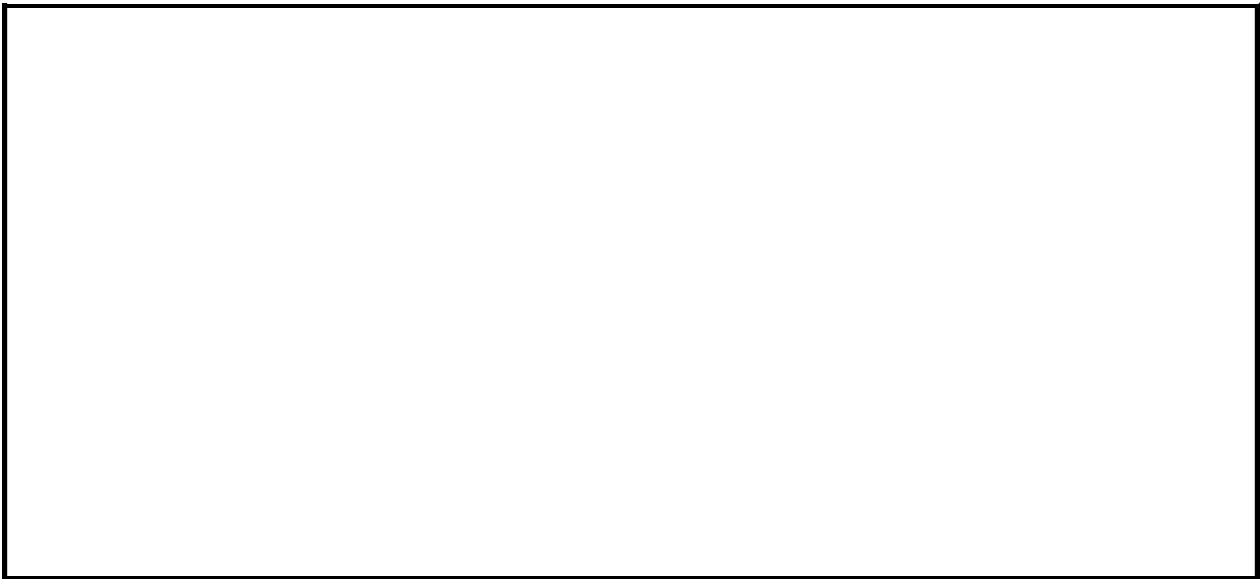
AngularJS set out to conquer some considerable hurdles when it began. The HTML5 spec was in its infancy and the front-end frameworks out there were achieving a few good things, but the **Angular** team wanted more.

Here are the reasons I gave for **AngularJS** (with some definite overlap with other frameworks) from my Chalk Talk Soup rebel slide deck:

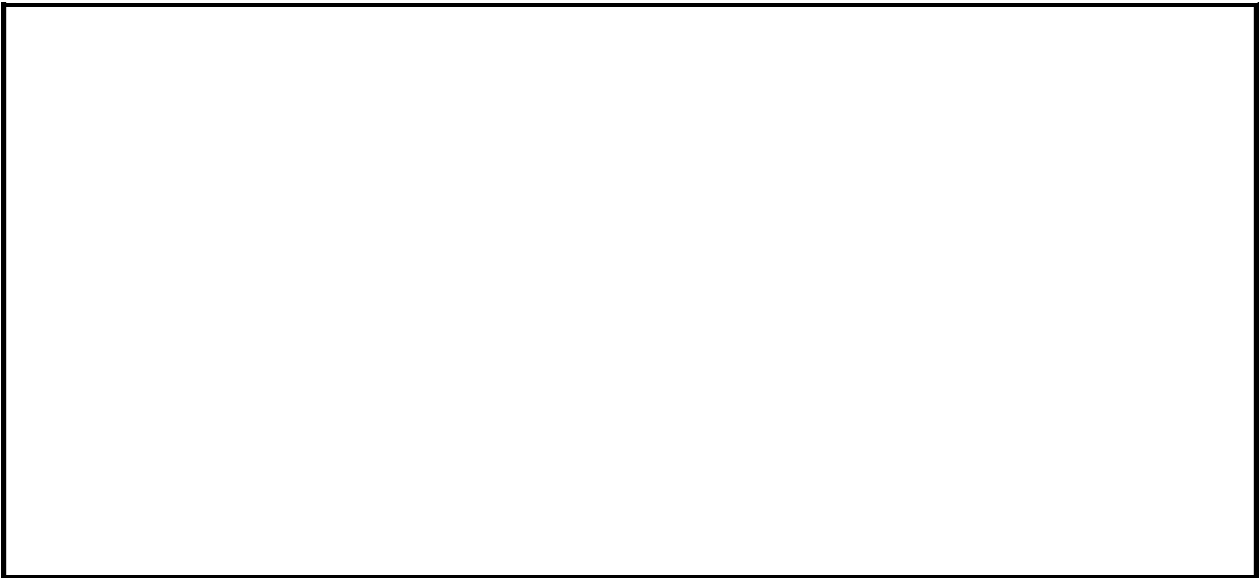
- bi-directional data binding (all data models by default auto-update their other references on data change, within the scope)
- templates (via ng-include or ng-route; also ui-router, 3rd party)
- OoB directives, services, filters, and more
- dependency injection
- unit testing (**AngularJS** was developed with e2e testing in mind, and docs examples include protractor scripts)

Here are a couple examples I had prepared for that slide deck:

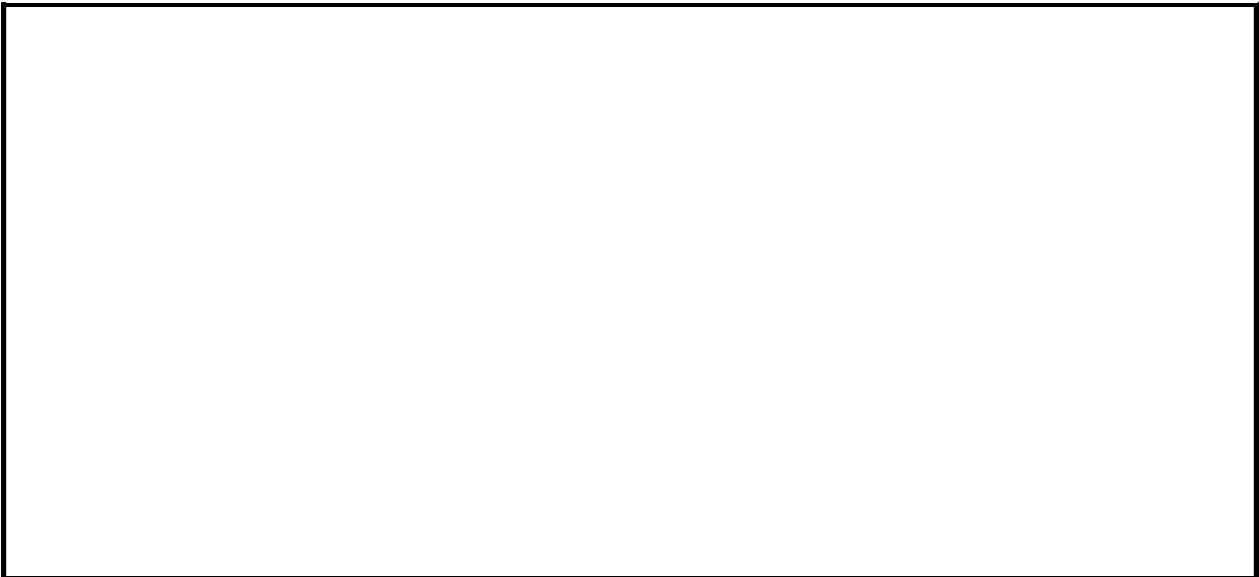
Bi-directional data binding:



Dynamic templates:



Filters (out of the box!):



To add some fuel to the fire, here is a link to the [Google Trends for Angular, Backbone, and Ember](#). As an side, check out [other combinations of search terms](#), it can be interesting to play with; it only yields results as scraped from Google search, so it's no absolute indicator, but interesting as it is.

For another [good comparison between Angular, Backbone, and Ember](#), this articles does a decent job of breaking down "the good parts" and the "pain points". The article is hosted on [airpair.com](#), a micro-consulting site geared for developer-to-developer support, be it mentoring, code review, and more.

It's a sign of one of the other advantages of this form of segregated, "stack" design; outside help that's not such a closed ecosystem as the one we work in. This may not be a huge deal for those who aren't customers, but for those who seek to at least stay afloat, it's a decent leap towards being able to outsource without a huge amount of 💰.

Scary Change is Scary

Recently you may have seen [David Leedy blog a link](#) and ask for perspective on [a particular post denouncing AngularJS and all its sins](#). All I can say is, read the comments along with the post. I personally found the post to be inconsisent with my experiences but, more importantly, ignoring certain facts and updates (which the [AngularJS](#) team does provide on a constant basis) for the sake of their argument. Make up your own mind, but be informed.

A Note on Version 2

AngularJS version 2.0 takes advantage of ECMAScript 6 and follows a format considerably more like [web components](#). This means that it will fit in well with the final release of the HTML5 spec. It's also on the early side and as the [AngularJS 2.0 site points out](#),

Angular 2 is currently in Developer Preview. We recommend using [Angular 1.X](#) for production applications.

For now, I'm rocking the 1.x line, specifically staying in 1.3.x for my current app. A lot of people are trying to make a big deal out of Google's choice to break 2.x from 1.x, but the fact of the matter is that 1.x isn't going anywhere just yet and will have a stable branch for quite some time to come. I first started dabbling on [AngularJS 0.9.8](#), and started grasping much more of it after 1.0 hit. If I was so inclined, there is a stable 1.0.8 release available right on [angularjs.org](#) including [documentation at that level](#), and 1.0.8 was released Aug 22nd, 2013.

So, all those naysayers, I say pick a framework. I'm going with [AngularJS](#) and it's been pretty pimp so far.



My Feelings About AngularJS Over Time

Tomorrow

Come back tomorrow for the conclusion of this epic journey.



[Video link](#)

Ever Onward

For as much theory and verbiage as yesterday's post was, today's will be primarily code-driven; something I hope you're ready for. I'll run through this all and hopefully I can illustrate succinctly as we go.

- [HTML Templating](#)
- [AngularJS App](#)
- [Bring It Home](#)

HTML Templating

HTML templating is useful because it frames out the structure of a page, in its components parts, and, possibly the most useful attribute, it can be cached by the browser. This is highly useful for a lot of traffic and saves on the overhead of transporting markup with your data in every update of data. It's one of the topics Marky Roden talked about during his [5 Questions with Marky Roden](#) video for SocialBizUG.org.

The initial page for the Houses of AnAppOfIceAndFire (*index.html*) is laid out like almost anyone would expect an *index.html* file that implements Bootstrap. I've snipped out everything but the `<body>` tag contents for space.

```
<!-- ...head contents... -->

<!-- defining where to inject our app definition, using the body tag -->
<body ng-app="houseApp">
  <div class="navbar navbar-default navbar-fixed-top" role="navigation">
    <!-- ...navbar contents... -->
  </div>

  <!-- the magic! -->
  <div ui-view></div>

  <script type="text/javascript" src="js/houseApp.js"></script>
</body>
<!-- ... -->
```

The "magic happens" part is where my application code structures in the HTML partials, which I route in, based on my config. We'll get there in a minute, for now, have a look at the two partial HTML files I'm using, one for the collection list and one for the individual house. You may notice that I'm also nesting my House Record inside the House Collection partial, this is one of the nifty features I like about ui-router.

House Collection

```
<div id="uiContainer" class="container">
  <div class="row">
    <div class="col-md-4">
      <div class="panel panel-default">
        <div class="panel-heading">
          <h3 class="panel-title">Houses of the Seven Kingdoms of Westeros</h3>
        </div>
        <ul class="list-group list-group-striped">
          <li class="list-group-item">
            <ng-repeat="house in housesOfWesteros | startFrom : curPage*pageQty | limitTo:pageQty">
              <a ng-href="#/houses/{{ house.unid }}" title="{{ house.unid }}">{{house.name}}</a>
              <a href="#"
                class="btn btn-danger pull-right"
                ng-really-message="Are you sure you want to delete this house from Westeros?"
                ng-really-click="removeHouse(house.unid)"><i class="fa fa-lg fa-trash-o"></i></a>
              <br />{{house.words}}
            </li>
          </ul>
          <div class="panel-footer col-xs-12">
```

```

        <nav>
          <ul class="pager">
            <li class="previous" ng-class="{ 'disabled': curPage == 0 }">
              <a ng-click="curPage = curPage - 1" href=""><span aria-hidden="true">&larr;</span> Prev:
            </li>
            <span ng-bind="(curPage+1) + '/' + numberOfPages()"></span>
            <li class="next" ng-class="{ 'disabled': curPage >= housesOfWesteros.length/pageQty-1 }">
              <a ng-click="curPage = curPage + 1" href="">Next <span aria-hidden="true">&rarr;</span>
            </li>
          </ul>
        </nav>
      </div>
    </div>
  <!-- single house content -->
  <div ui-view class="col-md-6 col-md-offset-2"></div>
</div>
</div>

```

House Record For obvious reasons, much more like a form.

```

<div class="panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title">House Details</h3>
  </div>
  <div class="panel-body">
    <form name="houseForm">
      <div class="form-group">
        <label for="houseName">Name</label>
        <input type="text" class="form-control" id="houseName" name="name" ng-model="myHouse.name" ng-disabled="!editForm" />
      </div>
      <div class="form-group">
        <label for="houseDescription">Description</label>
        <textarea class="form-control" id="houseDescription" name="description" ng-model="myHouse.description" rows="3" ng-disabled="!editForm" />
      </div>
      <div class="form-group">
        <label for="coatOfArms">Coat of Arms</label>
        <input type="text" class="form-control" id="coatOfArms" name="coatOfArms" ng-model="myHouse.coatOfArms" ng-disabled="!editForm" />
      </div>
      <div class="form-group">
        <label for="houseWords">Words</label>

```



```
<input
  type="text"
  class="form-control"
  id="houseWords"
  name="words"
  ng-model="myHouse.words"
  ng-disabled="!editForm" />
</div>
<div class="form-group">
  <label
    for="houseSeat">
    Seat</label>
  <input
    type="text"
    class="form-control"
    id="houseSeat"
    name="seat"
    ng-model="myHouse.seat"
    ng-disabled="!editForm" />
</div>
<div class="form-group">
  <label
    for="houseCurrentLord">
    Current Lord</label>
  <input
    type="text"
    class="form-control"
    id="houseCurrentLord"
    name="currentLord"
    ng-model="myHouse.currentLord"
    ng-disabled="!editForm" />
</div>
<div class="form-group">
  <label
    for="houseRegion">
    Region</label>
  <input
    type="text"
    class="form-control"
    id="houseRegion"
    name="region"
    ng-model="myHouse.region"
    ng-disabled="!editForm" />
</div>
<div class="form-group">
  <label
    for="houseTitle">
    Title</label>
  <input
    type="text"
    class="form-control"
    id="houseTitle"
    name="title"
    ng-model="myHouse.title"
    ng-disabled="!editForm" />
</div>
<div class="form-group">
  <label
    for="houseHeir">
    Heir</label>
  <input
    type="text"
    class="form-control"
    id="houseHeir"
    name="heir"
    ng-model="myHouse.heir"
    ng-disabled="!editForm" />
</div>
<div class="form-group">
  <label
    for="houseOverlord">
    Overlord</label>
```

```

    <input
      type="text"
      class="form-control"
      id="houseOverlord"
      name="overlord"
      ng-model="myHouse.overlord"
      ng-disabled="!editForm" />
  </div>
  <div class="btn-list pull-right">
    <button
      class="btn-default btn"
      type="button" id="buttonCancelGotHouse"
      ng-really-message="Are you sure?"
      ng-really-click="clearCancelForm()">
      <i class="fa fa-lg fa-pencil"></i> Cancel</button>
    <button
      class="btn-success btn"
      type="button" id="buttonSaveGotHouse"
      ng-show="editForm"
      ng-click="saveHouseForm()">
      <i class="fa fa-lg fa-save"></i> Save</button>
    <button
      class="btn-primary btn"
      type="button" id="buttonEditGotHouse"
      ng-disabled="!canEditForm"
      ng-show="!editForm"
      ng-click="setFormEditable()">
      <i class="fa fa-lg fa-pencil"></i> Edit</button>
  </div>
</form>
</div>
<!-- <div class="panel-footer col-xs-12"></div> -->
</div>

```

AngularJS App

0 - Structure

My app will consist of a few parts. I've broken them apart here into sections, for ease of reading. I've also taken the approach for my app.js of chain-loading each section off the main module definition, decreasing the number of handles for the same object.

1 - Config

I'll first need to configure any routing rules for my HTML partials and resolving URL route parameters as their respective variables; this will happen [in the config](#); the definition is for an [Angular](#) "module". Any 3rd party assets get plugged in here, as part of the dependency injection, such as [ui-router](#).

```

// a self-invoking, anonymous function to keep application code variables scoped anonymously
(function(){

  //defines the AngularJS app as a module
  angular.module('houseApp', ['ui.router']) //ngAnimate'

  //ui-router config
  .config(
    function($stateProvider, $urlRouterProvider){

      $urlRouterProvider.otherwise('/houses');

      $stateProvider
        .state('houses', {
          url: '/houses',
          templateUrl: 'partials/houseList.html',
          controller: 'HouseListCtrl'
        })
        .state('houses.item', {

```

```

        url:('/:item',
        templateUrl: 'partials/house.html',
        controller: 'OneHouseCtrl'
    });
    });

    // ... services/factories, controllers, filters, directives

})();

```

2 - Services/Factories

Any [services or factories \(or providers\)](#) get defined here.

```

//...config...
/*
 *   Factories
 */

//defines the $HTTP factory, one of the 3 service types
.factory('houseCollectionFactory', [ '$http', function($http) {
    return $http( {
        method : 'GET',
        url : 'houses'
    });
} ] )

.factory('houseFactory', [ '$http', function($http){
    return function(id){
        return $http( {
            method : 'GET',
            url : 'houses/'+id
        });
    }
}])
//...controllers, filters, directives...

```

3 - Controllers

[Controllers](#) are a binding of functional behavior to sections of the HTML. I have two controllers, each with different scopes. Mine are for my navigation handling and the primary application regarding houses.

```

//...config, factories...
/*
 *   Controllers
 */

//navigation controller
.controller('NavCtrl', function($scope, $location){
    $scope.isActive = function(route) {
        return route === $location.path();
    }
})

//provides the controller to the app, which handles the interaction of data (model) with the view (a la MVC)
.controller('HouseListCtrl', function($scope, $state, $http, $filter, houseCollectionFactory) {

    //defines filter/search/etc. vars
    $scope.pageQty = 5; //detectPhone() ? 10 : 30;
    $scope.curPage = 0;

    //calculates the number of results
    $scope.numberOfPages = function() {
        return Math.ceil($scope.housesOfWesteros.length / $scope.pageQty) || 0;
    };
}

```

```

}

//defines a boolean var
$scope.showSearch = false;

$scope.housesOfWesteros = [];
//the factory is returning the promise of the $http, so handle success/error here
houseCollectionFactory
    .success( function(data, status, headers, config) {
        $scope.housesOfWesteros = data;
        //$scope.predicate = "JobNum";
        //$scope.reverse = false;
    }).error( function(data, status, headers, config) {
        $scope.housesOfWesteros = null;
        console.log("data: " + data);
        console.log("status: " + status);
        console.log("headers: " + headers);
        console.log("config: " + JSON.parse(config));
    })
    .then( function(){
        //angular.element('div.screenMask').css('visibility','hidden');
    });

$scope.removeHouse = function(unid){
    $http( {
        method : 'DELETE',
        url : 'houses/'+unid
    })
    .success( function(data, status, headers, config){
        console.log("successfully deleted house with id: "+unid);
    })
    .error( function(data, status, headers, config){
        //might as well say something
        console.log("poop");
    })
    .then( function(){
        $state.go('houses',{reload: true});
    });
};

})

.controller('OneHouseCtrl', function($scope, $state, $stateParams, $http, houseFactory){
    // check for empty ID
    var tmpItm = $stateParams.item;
    console.log("unid: "+tmpItm);
    var re = /^[0-9A-Za-z]{32}$/;
    //var re = /\d/;
    if( tmpItm == null || tmpItm == undefined || (!tmpItm || !tmpItm.trim()) || !re.test(tmpItm) ){
        $state.go('houses');
    }

    $scope.editForm = false;
    $scope.canEditForm = false;
    $scope.myHouse = {};
    var fieldNames = [];
    houseFactory($stateParams.item)
        .success(function(data, status, headers, config) {
            $scope.myHouse = data;
            $scope.canEditForm = true;
            angular.forEach($scope.myHouse, function(value, key){
                if( key!="unid" ){
                    fieldNames.push(key);
                }
            });
        })
        .error(function(data, status, headers, config) {
            console.log("status: "+status);
            console.log("data: "+data);
            console.log("headers: "+headers);
            console.log("config: "+JSON.parse(config));
        });
};

```

```

$scope.setFormEditable = function() {
    if( $scope.canEditForm == true ){
        $scope.editForm = true;
    }
}
$scope.clearCancelForm = function() {
    $state.go('houses');
}

$scope.saveHouseForm = function(){
    var tmpOb = { "unid": $scope.myHouse.unid };
    //console.log("checking field names: "+fieldNames.toString());
    angular.forEach(fieldNames, function(fldNm){
        if( $scope.houseForm[fldNm].$dirty === true ){
            var tmpVal = $scope.myHouse[fldNm];
            //console.log("updated field: "+fldNm+" with value: "+tmpVal);
            tmpOb[fldNm] = tmpVal;
        }
    });

    $http( {
        method : 'PUT',
        url : 'houses/'+$scope.myHouse.unid,
        data: JSON.stringify(tmpOb)
    })
    .success( function(data, status, headers, config){
        console.log("successfully updated house with unid: "+$scope.myHouse.unid);
    })
    .error( function(data, status, headers, config){
        //might as well say something
        console.log("poop");
    })
    .then( function(){
        $state.go('houses',{reload: true});
    });

    //console.log("Simulated PUT complete with object to send: "+JSON.stringify(tmpOb));
}

})
//...filters,directives...

```

4 - Filters

Everyone tends to like directives in [AngularJS](#) (I do too), but one of my favorite aspects of [AngularJS](#) is the [out-of-the-box Filters](#) that we get for free. This is an entire subject on its own IMO, but for now, you can see my "startFrom" custom filter; part of my custom paging mechanism for the House Collection.

```

//...config,factories,controllers...
/*
 *   Filters
 */

// we already use the limitTo filter built-in to AngularJS,
// this is a custom filter for startFrom
.filter('startFrom', function() {
    return function(input, start) {
        start = +start; //parse to int
        return input.slice(start);
    }
})
//...directives...

```

5 - Directives

Directives are the higher level "do something" definitions. Most of the [AngularJS](#) attributes or tags you write into HTML are directives. As with Filters, you can write your own Directives all you like, but some of the most useful ones come OoB.

```
//...config,factories,controllers,filters
/*
 *   Directives
 */

//This directive allows us to pass a function in on an enter key to do what we want.
.directive('ngEnter', function () {
  return function (scope, element, attrs) {
    element.bind("keydown keypress", function (event) {
      if(event.which === 13) {
        scope.$apply(function (){
          scope.$eval(attrs.ngEnter);
        });
        event.preventDefault();
      }
    });
  };
});

/**
 * A generic confirmation for risky actions.
 * Usage: Add attributes: ng-really-message="Are you sure"? ng-really-click="takeAction()" function
 */
.directive('ngReallyClick', [function() {
  return {
    restrict: 'A',
    link: function(scope, element, attrs) {
      element.bind('click', function() {
        var message = attrs.ngReallyMessage;
        if (message && confirm(message)) {
          scope.$apply(attrs.ngReallyClick);
        }
      });
    };
  }
}]);
//...nothing, just close the parenthesis to make the JS object complete then invoke with another set of paren
```

Bring It Home

That's basically it. I find that once you isolate what elements of work you have, the pieces don't have to be ugly or scary. In fact, the craziest part of my whole app was defining my clear/cancel and save functions in my House Record Controller, and that was fairly easy.

You can clone my [Git](#) repository and play around with it yourself, if you like. I recommend following the build instructions in the ReadMe included there. Until next time, 🍺 .



The Road Ahead

This past week saw the completion of a series born of a couple discussions at [IBM ConnectED](#) at the end of January, combined with my musings on application structure and realizations from having been working on a couple large [XPages](#) applications since my first adoption of the platform over three and a half years ago.

The Series

My series is imperfect and doesn't cover ideal ways of rolling an `HttpServlet`, but it does show the concept and the ability to do so within an NSF with minimal external server configuration. I'm excited for [Toby Samples to follow up to his first post on getting JAX-RS](#) up and running on [Domino](#), as it accomplishes considerably more in the realm of automation of [endpoint](#) definition, documentation, and some of the hurdles involved with my ridiculously vanilla, NSF only based approach.

This isn't a bad thing. When I started my series, only a couple proof of concept examples were out there on using a straight [javax.servlet.http.HttpServlet](#), or geared towards OSGi plugins. These are great topics, but I wanted something self-contained and more approachable to those who aren't as versed in OSGi plugin deployment. OSGi plugins have great power, I'm just not as experienced with them yet and nowhere near comfortable blogging about it.. yet. Tomorrow is always a new day 😊.

Adventure Is Out There!

I forced myself to blog about the subject and hold as few assumptions as possible. When it came to my preference to [GSON](#), I also included a version using the [IBM commons library](#). This sets up a considerable amount more of what I would like to build on. I also wanted to get the conversation away from "how do I start", to something more constructive, like "what's the best way to do this?" I think I've accomplished establishing a small base of reference from which we can all build off of. That's what I set out to do.

But Eric, You Didn't Cover ...XYZ!

My [AngularJS](#) (side of my) app effectively became a shotgun of a delivery. I was tired of talking about theory, but lots of people have covered [AngularJS](#) principles, foundations, and more; so I hit on the key points and just figured you were along for the ride 😊. Several of those people are in the [XPages](#) development community and there are *many* outside who develop [AngularJS](#) with for their [RESTful API](#). That's the beauty of this approach, you can use all kinds of universal resources to learn, as it's industry-norm and not specific to our application platform.

So no, I didn't cover everything explicitly, but to read up on how and why my [AngularJS](#) code is how it is, just check out some [AngularJS fundamentals in 60-ish minutes](#) and then on [using ui-router](#). Seriously, if you can walk yourself through an [AngularJS](#) app and [ui-router](#) principles, you're good to go. I do much too far out of the "reading level" of most [AngularJS](#) examples and it was a good demonstration of standardization of application structure in the front-end.

Let's Take a Walk

You never know what you'll find when you step out of your usual norms. Without stressing ourselves out of our comfort zones, we don't always find what we're capable of. I'm still a developer learning many things, I'm just lucky to count myself in the company of those who have a similar thirst for knowledge and willingness to share their triumphs and tribulations.



[Video link](#)

Supporting

Here are some of my blog posts which directly support the concepts used throughout the series on HTTP Servlets.

REST Consumption

RESTful APIs have seen [prolific growth in the last few years](#). Not only has it made for faster transactions between servers and clients, it's also become a great standard for server-to-server transmission of data. While many may argue in favor of [SOAP](#) or [XML-RPC](#), I'm not going to debate those merits for or against. If you are most comfortable with [XML](#), go for [SOAP](#) as 1) it uses [XML](#) and 2) is supported (to the 1.1 standard) natively in [Domino](#) as native Web Service Providers/Consumers elements. For even more on [SOAP](#) in [Domino](#), I recommend checking out [calling web services from XPages, the missing part](#) by [Fredrik Norling](#); the link to his [Notes](#) in 9 video on the topic is on the article page.

What Makes REST Good

RESTful APIs make use of [JSON](#) as their data medium; either via `application/json` or [JSONP](#). As a general rule, I find [JSON](#) syntax to be more readable by nature, as `key: value` pairs, comma separated as opposed to the encapsulated method with [XML](#), such as `<key>value</key>`. Additionally, [JSON](#) compacts a (little) bit better than [XML](#). This makes it well suited in the modern age with mobile devices lurking around every corner. For a more thorough examination of [REST](#) vs [SOAP](#), please check out [this StackOverflow answer](#) or [this article](#). Basically, [SOAP](#) (and [XML-RPC](#), its quintessential predecessor) has been around for a while and is used by many major organizations and still performs well, so it's far from dead. I just believe the same benefits of [REST](#) can apply to server consumption.

Using REST in Java

For the places I make use of server-to-server [REST](#), it's usually to stitch together some custom joining of data between multiple [Domino](#) files. This is not ideal as it makes for additional computations when they (arguably) *should* exist together already. I do regard myself as a "realist" though, so sometimes I have to acquiesce to the will of my admins and deal with the cards as they are dealt. My example use cases all sound just plain weird out of context of our existing systems, so I'm going to let you all imagine a use case of your own. Ultimately, from the consumer I wish for my call against a parameterized RESTful [endpoint](#) to produce the `JsonObject` (I'm using Google's [GSON](#) library) of the data I'm looking for. An alternate method, to keep the data independent of needing to invoke a Google [GSON](#) class would be to return the string-ified version then, if you're using [SSJS](#), perform the `toJson` or `JSON.parse` on the string.

My Sample Java Consumer

```
package com.eric.restful;

import java.net.URL;
import java.net.URLConnection;
import java.io.BufferedReader;
import com.google.gson.*;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.net.MalformedURLException;
import org.apache.commons.validator.routines.*;

/**
 * Class with a single, public, static method to provide a REST consumer
 * which returns data as a JsonObject.
 *
 * @author Eric McCormick, @edm00se
 */
public class CustRestConsumer {
    /**
     * Method for receiving HTTP JSON GET request against a RESTful URL data source.
     *
     * @param myUrIstr the URL of the REST endpoint
     * @return JsonObject containing the data from the REST response.
     * @throws IOException
     * @throws MalformedURLException
     * @throws ParseException
     */
}
```

```

public static JsonObject GetMyRestData( String myUrlStr ) throws IOException, MalformedURLException {
    JsonObject myRestData = new JsonObject();
    try{

        UrlValidator defaultValidator = new UrlValidator();
        if(defaultValidator.isValid(myUrlStr)){

            URL myUrl = new URL(myUrlStr);
            URLConnection urlCon = myUrl.openConnection();
            urlCon.setConnectTimeout(5000);
            InputStream is = urlCon.getInputStream();
            InputStreamReader isR = new InputStreamReader(is);
            BufferedReader reader = new BufferedReader(isR);
            StringBuffer buffer = new StringBuffer();
            String line = "";
            while( (line = reader.readLine()) != null ){
                buffer.append(line);
            }
            reader.close();
            JsonParser parser = new JsonParser();
            myRestData = (JsonObject) parser.parse(buffer.toString());

            return myRestData;

        }else{
            myRestData.addProperty("error", "URL failed validation by Apache Commons URL Validator");
            return myRestData;
        }
    }catch( MalformedURLException e ){
        e.printStackTrace();
        myRestData.addProperty("error", e.toString());
        return myRestData;
    }catch( IOException e ){
        e.printStackTrace();
        myRestData.addProperty("error", e.toString());
        return myRestData;
    }
}
}

```

You'll notice I've an extra feature to my `CustRestConsumer` class. On being invoked, it validates the string according to the [Apache Commons URLValidator](#). If it fails this, my class returns a `JsonObject` with property `error` set to `true` and the exception message. If it's successful, it just establishes the `URLConnection`, receives the `InputStream` and via the `StringBuffer` puts the response out into my `myRestData` `JsonObject` via the `JsonParser`. Please feel free to use and/or modify as you need and, as usual, if someone has a better way of doing it, I'd love to see it.

To See It In Action

If you're looking to play with this without generating your own use case, try out the following `XPage`. In the `beforePageLoad` phase of the `JSF` lifecycle, it imports the `Java` package, creates a string which is the URL representation of Google's feed reader service (which formats RSS, Atom, etc. into a standardized format) with the source of NPR's News RSS feed, and then puts the processed data into a `viewScope` variable. To display the entries, all I needed was to access the contents from a `Data View`, `Data Table`, or `Repeat` control and reference the object's properties by key. In my opinion, a greatly simple way of accessing the data in an `XPage`'d fashion, which reads a lot like two-way binding in client-side `JavaScript` frameworks.

```

xml version="1.0" encoding="UTF-8"?>
<xp:view
    xmlns:xp="http://www.ibm.com/xsp/core">
    <xp:this.beforePageLoad></xp:this.beforePageLoad>
    <xp:dataTable
        id="dataTable1"
        rows="30"
        var="rowData"
        indexVar="rowCount">
        <xp:this.value></xp:this.value>
    <xp:column

```

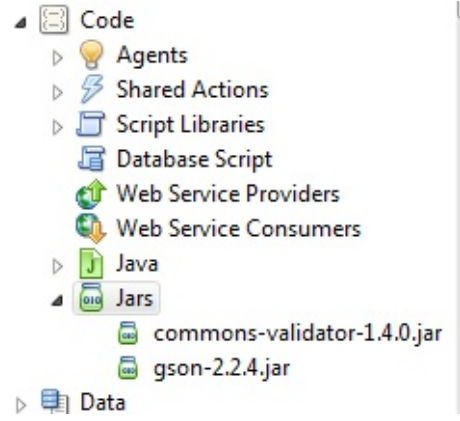
```
        id="column1">
        <xp:this.facets>
            <xp:panel
                xp:key="header"
                tagName="h1">
                <xp:link
                    escape="true"
                    id="link1">
                    <xp:this.text></xp:this.text>
                    <xp:this.value></xp:this.value>
                </xp:link>
            </xp:panel>
        </xp:this.facets>
    </xp:column>
    <xp:column
        id="column2">
        <xp:panel
            tagName="h3">
            <xp:link
                escape="true"
                id="link2"
                text="#{javascript:rowData.title}"
                value="#{javascript:rowData.link}">
            </xp:link>
        </xp:panel>
        <xp:br></xp:br>
        <xp:text
            escape="false"
            id="computedField2"
            value="#{javascript:rowData.content}">
        </xp:text>
        <xp:br></xp:br>
    </xp:column>
    <xp:column
        id="column3">
        <xp:text
            escape="true"
            id="computedField3"
            value="#{javascript:rowData.publishedDate}">
        </xp:text>
    </xp:column>
</xp:dataTable>
</xp:view>
```

[update: the fromJson call was added to this gist since original posting]

JAR Resources

I've used two [Java ARchive resources \(JARs\)](#) in my example `CustRestConsumer` class. A JAR can be opened easily, as it's essentially a zipped folder with a given structure. One of the best aspects of [Java](#) development is the ability to use and provide these self-contained archives with resources, making [Java](#) a fairly modular language, allowing you to build your own constructs on common building blocks. Essentially, why reinvent the wheel when you can use someone else's wheel class?

Per request, here's a breakdown of where you can get the two libraries I used; the [Apache Commons Validator library](#), for the URLValidator, and the [Google GSON library](#), for the ease of building and returning a JSON Object. To bring these files into an NSF,



merely import them as Code > JAR elements in Designer, as such:

For Starters

This post [relies on the previous one](#), which covers the use of a [Java](#) class to consume RESTful data. By implementing this, we were able to pull data and assign it, in the example via a `viewScope` variable, into an `xp:dataTable` element. This post is basic, but shows how powerful this can be. Some implied aspects are:

1. URL building to contain
 - the appropriate [endpoint](#)
 - URL query parameters
2. authentication via
 - same domain
 - trusted domain (or public)
 - or authentication via [basic auth](#) or otherwise
3. that you can properly handle the retrieved data

Handling the Data

Consistent formatting is key, which is why this *may* be an argument in favor of [SOAP](#); the [WSDL](#) provides action and format definition before you even execute the GET(/etc). It also highlights the importance of having your [RESTful API](#) properly documented. A case in point is the [Notes](#) View from [Domino](#) Data/Access Services. If you want to repeat the response of a categorized set of data from a View, did you remember to account for the `@category: true` entry? Remember, DAS will basically just expose the `NotesViewEntry` contents, and category entries are valid and expected.

Also, especially in [XPages](#), it helps to format/reformat your data. Since an `xp:repeat` control doesn't inherently know how to iterate a `com.google.gson.JsonObject`, we need to account for that. My post on the basics had to have its gist updated in the sample XPage to reformat the `JsonObject` into a format that the [Domino](#) flavor of [SSJS](#) could understand. I used [the fromJson method](#), which is handy and a part of [XPages](#) out of the box, but [as as noted by Tommy Valand](#) and others, needs a quick fix before you use it. For some time, I've been using [Douglas Crockford's JSON2 library](#) as an [SSJS](#) library, which achieves the same thing. So pick your poison and stick with it for consistency's sake.

Authentication

My example below uses basic HTTP authentication. This was the easiest to roll and really just comes into play if you're interacting with another server, outside of a trusted domain (or when your admins don't want to do much with existing network topology). You'll notice that I'm once again using an Apache Commons library for the Base64 encoding; isn't open source great? To get it, you'll need the [Apache Commons Codec jar](#); I'm using version 1.9.

URL Computation

As you've probably caught on by now, a similar private method could/should be used for computing the URL which your [REST](#) consumer will interact with.

My Sample Java Consumer with Basic HTTP Authentication

```
package com.eric.restful;

import java.net.URL;
import java.net.URLConnection;
import java.io.BufferedReader;
import org.apache.commons.codec.binary.Base64;
import com.google.gson.*;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.net.MalformedURLException;

/**
```

```

* Class with a single, public, static method to provide a REST consumer
* which returns data as a JsonObject, with authentication.
*
* @author Eric McCormick, @edm00se
*
*/
public class MyDataConsumer {

    /**
     * Modified copy of co.3edesign.eric.restful CustRestConsumer GetMyData
     * method. This one requires provides basic authentication with Base64
     * encoding.
     *
     * @param myUrlStr URL of the REST endpoint
     * @return JsonObject of the REST data
     */
    public JsonObject GetMyAuthenticatedRestData( String myUrlStr ) {
        JsonObject myRestData = new JsonObject();
        try{
            URL myUrl = new URL(myUrlStr);
            URLConnection urlCon = myUrl.openConnection();
            urlCon.setRequestProperty("Method", "GET");
            urlCon.setRequestProperty("Accept", "application/json");
            urlCon.setConnectTimeout(5000);
            //set the basic auth of the hashed value of the user to connect
            urlCon.addRequestProperty("Authorization", GetMyCredentials() );
            InputStream is = urlCon.getInputStream();
            InputStreamReader isR = new InputStreamReader(is);
            BufferedReader reader = new BufferedReader(isR);
            StringBuffer buffer = new StringBuffer();
            String line = "";
            while( (line = reader.readLine()) != null ){
                buffer.append(line);
            }
            reader.close();
            JsonParser parser = new JsonParser();
            myRestData = (JsonObject) parser.parse(buffer.toString());

            return myRestData;

        }catch( MalformedURLException e ){
            e.printStackTrace();
            myRestData.addProperty("error", e.toString());
            return myRestData;
        }catch( IOException e ){
            e.printStackTrace();
            myRestData.addProperty("error", e.toString());
            return myRestData;
        }
    }

    /**
     * Uses the Apache Commons codec binary Base64 package for encoding
     * of credentials, so none transmit 'in the open'.
     *
     * @return String of credentials for use with authenticated REST source
     */
    private String GetMyCredentials () {
        String rawUser = "SomeUsername";
        String rawPass = "SomePassword12345";
        String rawCred = rawUser+":"+rawPass;
        String myCred = Base64.encodeBase64String(rawCred.getBytes());
        return "Basic "+myCred;
    }

    /**
     * @param some parameters from which you build your URL source
     * @return String of the properly built URL of the source REST data
     */
    private String BuildMyURL( String param ) {
        //the string this returns is
        String base = "https://";

```

```
String srv = "my.company.com";  
//String port = ":443";  
String pth = "/api/data/collections/name/ViewName";  
return base+srv+pth+"?" + param;  
}  
}
```

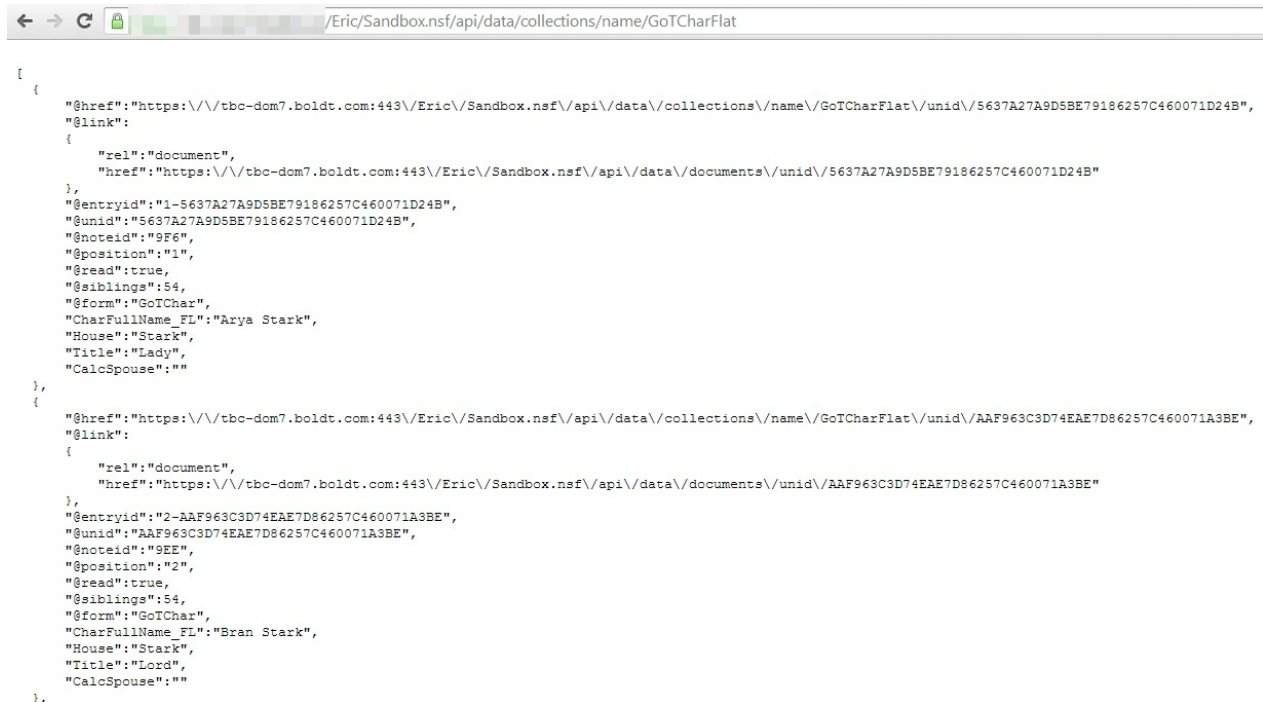

TL;DR

Impatient and want to see the code? Jump down to [my Java class](#).

[Update] I really don't recommend people use [Domino](#) Access Service (DAS), as originally outlined below, unless you're willing to accept the caveats of needing to abstract your NSF with DAS enabled to be entirely behind any external firewall; aka- not externally visible. This technique can be well used, provided you're not exposing your application to the public Internet, but still carries an element of risk in exposing full CRUD operations for the database component without any further requirements. It's handy and quick, but needs to be understood to be used properly. For the minimal effort it takes to roll an HTTP Servlet, *xe:restService* control, *xe:jsonRpc* control, or XAgent to do the same, it's really not worth opening your production environment to that potential security issue. [/Update]

What and Why?

Generating custom [JSON](#) data is, unless you're on a version of [Domino](#) server previous to 8.5.3 UP1, *virtually* unnecessary. Everything you see below can be fully replicated via the [Domino](#) Data/Access Services. The reason for that is the fact that I made use of a simple NotesView iteration pattern to generate and return the application/*json* data. The missing piece, the *whole reason why*, is on *your application requirements*. When you need [JSON](#) formatted data in a custom format due to formatting preferences or application logic needs, and it can't just be in a View, that's when this comes into play. So if you start doing what I've done, ask yourself first, can it be just in a View?



```

{
  {
    "@href": "https://tbc-dom7.boldt.com:443/Eric/Sandbox.nsf/api/data/collections/name/GoTCharFlat/uid/5637A27A9D5BE79186257C460071D24B",
    "@link": {
      "rel": "document",
      "href": "https://tbc-dom7.boldt.com:443/Eric/Sandbox.nsf/api/data/documents/uid/5637A27A9D5BE79186257C460071D24B"
    },
    "@entryid": "1-5637A27A9D5BE79186257C460071D24B",
    "@unid": "5637A27A9D5BE79186257C460071D24B",
    "@noteid": "9F6",
    "@position": "1",
    "@read": true,
    "@siblings": 54,
    "@form": "GoTChar",
    "CharFullName_FL": "Arya Stark",
    "House": "Stark",
    "Title": "Lady",
    "CalcSpouse": ""
  },
  {
    "@href": "https://tbc-dom7.boldt.com:443/Eric/Sandbox.nsf/api/data/collections/name/GoTCharFlat/uid/AAF963C3D74EAE7D86257C460071A3BE",
    "@link": {
      "rel": "document",
      "href": "https://tbc-dom7.boldt.com:443/Eric/Sandbox.nsf/api/data/documents/uid/AAF963C3D74EAE7D86257C460071A3BE"
    },
    "@entryid": "2-AAF963C3D74EAE7D86257C460071A3BE",
    "@unid": "AAF963C3D74EAE7D86257C460071A3BE",
    "@noteid": "9EE",
    "@position": "2",
    "@read": true,
    "@siblings": 54,
    "@form": "GoTChar",
    "CharFullName_FL": "Bran Stark",
    "House": "Stark",
    "Title": "Lord",
    "CalcSpouse": ""
  }
}

```

If that's the case, make sure you've turned on [Domino](#) Data Services for your NSF and the View you need. If your use case is more specific, that's what follows.

Custom JSON Data Generation

My approach here is super simple, at least as far as the [XPages](#) part goes. The only thing I'm using the XPage for is as an end point, in [XAgent](#) fashion. Seriously, it's just a hook into the [Java](#) method, have a look:

```

xml version="1.0" encoding="UTF-8"?>
<xp:view
  xmlns:xp="http://www.ibm.com/xsp/core"
  rendered="false"
  viewState="nostate">
  <xp:this.afterRenderResponse>

  </xp:this.afterRenderResponse>

```

```
XAgent. This will not render as a page, but as application/json data.
</xp:view>
```

Just invoke the fully qualified package.Class.Method() in the afterRenderResponse and you're ready to go.

XAgent-ize Your Java Method

Note: I'm assuming you know what they are, what they do, and how to implement them.

Recommended: separate the JSON data build into a method separate from the handler for the XAgent, which does the grunt work of the FacesContext interaction. I recommend this as you can then just pass the data without using an XAgent, for consumption via server-side (e.g.- extending into another class for bean or POJO use), as opposed to the client-side application logic I'm assuming. No matter how you slice it, you should know how you want to provide and consume your data.

As is the usual, we establish our handles on the FacesContext and give ourselves access to the ResponseWriter; the same as [any XAgent](#). This is how we'll be outputting data. In the ExternalContext response, we set the header information; e.g.- application/json as the content-type, no-cache so as to keep the data from becoming stale, and set the character encoding.

As my try/catch block begins, you'll note (if you're following along at home and building your Class off of mine) that there's an unused variable warning for the paramters. I left this in there so that no one else need repeat my efforts at discerning a good way of getting the URL query parameters; it took a little trial and error for me, as I hadn't seen it done at the time.

The Good Stuff

I have one single external library/JAR dependency, my good friend [com.google.gson](#). I'm only using JsonObject and JsonArray in this example, as you can see from my imports. For the full example [Java method](#), scroll down to the bottom.

So, here's the application logic portion. For my example, I iterate a View, grabbing two fields out of my *semi-improved fake names*, full name and title. The *semi-improved fake names* is a collection of basic info of Game of Thrones characters from the first two books; technically making them Song of Ice and Fire characters (for you fellow George R.R. Martin fans). The original Fake Names Database is handy for prototyping against consistently formatted sample data and is available from [xpagescheatsheet.com](#).

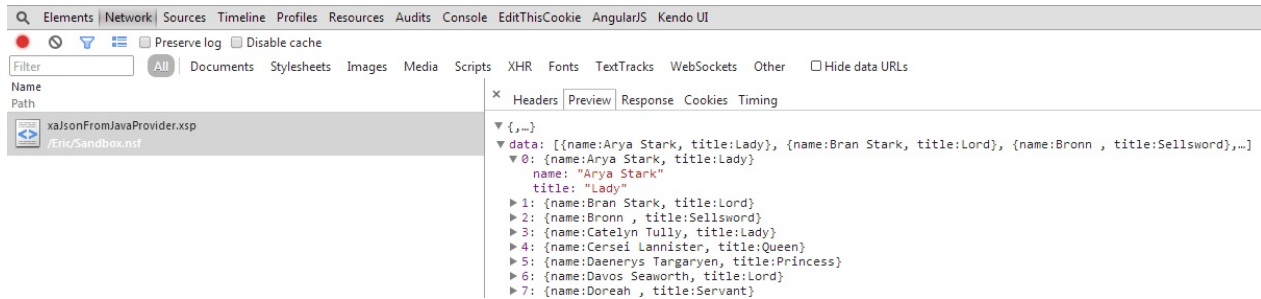
Creating a JsonObject with with the Google [GSON](#) library can be done a couple ways, in this example, you'll note I've opted to instantiate the object right away and populate the *error: true/false* and *errorMessage: message* properties at the end of the try or catch blocks, so as to always return a **valid application/json object**. As I'm iterating a number of objects with the same format of properties, I shove them into a JsonArray, which gets added under the property of *data*. This move makes it easy to segregate your client-side error-handling and/or valid response elements, all based on computed visibility of your data response of *error==true/false*.

The result gives us exactly what we're looking for.

The screenshot shows a web browser displaying a JSON response from a page titled 'Eric/Sandbox.nsf/xajsonFromJavaProvider.jsp'. The JSON data is a large array of objects, each representing a character with 'name' and 'title' fields. Below the browser, the Chrome DevTools Network tab is open, showing a GET request to the same URL. The request is successful (200 OK) and returns application/json content. The response size is 877 B (2.2 KB) and the time taken is 39 ms.

Pro Tip: Chrome DevTools

With the right tools, things get easier. Chrome's DevTools give a nice Preview tab to individual network requests. When it comes to [json](#) data, it lets us drill down nicely or switch over and view the raw response. Like this:



New to Chrome DevTools? Check out [this free primer course](#) from codeschool.com.

A Brief AngularJS Plug

In client-side [JavaScript](#), you can programmatically determine whether to take one path or another, but with [AngularJS](#), this gets much easier with [ng-show](#) and [ng-hide](#). For those used to computing the visibility property in [XPages](#), similar to

```
<xp:div rendered="#"#{javascript:myVariable==true}">
```

, this is `_mildly_` analogous; as such:

```
<div ng-show="myData.error == false">
```

Handling the Data

[Update:] [As pointed out by Paul T. Calhoun](#), a package available, if you're not looking to add the Google [GSON](#) jar, or any external library, you can implement [com.ibm.commons.util.io.json](#). The largest difference I saw was the syntax. I'm sure someone more learned could tell me about the mechanics of the two packages. To view my Class with the [IBM com.ibm.commons.util.io.json](#) library implementation, check out [this gist here](#).

Here's my method, complete with slightly rambling, but hopefully insightful to a newbie, comments.

```

package com.eric.test;

import javax.faces.context.ExternalContext;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.Map;
import lotus.domino.*;
import com.ibm.xsp.model.domino.DominoUtils;
import com.google.gson.JsonArray;
import com.google.gson.JsonObject;

/**
 * Data provider Class with a single, public, static method
 * to provide an XAgent micro-service, returning formatted
 * data as application/json.
 *
 * @author Eric McCormick, @edm00se
 */
public class DataProvider {

    /**
     * This method performs some sample actions against
     * a Domino View's Documents, reads them into a
     * JsonArray, attaches it to the JsonObject response
     * and returns it as a data response via FacesContext.

```

```

* This should be invoked as part of an XAgent.
*
* @return JsonObject sample response
* @throws IOException
*/
public static void GetMyDataAsJson() throws IOException{
    //initialize the main JsonObject for the response
    JsonObject myData = new JsonObject();
    /*
     * Here we're establishing our external context handle,
     * where we get our response writer from.
     */
    FacesContext ctx = FacesContext.getCurrentInstance();
    ExternalContext exCon = ctx.getExternalContext();
    /*
     * Using a response writer is one way of directly dumping into the response.
     * Instead, I'm returning the JsonObject.
     */
    ResponseWriter writer = ctx.getResponseWriter();
    HttpServletResponse response = (HttpServletResponse) exCon.getResponse();

    //set my content type, use a robust character encoding, and don't cache my response
    response.setContentType("application/json");
    response.setHeader("Cache-Control", "no-cache");
    response.setCharacterEncoding("utf-8");
    try {

        /*
         * This is how we can get a handle on and use any URL parameters
         * instead of the Domino SSJS param handle. Note that I check
         * for the existence of the the parameter of myKey before assigning
         * it, via ternary operator.
         */
        Map<String, Object> exConP = exCon.getRequestParameterMap();
        String myParam = (exConP.containsKey("myKey")) ? exConP.get("myKey").toString() : null;

        /*
         * Using the Domino Session class, we can get a handle on our current
         * session and interact with anything via the Java NotesDomino API.
         */
        Session s = DominoUtils.getCurrentSession();
        Database db = s.getCurrentDatabase();
        View vw = db.getView("GoTCharFlat");

        /*
         * perform any necessary business logic with the data
         */

        //creating an array of objects
        JSONArray dataAr = new JSONArray();

        /*
         * This is an example only as there are easier ways to
         * get a JSON response of a View; e.g.- Domino Data/Access Services.
         */
        Document first = vw.getFirstDocument();
        //simple View iteration of documents and adding of a given value
        while(first!=null){
            //creates current object
            JsonObject curOb = new JsonObject();
            String name = first.getItemValueString("CharFullName_FL");
            String title = first.getItemValueString("Title");
            curOb.addProperty("name", name);
            curOb.addProperty("title", title);
            //adds current object into JSONArray
            dataAr.add(curOb);

            //no OpenNTF Domino API implemented, ham fist away!
            Document tmpDoc = vw.getNextDocument(first);
            first.recycle();
            first = tmpDoc;
        }
    }
}

```

```
//wrap it up and add the JSONArray of JsonObjects to the main object
myData.add("data", dataAr);

/*
 * Business logic done, setting error to false last, so
 * if anything errors out, we'll catch it.
 */
myData.addProperty("error", false);

}catch(Exception e){
/*
 * On error, sets a boolean error value of true
 * and adds the message into the errorMessage
 * property.
 */
myData.addProperty("error", true);
myData.addProperty("errorMessage", e.toString());
System.out.println("Error with data provision method:");
System.out.println(e.toString());
}
/*
 * This will always return a fully formed JsonObject response.
 * Meaning that if there's an error, we hear about it and can
 * handle that on the client side for display while developing,
 * or logging when in production.
 *
 * Note: since we're hijacking the FacesContext response, we're
 * returning a string (not data object) into the ResponseWriter.
 * This is why the method is void. Don't worry, it's application/json.
 */
writer.write(myData.toString());
}
}
```

Application Logic

All applications require a certain logic. Even the most simple application, which is ultimately access to a data store, must have some definition of how it performs when certain events happen (what to do on a save event, what to validate and how). So, ultimately, the relevant question is to the effect of "where does my application reside?" Developing [Domino/XPages](#) applications, it manifests primarily in how you handle your server logic, interface logic, and the display layer of [XPages](#) and Custom Controls. I know it's an intuitive concept, but they don't all have to be mixed into one blended mess.

The Spaghetti Code™ Situation

If you're suffering the effects of having to support applications which implement less-than-awesome "code patterns", then you'll be well aware of the fact that the applications logic, if handled poorly, gets strewn about through all the various and potential bindings for your controls. Should it be defined in-line with every control what specific (non-default) formatting of date you want across multiple input fields and multiple design elements, you can mistakenly (more easily) wind up with several permutations, should you have to enter the patterns at different times. Note: hopefully you'll at least put the pattern into a config object for consistent referencing 😊.

So, your application logic is already residing, in part, in the client-side; assuming that you do any client-side executions. If your application is truly a collection of web forms with the only events being navigation, open, and save events, then you probably don't need this approach. If you do anything more while the browser has a page loaded, then you'll want to adopt a more unified approach, at least for larger applications.

Controller Classes Are On The Server... Already!

I know that's a rather obvious statement, but if you're sticking to a development pattern that at least includes Controller classes, then you've got your work flow actions and validation requirements are all available to you on the server. Say you want to provide your DB's CRUD operations with server-side actions and validation (to keep from cramming malformed data into your DB) via a RESTful servlet, you'll want these all in place.

This sort of implementation also lends itself to, not just validation, but 'scrubbing' of all input data. For example, say you want to use a "Rich Text Editor"-like component, such as [textAngular](#) (in contrast to implementing workarounds for the `xp:inputRichText` control; keeps markup but limits to text-only), you can ensure that all input text is properly escaped, immediately prior to your save operations. Major actions, such as sending notification emails, applying advanced permissions (Readers/Authors), and other, more intensive, operations should all occur on the server. This decreases the work load on your client/browser and keeps it nice and tidy.

Client-Side Logic

For a given page at a given state of work flow, you likely only need a smaller set of logic. The goal is to provide consistent and well formatted data back to the server. So long as your client-side controllers (a la [AngularJS controller modules](#)) know how to act at *that point in the larger work flow*, you've achieved your objective in enforcing well formatted data. It's this subset of information that makes for the "extra work" that some developers complain about, but I will always hold to the fact that it may require a *different* set of work, and that your focus as a developer *only changes* for the implementation. It's my belief that done properly, it's the same amount of "work".

Full Stack Approach

So if the work's the same, what should we do differently? As a reader of my existing posts, you're likely aware that I'm a big fan of [M-V-C development patterns](#). I'm not only a big fan of M-V-C when it comes to the multiple aspects of an application, but also across the layers a web application operates on. The [JavaScript](#) that's used with the interface layer, that runs in the user's browser, should really just be concerned with how that user interfaces with the page they're given and be independent of the server-side logic which governs things like notifications. This forced segregation helps with the [partialRefresh hell](#) which is too easy for a fresh [XPages](#) developer to (overly) rely on.

Structure is Sanity

Cross-system integration is increasingly a component of my work at my day job and keeps bringing me back to the fact that more organized code, segregated to the layers of application architecture, according to an M-V-C approach, is the way to go. My goal is to have our applications semi-independent of our database storage and db operations. This is primarily because I'm no longer the lone web developer in my day job, but one who's working with a developer who has a drastically different experience and existing skill set. I'm currently bringing him up to speed on what [Domino](#) and [XPages](#) are, but as a beginner to the [XPages](#) platform (a la myself three years ago), it's easy to blur the lines between database and application layer. This is not a major sin, but in an environment of interconnected systems, it's at least worth [persuing](#).

Put Your Code Where Your Mouth Is

Some of my in-progress efforts will help to quantify this identification of an *allotment of development work*, for comparison between beginner ("traditional"?) [XPages](#) development with [SSJS](#) libraries to contain relevant control and validation mechanisms and otherwise "vanilla" *xp*: control elements, [Java](#) bean backed [XPages](#) with controller classes, and a client ([AngularJS](#)) app with RESTful servlet implementation (utilizing those controller classes). I want to show off a more complete spectrum, highlighting the benefits of the theory ~~I've talked about~~ I hope I'm done talking about, so I can *just show you*. While this is all relatively not complex, with the example I have in mind, it is taking some time, which I seem to have increasingly less of. In the end, I'll get there, so while my blog may be quiet until I have something to share, [rest](#) assured that it is on its way.

For more on application structuring, I recommend [the recording of Jesse's MWLUG 2014 session on the subject](#); he also writes on his blog about the intricacies of [how to use his Frostillic.us framework](#). This is also (I hope) the last time I need to cite Jesse's efforts and can begin to cite my own efforts as I progress in my examples.



Intro

I'm sorry for the long post, but I can assure you that this is the shortest version of this post I drafted (over multiple days). M-V-C is a big topic, and I hope I've parsed out the reasoning in favor of its adoption.

Drink the Kool-Aid®

At MWLUG, [Jesse Gallagher](#) showed off what I believe to be the gold standard form of [XPages](#) development; [recording on YouTube](#). Jesse showed off his [Frostillic.us](#) framework (an evolved version of his [XPages Scaffolding](#)), which addresses much of what I talk about. Jesse is most of the way through [a blog series](#) on how to use said framework, which is definitely worth the read. I'm not trying to man-crush on Jesse, but the next conference I see him at, I will do my best to buy him a beer.

A Brief History

[XPages](#) is an [IBM](#) proprietary abstraction layer for [Java Server Faces](#). [Java Server Faces](#) has evolved over the years since its creation in 2004, but when [IBM](#) began adopting the [XFaces](#) (eventually renamed to [XPages](#)) platform, it initially began about 2005; [according to Wikipedia.org](#). While some of the later features of the 2.x line have been back-ported, we're definitely dealing with a unique platform.

JSF as the Foundation

When [JSF](#) first started, it set out to [accomplish a couple very specific things](#):

- create a way of handling application and UI logic, with managed state
- provide [JSP](#) custom tag library, for expressing design within a [JSP](#) page (which evolved into Facelets with [JSF 2.x](#))

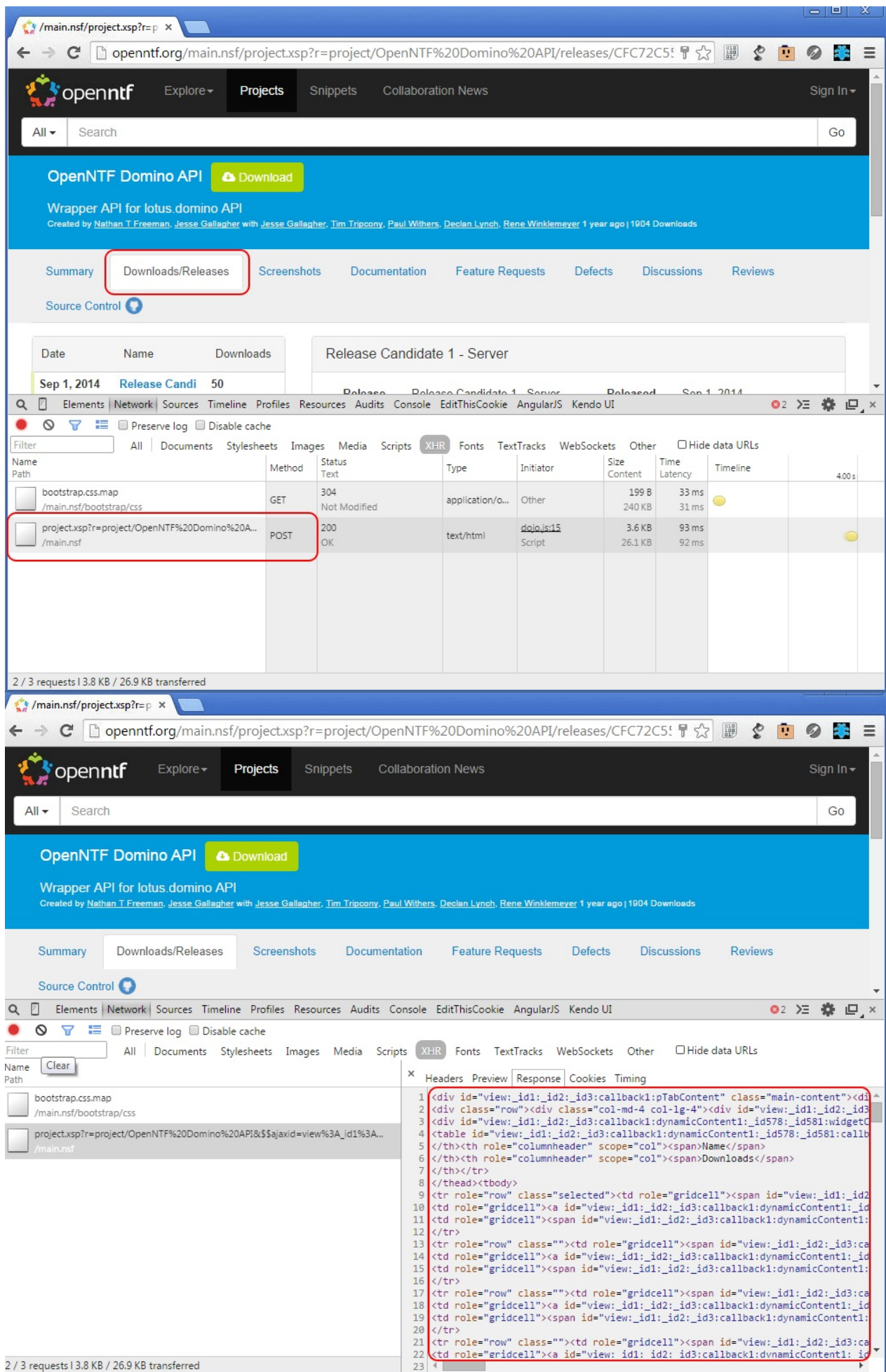
So, [JSF](#), which is geared to [Java](#) Enterprise Edition developers, is meant to abstract the handling of managed states and design elements to speed up application interface development. This is, ultimately, what [XPages](#) does; this shouldn't be surprising, as it's an [abstraction layer](#), not a replacement.

AJAX and XHRs in XPages

<Voice of David Attenborough> When "Web 2.0" was still just a catch phrase ([prior to around 1999](#)) the web cried out in anguish. Then, a champion appeared, [AJAX](#)) (Asynchronous [JavaScript](#) + [XML](#)). [AJAX](#) introduced us to the [XMLHttpRequest](#) ([XHR](#)) and brought in the ability for a programmatic, asynchronous loading of content, based on the user's interaction. The web rejoiced and new development patterns were introduced.

[XPages](#) makes use of [XHRs](#) with every [partialRefresh](#) event, usually in the form of a [POST](#). [AJAX/XHRs](#) are great, with state-ful scenarios, as you're getting "just a piece" of the whole.

Here's an example, taken from the always excellent [OpenNTF.org](#) site. Inside a project page, there are the tabs for the content pane. Selecting a tab fires a [dojo XHR POST](#) to the server, which then loads the content for the element to be changed, and the client-side [XSP](#) object loads it into the [DOM](#) (at the [ID](#) specified). [XPages](#) does this with [HTML](#) generated from the server session [Domino](#) has established for the user's interaction. You can view these interactions from most web browsers, just open up the developer tools for your browser (shown is Chrome's [DevTools](#)) and look for network events.



Why XHRs?

These XMLHttpRequests occur a lot, especially in *partialRefresh* (`refreshMode="partial"`) heavy applications. This increases the reliance on the managed state aspect of your XPage'd application and your user's in-memory session (on the server). This makes XPages quite state-ful, IMO. It's also easy, especially in complex, *partialRefresh* heavy applications, to overdo what your *partialRefresh* requires. Many in the community talk about the performance hit of *partialRefresh* es that don't use *partial execution* (`execMode="partial"`, which only evaluates what resides inside the *execId*, for the server's computation). To help automatically remove some of this bloat, [Sven Hasselbach shared a client-side JavaScript snippet](#) which assists in the reduction of the traffic. To me, this is great (good developers are lazy, right?), but still doesn't quite get us to the fundamental issues that many novices make in XPages development.

Stop Using Your XPage for Application Logic

Yes, you read that correctly. Ideally, your XPage'd content should be strictly presentation layer code. The more we jam into our XPage, be it execution blocks or SSJS libraries, the more must be computed. When we run a build on an application, it builds out your design elements to a more XSP engine friendly format, but leaving that code block in the design element makes for *spaghetti code*TM, which is far less maintainable in very large applications. Being accustomed to supporting large scale applications, I'm used to performing more searching of design elements to find what to fix, than implementing an actual fix, and that's just silly. Domino SSJS libraries are worse as, when they become large, [suffer the effect of being run-time executed code](#). Think of a very large string being parsed on-demand. There is plenty of discussion and approaches on the subject, but ultimately, in large applications, it works, but [it's inefficient](#).

M-V-C is the Way

So, be it an XPages "client" / *purely* presentation layer approach or [a\(n arguably\) more modern approach](#), we *need* to separate our application logic from our presentation layer. The bottom line is to write applications which are not state-ful, *except in the presentation layer*. In other words, ["I stand with Jesse"](#).

What to Focus On

- (Controller) controller classes, which handle how our application works (work flow, sending notifications, etc.)
- (Model) model classes, which handle how we interact with our data store ([Domino](#) document, etc.)
- (View) rendering classes, which handle how to present to the page (a dirty approach would merge this with the model classes)

The biggest development shift for many, I believe, is to adopt the controller classes as being separate from the [rest](#) of their application logic; to get it into a one-stop shopping for app logic. For more on why M-V-C in particular is best, I'll leave that to those who have already done the work. Jesse Gallagher has gone through his series on [XPages MVC Experiments](#) and Gary Glickman has a great series on ["Rethinking the Approach to XPage Development"](#).

The XPages Approach

Managed Java Beans, with [Expression Language \(EL\)](#) bindings. Seriously, just have managed beans for your respective M-V-C classes and you invoke your entire application by EL. For more demonstration of this, see the [video on YouTube of Jesse's "Building an Structured App with XPages Scaffolding"](#); have I plugged that enough yet?

You will also notice that this sounds easy and the truth is, it is much easier when you don't have to search through your code, worry about where to put validation (in the control tag? on submit of the form?), or how you can interface your application to external sources/applications.

The Modern Web Approach

[Mark Roden](#) has been tackling the subject of [Angular.js in XPages](#). What this really shows is the flexibility of segregated application logic on the server, accessed via [a super-heroic \(client-side\) JavaScript framework](#), which is RESTful (REpresentational State Transfer, without state defined, except in the network request via end point) by nature. This approach has great appeal as it performs great on mobile devices and desktop browsers alike. I think Mark's session at MWLUG should have been named "Write Once, Run Everywhere" (as opposed to "... Anywhere"), as it demonstrated the flexibility of this M-V-C approach with the server.

The reason I regard this as the "modern web" approach is that most not-**IBM** specific development that make great use of M-V-C practices are using client-side frameworks to do the serious grunt work with validation on the server via **RESTful API**. If it makes you feel better, you can consider this an "alternate" approach, but this maps to what the majority of the modern (and awesome!) web development world is doing.

In Summary

I hope you see how this all maps in the progression of [the types of XPages development](#). If nothing else, I hope this post may give you a good number of ideas with which to try and, hopefully, improve how you build your applications to make things easier on yourself, as a developer. As always, best of luck, and please, discuss!

EDIT]

Regarding XHRs

XMLHttpRequests encompass nearly every partial refresh under the sun. I loosely describe [AJAX](#) calls (and [XPages' dojo.xhrPost](#) calls) to encompass "fat XHRs", or XHRs which include *markup* in their response. This is a terrible way of doing business, as we as developers ought to *lighten the load* a bit in the age of mobile devices and cellular connections. I touched on this when I originally wrote the post, but thought that it could use some clarification up front. So, transport data for partials, not markup.

[/EDIT]

REST is Best

Recently I became a father. It's pretty awesome. I've got a daughter who gives me some pretty good smiles and other funny faces, so I've always got some good motivation to go home at the end of the day. This also means I've gone through some birthing classes in recent history. So consider this post's title to be a play on words, regarding the interpretation of infant feeding. You know, the old adage of "REST> is best" (unless contraindicated by medical or other conditions).

Why is [AJAX](#) Not Good Enough?

My last post, [How to Bore Your Audience](#), spent a bit of time on the "big picture", for the structure of modern and awesome XPage applications. It also outlined my general distaste for overly large [AJAX](#) calls (specifically [dojo](#) xhrPost) when a simpler method (at least an xhrGet) would suffice. [AJAX](#) can return [JSON](#) data, though it is, by default, Asynchronous JS and [XML](#). So what [AJAX](#) really is, if we're data format agnostic, is really just a programmatic network call to return a data payload of *something*.

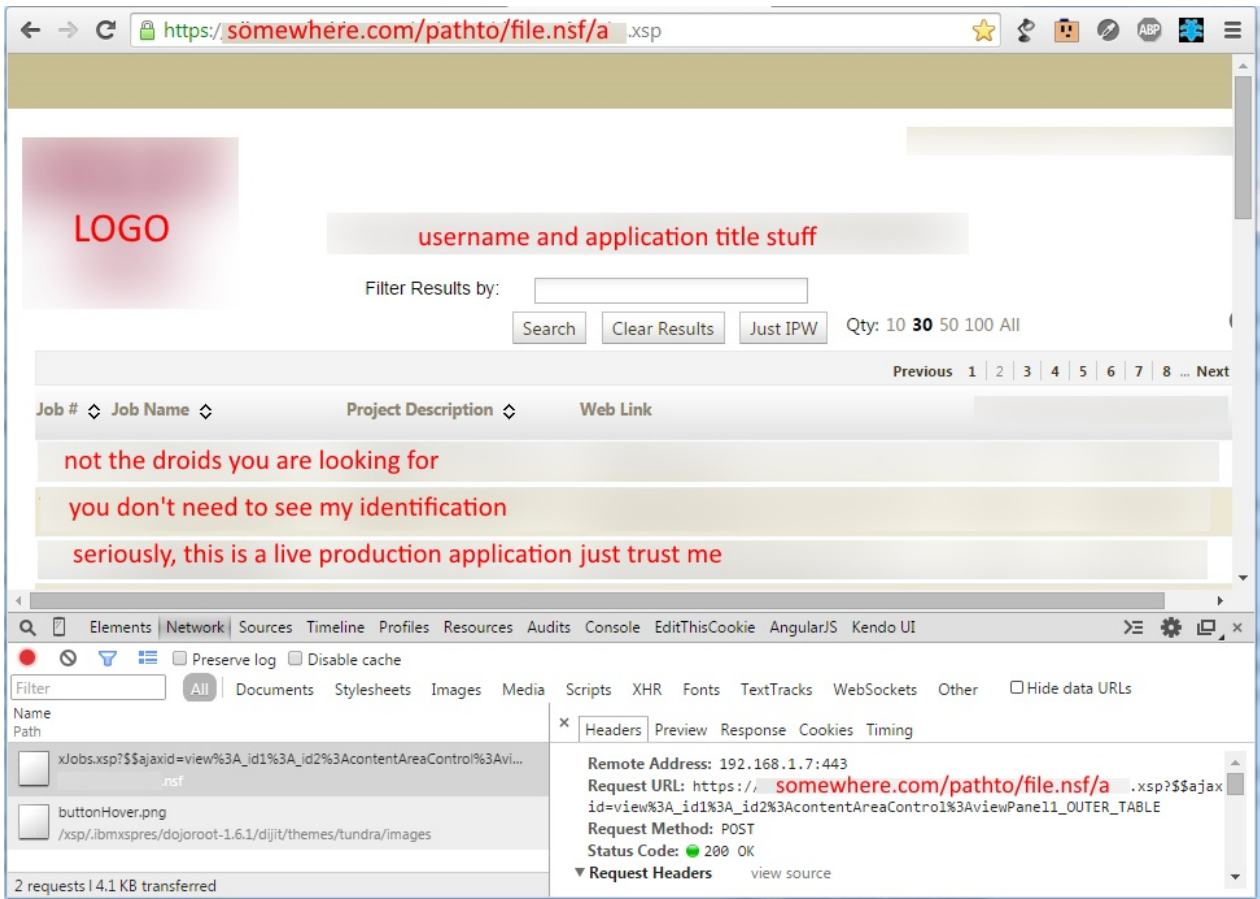
[XPages](#) does this by that [dojo](#) xhrPost call to call out where (the *partialRefresh* id) to inject/replace the newly returned data. This happens to be (usually) HTML, a [Dojo](#) data store (in the event of an *xp:restService* control, depending on your properties), and more (like if you refresh an *xp:scriptBlock*). This works, but when you keep your application logic on the server (and I suggest you do), that means you're often sending increasing amounts of information back and forth, in a partial(Refresh) capacity.

REST is Lean

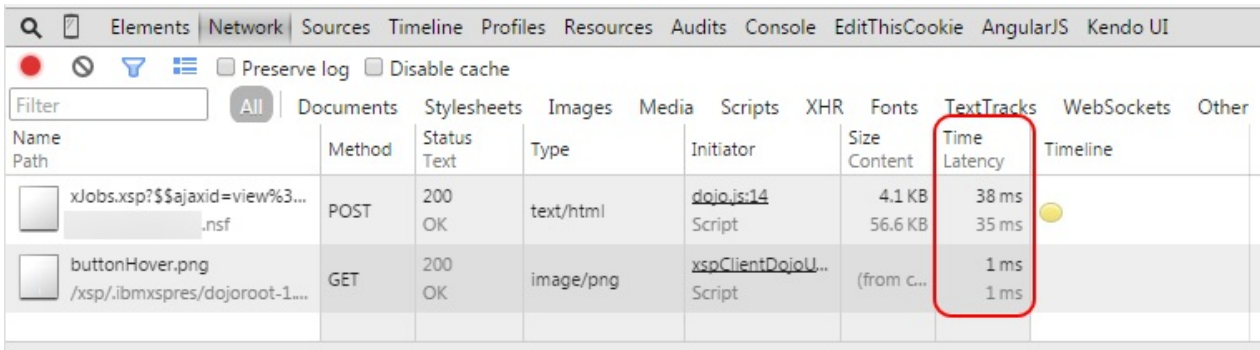
Having recently read Paul Akers' book, [2 Second Lean](#), and having seen him speak in-person, I can honestly say that when I look at a process, I think "I see waste" and I want to eliminate it. This is a part of what we do as developers, and I'm sure is intuitive to you, but we must always strive for the path of least resistance in our applications. It makes for better application structure and better user experiences.

Without the need for an in-memory session on the server, we no longer require a session "[state](#)"#Programstate). *To get the data we need, we have to formulate what to request in the client, using the browser's JavaScript, and then execute the call and handle its receipt. Many of the modern JavaScript frameworks out there, like my beloved AngularJS, automate this process. To do so, they use a combination of http event handlers (\$http in Angular) and callback functions. In the XPages world, think of the CSJS event functions for _onComplete and onError (etc.) which we use in xp:eventHandler tags.*

Let's compare a simple thing in [XPages](#). Using the stock *xp:viewPanel*, *xp:pager*, with the *partialRefresh* option, this is a fairly normal way for an XPage developer to put a View into an XPage. This is also my hallmark argument against this variety of implementation, for such a simple task. Here's what happens when I hit "Next" in the pager:



When we execute these [AJAX](#) calls, it takes time and processing effort (both for the server and the client/browser). Here's what I mean:



The above doesn't show a whole lot of time elapsing, only about 38ms. It also shows a hover state being fetched; I didn't even plan on that (and is an argument against [Dojo](#), IMO; I mean, lazy loading images for button styles!?). I can also tell you that that server is having a good day and isn't refreshing anything more than the `xp:viewPanel` for this page (so less intense computations). The application above has been re-developed, as a case study (with which I've been able to sell to my management and direct my development efforts accordingly), into a Bootstrap 3 with [AngularJS](#) application. Here's what happens when I perform the same paging task in the [Angular](#) version of this app. Apologies for the reduction in quality with the gif and redaction of company-specific information.

No network requests during paging, it's that cool. What's happening? It's behaving as a modern web application; a single page app, in fact, but I'll get to some of those specifics in a moment. Here's the same page again, with live full-text searching, across all fields (keys, as in [JSON](#) key: value pair, you can also filter by key) in the data array.

So why is [REST](#) lean? [REST](#) means a less cluttered network request, performed less frequently. This also comes down to your implementation of it, which is why I'm showing off [Angular](#), which plays to a [RESTful API](#)'s strengths. The idea is to invoke *just what you need* from the server, at the state of what you're looking for, [HATEOAS style](#). You still have to load a page with a [JavaScript](#) library to know what to invoke, but you should reduce as much as possible afterwards.

SPAs and Application Structure

You knew I was going to bring up application structure, didn't you? The dichotomy of the server-side application logic and the client-side application logic must be apparent now. It's precisely why, when [Mark Roden](#) gave his [Write Once, Run Anywhere: Angular.js in XPages](#) session at MWLUG, he admitted (begrudgingly, I might add) that to properly build a larger application, a developer would want to enforce application and work flow validation on the server; aka- "everybody needs a Toby". This would be done by writing a custom servlet or [REST](#) implementation, which would validate before directly committing into a [Domino](#) document. If your application is simple and your field data is strictly textual and you trust your users to not put bogus data into their network POST or PUT operations, DDS is great.

Domino Data Services

This is the biggest downside of the [Domino](#) Data Service in my opinion. The [Domino](#) Data Service gives us the ability to perform the CRUD operations against [Domino](#) Documents and Views, but there's no `computeWithForm`, which would give us at least a way of invoking an agent on save. But, it's better than nothing. So, would a developer benefit from structuring their application with data models and controller classes? Absolutely! In fact, you might think there was a reason I wrote [that long winded post last](#) before this one ;-).

Summarizing

As you can see, M-V-C is a thing. It's great idea for your server-side application logic and there are a great many awesome M-V-C client-side frameworks (like [Angular](#)) that can help you expedite your front-end logic. So please, let's build better apps. [REST](#) can get us there with lighter weight network requests and in-browser processing of data and application logic. We can reduce our network calls, sizes of data transferred, and made our performance response time nearly negligible (limited only to the time it takes the client-side JS code to perform the rebuild of the HTML and the initial page load).



No silly Keanu, it just might keep us sane.

Intro

This is a quick post, covering something I overheard while at MWLUG and comes back to some application architectural principles which I have a bit of a passion for. Read on at your own peril 😊

[Update] I added a bit from a tweet by Tony McGuckin about the [XPages](#) runtime's components. [/Update]

Back to the Grind After MWLUG

There's an interesting slump I experience after getting back from a conference. Not only do I get to clear out of the mountain of things that seem to crop up only while I'm out of the office, but it seems that I'm able to come away from such a gathering with other developers with lots of great ideas for both my growth as a developer but also for what I'm able to accomplish for my company and its users that it's a nearly unbearable amount to be able to extract from my own head, but I try. For this last week though, I just worked the tasks at hand.

A Funny Thing

While at MWLUG, I overheard someone make a reference to people "not liking POST" as an HTTP request. I'm fairly certain it may have been said in jest as [a blog post I wrote previously](#) talked about "classical" [AJAX](#)-y requests containing markup (specifically analogous to an [XPages](#) partial refresh event, which fires a `dojo.xhrPost` that returns the HTML content of the selected `refreshId` and re-injects the content to the page). So here's a reminder to all, [XPages](#) gives us:

- an application runtime ([JSF](#))
- a striped database (though for performance it's best to separate it)
- with a greatly convenient security model (that makes it *very* easy to map roles and groups across applications)
- and a bunch of OoB (out of the box) controls that aid in [RAD](#) and have some excellent hooks to automate a large number of functions ([ExtLib Relational controls](#), for instance)

[Update] As Tony McGuckin pointed out on Twitter, there's more to the [XPages](#) runtime than just the [JSF](#) runtime components. Since it's a larger list than my couple of bullet points, I'll let the tweet and screen shot from the liked [XPages](#) Masterclass video do the talking.

The Tweet

[@flinden68](#) Congrats Frank! +1 [#XPages](#) includes features not even in JSF2.2 - some info at 1min 22secs in <https://t.co/Yzzvf2BhYx> [#GoodPoint](#)

— Tony McGuckin (@tonymcguckin) [September 1, 2015](#)

The Screen Shot




XPages and the JavaServer Faces Framework

- **JavaServer Faces® Reference Implementation (aka JSF RI)**
 - Provides a Component based development model and framework
 - Provides a Stateful runtime for Server-Side User-Interfaces / Java Applications
 - Provides a Server-Side EL Scripting / Binding Model
 - Provides an Extensible development framework
 - Part of the J2EE specification / stack
 - <http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>

- **XPages extends the JSF RI Implementation**
 - Provides Specialized Controls and DataSources for N/D application development
 - Provides Extended and Optimized Component / View Event Model
 - Tightly integrated with Client-Side and Server-side Scripting Models
 - Specialized Partial Refresh and Partial Execution Models
 - Provides Server-Side EL, JavaScript, and XPath Scripting / Binding Model
 - Specially adapted for N/D application development (Domino Object Model)
 - Provides Client-Side JavaScript Scripting Model
 - Provides Specialized Stateful Runtime for highly optimized application behavior

© 2013 IBM Corporation

[Update]

My Beef With Transporting Markup

My list above of what [XPages](#) provides us lists the controls that we most often associate as being "XPages" (specifically the design elements of [XPages](#) and Custom Controls, along with the *xp* and *xe* controls) at the end. My approach is opinionated in the absolute segregation of front-end and back-end, but it uses the application runtime provided by [XPages](#) (and its [JSF](#) implementation) to provide a great experience with excellent results. The OoB elements give us *one way* of doing things, so since it bugs me, I'm going a different way, without issue.

Transporting markup in our post-page load XHRs is inefficient; regardless of whether we call them [AJAX](#) calls, partial refreshes, or whatever. That's my bottom line. I don't think POST as an HTTP method is inferior (they're just HTTP methods), but to add to a request's body content, just so we can get a small data update in the screen, when we could otherwise have our data providing the same response just in how we build our URI from a simple GET, the logic seems clear to me.

What to Do About It

As my demo application has been showing, I've been going the direction of a front-end heavy app ([AngularJS](#) app in the UI) with HTTP servlet [RESTful API](#) driven data access. This doesn't necessarily need an [XPages](#) design element (root XPage and Custom Control) since it can/is served from the WebContent directory. For any still wondering, yes I do continue development on and maintain applications that aren't entirely "my way" when it comes to what are already in use. I've refactored core code when able (I've seen some scary [SSJS](#) libraries and vanquished as many beasts as I've been able to) because I firmly believe in keeping core business logic separate from UI logic.

That being said, my advice to any in a "normal" [XPages](#) context is **embrace the JSON-RPC** control and [Java](#) beans (managed or POJO). If your business logic is driven inside a controller class, it can just as easily be invoked from a bean or from a servlet (or from [SSJS](#) inside a [JSON-RPC](#)'s method script). If [Java](#) isn't your thing (if you're still learning, and you should embrace your JEE stack), [the](#)

[JSON-RPC control](#) lets you accomplish most of the same task of exposing server-side operations and logic, with minimal overhead, to the client-side (browser) for [CSJS](#) access; it's a win-win either way.

In Case You Missed It...

My last post, [recapping my session at MWLUG](#), managed to get missed in my first tweet of the link. I updated the previous post (announcing my session at MWLUG) with a link at the top, but some may have missed it. Please check it out, there's a link to [my slide deck](#), [GitHub repository](#) with my configs, screen shots of my configs in use, and (you guessed it) my slide deck in PDF and PPTX formats.

Single Page Applications

In my previous posts, you'll have noticed that I've referenced Single Page Applications (SPAs) and how they relate to assisting in building better web applications. Here I'm going to try and break down what an SPA is and isn't and show what we can learn to apply to any web application for better development practices. Ultimately, each application is unique and requires its own implementation, my intention is to help show off some of what makes an SPA great to give others ideas in their web applications, regardless of implementation. This post is meant to be more of a reference, with other topics talked about "coming to a blog near you" soon.

What They Are

Excerpts from the [Wikipedia page on Single-page applications](#).

- "...a web application or site which fits in a single web page with the goal of providing a more fluid user experience..."
- "...either all necessary code - HTML, [JavaScript](#), and CSS - is retrieved with a single page load, or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions"
- "...often involves dynamic communication with the web server behind the scenes"

So, what I believe a single-page structured application does well is containing the initial application logic, and its methods for other partial elements (html templates, [json](#) data, etc) without requiring additional full-page loads (from the browser's perspective). This eliminates some of the overhead for always loading certain images and stylesheets while keeping the focus of what network traffic exists, after the initial page load, being only what's necessary (the data or html partials).

What They Are Not

- the only way to build modern web applications
- an argument against having multi-page web applications (let's face it, you can't cram it all in every time)
- perfect for every application
- [necessarily new](#) (some of the mechanics are, but straight up web pages with JS manipulations have existed ever since [JavaScript](#) was implemented); making SPAs more of a design strategy
- an application strategy that requires "less work" (it just shifts where your attention goes)

A Brief Anatomy

As you probably know by now, I'm a big fan of [AngularJS](#). It makes a lot of the client-side application development easier than you might think. The example file is the `app/index.html` file from the [angular-seed project on GitHub](#).

```
<!DOCTYPE html>
<!--[if lt IE 7]>      <html lang="en" ng-app="myApp" class="no-js lt-ie9 lt-ie8 lt-ie7"> <
<!--[if IE 7]>        <html lang="en" ng-app="myApp" class="no-js lt-ie9 lt-ie8"> <![endif
<!--[if IE 8]>        <html lang="en" ng-app="myApp" class="no-js lt-ie9"> <![endif]-->
<!--[if gt IE 8]><!--> <html lang="en" ng-app="myApp" class="no-js"> <!--<![endif]-->
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <title>My AngularJS App</title>
  <meta name="description" content="">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="stylesheet" href="bower_components/html5-boilerplate/dist/css/normalize.css">
  <link rel="stylesheet" href="bower_components/html5-boilerplate/dist/css/main.css">
  <link rel="stylesheet" href="app.css">
  <script src="bower_components/html5-boilerplate/dist/js/vendor/modernizr-2.8.3.min.js"></
</head>
<body>
  <ul class="menu">
    <li><a href="#!/view1">view1</a></li>
```

```

    <li><a href="#!/view2">view2</a></li>
  </ul>

  <!--[if lt IE 7]>
    <p class="browsehappyy">You are using an <strong>outdated</strong> browser. Please <a
  <![endif]-->

  <div ng-view></div>

  <div>Angular seed app: v<span app-version></span></div>

  <!-- In production use:
  <script src="//ajax.googleapis.com/ajax/libs/angularjs/x.x.x/angular.min.js"></script>
  -->
  <script src="bower_components/angular/angular.js"></script>
  <script src="bower_components/angular-route/angular-route.js"></script>
  <script src="app.js"></script>
  <script src="view1/view1.js"></script>
  <script src="view2/view2.js"></script>
  <script src="components/version/version.js"></script>
  <script src="components/version/version-directive.js"></script>
  <script src="components/version/interpolate-filter.js"></script>
</body>
</html>

```

This Gist brought to you by [gist-it](#).

[index.html](#) [view raw](#)

The general progression is:

- load the file with the usual fixes up front (old IE conditionals, IE=edge) and other meta tags (viewport)
- load the structural elements, such as framework CSS, app CSS, and Modernizr
- load the body (structural elements)
- you'll note this one has a div with the ng-view directive (in the example file, as an attribute of the content div), that's how [Angular](#) does its partial html injection (e.g.- **content goes here**)
- end it all by then loading the JS framework library, then
- your application script
- and your partial html files (though those can be injected, with the controllers, [via \\$routeProvider](#))
- any universal custom filters, directives, etc

This lets the page start all its loading all the needed elements before the client starts modifying its contents. Not everyone does it this way, but it can help quite a bit when your client-side app performs a lot of initialization work.

When a client-side framework like [Angular](#) detects changes (like in the partial html content being triggered), it then grabs the necessary controlling code and logic and begins to modify the DOM to suit its needs. That's what that [ng-view directive](#) does.

The Biggest Pieces

How a page loads in the web browser is the ultimate destination and, by proxy, make or break end point for any web application. The user's experience is truest and only *real* common denominator for how a user interacts with the server. The server can perform amazingly and do great things, but if the application is consumed in a browser that is old or outdated (looking at you, old versions of IE and the users that run them), or the loading of that page is just ridiculously network call heavy (when it could be avoided), then the user suffers, which means the application suffers.

A lot of front-end developers spend quite a bit of time on the below topics. Basically, if you open up and make use of the [Page Speed Insights extension for Chrome](#) or [app.telemetry for Firefox](#), you can find a number of good statistics and recommendations for how to speed up your web app and "milk it for what it's worth". When I first saw this done, it felt like someone was attempting to divine the aether of the Internet, but there's a lot to be said for end user performance.

IBM has thankfully thought of some of these concepts and the XSP properties let us set a few things to help with UX, including runtime optimized JS and CSS, compressing resources files (CSS and [Dojo](#)), and [other tasty tidbits](#) that I'm sure other people know more about than I. In fact, one of the easiest ways to improve a partial refresh in [XPages](#) is to better manage [how much you refresh](#) to make for a better *partialRefresh* experience.

Resource Aggregation

This is definitely one where IBM has tried to give us something that the front-end world has been big on, of late. combination of (as much/many) static resources as possible, along with *g*zipping for even lighter footprint while in transmission, gives the browser a bit of a faster load. With *g*zip'd content, it still takes a decompression on the browser's part once that's done, but hey, I've been focusing on network requests/responses. In XSP Properties, just set *xsp.resources.aggregate* to true in [Domino](#) 8.5.3 and up.

Cache Control

Something I haven't figured out how to do yet in [Domino](#) is to regulate the content cache for certain resources. I'm interested in particular in things like CSS and images. As I try to make life easier on myself, I tend to host most of the elements I'm interested in caching (ideally for about 30 days) reside in my `..\Domino\data\domino\html\` path, for what I don't use from a [CDN](#) (and CDN fall-back copies). I also don't know how this interacts with the resource aggregation property (*xsp.resources.aggregate* see above). I'm also uncertain about how the use of the *xsp.expires.global* property compares against server hosted resources (`..\Domino\data\domino\html`).

Lazy Loading

I've [previously talked](#) about how I have a dislike for (at least 1.6's and/or older versions of) [Dojo's](#) lazy loading of button styles. Aside: I also think that not using *dijit.form.Button* would take care of that, but that's not my call, sadly. I also mentioned how some of this [has improved with Dojo's AMD](#) over time and multiple releases. The fact of the matter is: I want to control what and how much is transported over the network connection at any given request; and *I'm not alone*.

This is such a big topic to me, because I spend a lot of my time and development effort building, extending, and maintaining a (very) large application for my company. This application, which has turned into more of a platform, spans every individual location of ours across the country. This application calls home to our corporate servers making line quality a bit of a topic as well. Server locations,

The Way Forward

So how can we make better applications? My theory is that we need every single tool in the tool box. *Traditional XPages development* doesn't go away, not in the least. In fact, I look at [Angular](#) and other client-side frameworks as a tool to expand on what we already do.

The XPages Way

There is no one, true gold standard "XPages way" of creating an application. It's one of the strengths and weaknesses of [XPages](#), simultaneously. First, the weaknesses. [XPages](#) lets us dump code virtually everywhere, which is great, except for the potential of [spaghetti code](#)TM. But as a developer advances in both ability and development practices, the path eventually seems to lead to a combination of managed beans and plugins. These make for some great, business grade applications that have strength in utility and capability.

Leading us to [XPages'](#) strengths, the ability to adapt and adopt newer (yes, it's debatable) ways of development with our platform.

Want More [AngularJS](#)?

I recommend checking out [Dan Wahlin's AngularJS Fundamentals in 60-ish Minutes](#). It's a good overview and he can probably sell you on the concepts a bit better than myself.

Want More on SPAs?

The hall mark for comparison of front-end [JavaScript](#) frameworks has tended of late towards [TodoMVC](#). This is geared towards assisting a person to select a (client-side) MV* framework. It also shows the same, simple but illustrative application in use on numerous frameworks. It's also an SPA.

I hope you (are starting to?) see how a more robust front-end application logic can compliment your applications. No one development style or individual tool can "do it all", but why not have another tool that can help you do your job? It never hurts to expand the skill set. Even if it's not the best tool for you right now, it's worth getting a little familiar with some client-side frameworks, even if it just gives you new or different, hopefully better, ideas in your application development.

Related

Blog posts related to the series on HTTP Servlets, though not directly supporting and can stand on their own merit.

For Starters

¡Feliz Cinco de Mayo!



It's been a little longer to get to this installment of my [Saga of Servlets series](#), but I guess that happens when things like the day job pick up with trouble shooting server issues and family life all seem to get in the way.

Intro

This isn't the most "developer sexy" topic, but I hope is worthwhile (and something I promised would be in this series).

The intention of this post to tackle the concept of what an *XAgent* is (I'll be brief) and why our use of them can be substituted (in most cases) with an *HttpServlet*. There's a caveat to this, covered below, and for all intents and purposes, I'll be using *HttpServlet* interchangeably with *DesignerFacesServlet*; the implementation of which I use being [Jesse Gallagher's AbstractXSPServlet](#).

Note

My intentions throughout this series have included to avoid any specific frameworks for building out an *HttpServlet* and/or [RESTful API](#), hence the pure [Java](#) implementation, NSF-level implementation (making it easily accessible before getting into OSGi *HttpServlets*), and not being so keen on [GSON](#) as to put off people who can accomplish the same thing in the [IBM](#) commons library. That being said, the intention of *this post* is to bring us back to some common ground with other [Java](#) EE developers in how we perform some tasks; so if you're interested in such things (as I am!), please read [Jesse Gallagher's post on using JAX-RS](#) or [Toby Samples' blog](#), as he's kicking off a series on [using JAX-RS with Domino in an OSGi plugin](#).

XAgents

Ultimately, the purpose is to provide a data response after performing some computation, over HTTP (effectively the same steps in an *HttpServlet*); whether that's a binary file like a PDF or web-consuable data response. *XAgents* provide an *XPages* developer an easy way of creating an [endpoint](#), the *XPage* name, with which we can easily hook into the data response by setting it non-rendered and [overriding the HttpServletResponse](#) (and unless it's a response we don't need to persist state with, [setting the xp:view attribute viewState](#) to "nostate").

XAgents are relatively easy to create, especially for a less experienced *XPages* developer. My experiences in life have taught me that "easier" doesn't always translate to "better", but an *XAgent* is handy, convenient, and easy to get started with.

How Much Overhead is in an XAgent?

I wish I had the time to invest in some benchmark comparisons. This may be something I revisit, as it will probably bug me until I have some actual data. In any case, the main idea here is that the full [JSF](#) lifecycle is invoked, causing a more-than-needed increase in server processing. An *HttpServlet* will take a request, process as needed for a valid response (stateless, if you go the RESTful route), and kick out a data response. An *XAgent* can do the same, but all the moving parts of [JSF](#) are still invoked.

In lieu of some recorded tests to back this up, I'm going to link you to [a blog post by Karsten Lehmann talking about XAgent bottlenecks](#) and an excerpt here that should sum things up nicely.

The consequence is that you should think twice about your [XPages](#) application architecture, if you have many concurrent HTTP requests or if some of them take a lot of time to be processed. An *XAgent* may be the easiest solution to deploy, but may not produce the best user experience in all cases.

[Insert Data to Back Up Reasoning Here]

When to Keep Using an XAgent

The caveat to using an *HttpServlet* is the need for reliable *sessionAsSigner* access. While I believe this is feasible at a conceptual level (especially inside an NSF), it would be lacking in the context of an OSGi plugin, as there would be no actual design element. I've tried to read up on and ask around on this subject, but the most I've found is [an old question on OpenNTF](#) and some confusing talk from Jesse Gallagher about ClassLoaders and the underlying [Domino C API](#). Talking with Jesse about these things make it sound like a really good idea for me to take his word on it 😊.

[Update] I've had some good success resolving *sessionAsSigner* via *ExtLibUtil.getCurrentSessionAsSigner()*, which makes my above comment a bit less pressing. [/Update]

When you do have to use an *XAgent*, I recommend having a single line of invocation in your before/after ...RenderResponse. This should invoke the fully qualified package.class.Method with a parameter being passed as a handle to *sessionAsSigner*. This keeps things clean and simple, and your class will be easily maintained in a consistent fashion to any *HttpServlet* you create. For example:

```
<xp:this.afterRenderResponse>  
  
</xp:this.afterRenderResponse>
```

Why Should I Care?

While we may be using a uniquely abstracted variant of an *HttpServlet*, by building our logic as an *HttpServlet* as opposed to an equivalent *XAgent* (especially in [Domino/XPage's SSJS](#)), we create our data service in an industry normal fashion. If this on top of the performance increase doesn't sell it for you, I'm not sure what will.

Summary

If we want to be more of a [Java EE](#) developer, which is the industry equivalent norm for an [XPages](#) developer (by my interpretation), then we should embrace the more industry norm practices. In this case, it also means we drop some of the unnecessary [JSF](#) "baggage" from the process of merely handling a data response.

The final part of this series will cover some of the client-side application in using the the *HttpServlet* we set up in the [Round House Kick Tour of data handling](#). It may come soon, if I can keep my spawning of non-series post ideas in check.

[Update] This topic is so awesome I turned it into [a video for Notes in 9, check it out.](#) [/Update]

- [Intro](#)
- [A Segregated Approach for the Front-End](#)
- [A Sample Data Set of JSON](#)
- [Json-Server](#)
- [Examples](#)
- [One Last Thing](#)

Intro

It's no secret I'm a strong advocate for segregated application design practices. In my quest to "make everything work the way I want it", I've chosen a front-end framework that my ui-level application is written in, structured my primary application layer into RESTful [Java HTTPServlets](#) (*DesignerFacesServlets*, specifically), and life is generally good. My endeavors in this area are for a few, specific reasons, namely:

- keep my development efforts focused (e.g.- identifying whether a problem is front-end or back-end can greatly speed up trouble shooting)
- focus on data as a service (which makes it easily consumed by other systems)
- make more easily documented applications
- make more easily tested applications
- make applications more easily outsourced

That last one is probably foreign to a lot of people, but as one of two web developers on my company's organic staff (and the only [Domino/XPages](#) developer), this means I want to unify efforts across our application platforms and also make things easier to plug a contract developer into. It speeds up their efforts, makes it easier to plug into [source control](#) (for not just tracking, but also support purposes), and overall will aid my sanity.

A Segregated Approach for the Front-End

I've spent a considerable amount of time covering the back-end approach that I'm migrating to; just [look at all but the last two-part piece in my series on Java HTTPServlets with XPages](#). But what if we hire out some work to be done on the front-end; wouldn't it be nice for that developer to work on *only* on that front-end, without any need for other setup? I think it would and this ought to outline how this can be achieved fairly quickly.

A Sample Data Set of JSON

Since I'm interacting with my data via a RESTful *HTTPServlet*, this makes things rather easy for me to create one of the required parts. I need to have a sample set of data, which I can interact with to confirm/deny my interactions are well formed and test with some form of data. In this case, I'm assuming this is for making changes to an existing application, but if it's a new one, someone would want to sit down and define a sample set of data; this is normal operation for myself and, I expect, most developers. For my use case, I copied the network response from my `.../xsp/<collectionName> GET` call into a `db.json` (I'm using `housesDB.json`, except for my GIFs below, which are another source, but follow the same structure) file. This will need one minor change, and that's to wrap the data array into a slightly different format, so that `json-server` can read it correctly (I stripped out my usual *request* block with any params and the *error* true/false, for simplicity).

After your minor transforms, I recommend sanity checking your data with a tool like [jsonlint.com](#). You'll note my collection is a member of an *xsp* object (to route similar to my production path). Here's one I prepared earlier:

```
{
  "houses": [
    {
      "region": "Vale of Arryn",
      "unid": "F3C2CE924605412888257E0000128173",
      "seat": "The Eyrie (summer), Gates of the Moon (winter)",
      "heir": "Harrold Hardyng",
      "title": "King of Mountain and Vale (formerly), Warden of the East Lord of the Eyrie Defender of the Vale",
      "overlord": "House Baratheon",
```

```

"words": "As High as Honor",
"name": "Arryn",
"description": "House Arryn of the Eyrie is one of the Great Houses of Westeros, and is the principal noble house",
"coatOfArms": "Azure, upon a bezant argent a falcon volant of the field",
"currentLord": "Robert Arryn"
},
{
"region": "Stormlands",
"unid": "EF827514E00D43CA88257E000016915D",
"seat": "Storm's End King's Landing (House Baratheon of King's Landing) Dragonstone (House Baratheon of Dragonstone)",
"heir": "Princess Myrcella, Princess Shireen (disputed)",
"title": "Lord of Storm's End, Lord Paramount of the Stormlands",
"overlord": "Baratheons of King's Landing",
"words": "Ours Is The Fury",
"name": "Baratheon",
"description": "House Baratheon of Storm's End is one of the Great Houses of Westeros, and is the principal house",
"coatOfArms": "A crowned stag sable",
"currentLord": "King Tommen I, King Stannis I (disputed)"
},
{
"region": "The Reach",
"unid": "09887656D18F175188257E00001561B0",
"seat": "Highgarden",
"heir": "Extinct",
"title": "King of the Reach",
"overlord": "none",
"words": "-Extinct-",
"name": "Gardener",
"description": "House Gardener of Highgarden is the extinct house of the old and famed Kings of the The Reach. The",
"coatOfArms": "Argent, a hand coupé vert",
"currentLord": "Extinct"
},
{
"region": "Iron Islands",
"unid": "19A73BBCEAB0BC3188257E000016658F",
"seat": "Pyke",
"heir": "Theon Greyjoy",
"title": "King of Salt and Rock, Son of the Sea Wind, Lord Reaper of Pyke",
"overlord": "None, sovereign (disputed by House Baratheon of King's Landing and House Baratheon of Dragonstone)",
"words": "We Do Not Sow",
"name": "Greyjoy",
"description": "House Greyjoy of Pyke is one of the Great Houses of Westeros. It rules over the Iron Islands, a h",
"coatOfArms": "Sable, a kraken Or",
"currentLord": "Sable, a kraken Or"
},
{
"region": "The Reach",
"unid": "D3B19F250F6AEE6988257E000015FAA0",
"seat": "The Hightower, Oldtown",
"heir": "Ser Baelor Hightower",
"title": "Voice of Oldtown Lord of the Port Lord of the Hightower Defender of the Citadel Beacon of the South Kingdom",
"overlord": "House Tyrell",
"words": "We Light the Way",
"name": "Hightower",
"description": "House Hightower of the Hightower is one of the most important and powerful vassals of House Tyrell",
"coatOfArms": "Cendrée, a tower argent with a beacon on fire gules",
"currentLord": "Leyton Hightower"
},
{
"region": "Iron Islands, Riverlands",
"unid": "4DE933E58F65D21388257E00001641A6",
"seat": "Orkmont, Hoare Castle, Fairmarket, Harrenhal",
"heir": "Extinct",
"title": "King of the Iron Islands, King of the Isles and the Rivers",
"overlord": "none",
"words": "-Extinct-",
"name": "Hoare",
"description": "House Hoare of Orkmont is an extinct house of the Iron Islands. Known as the black line, or the black",
"coatOfArms": "Per saltire: two heavy silver chains crossing between (clockwise) a gold longship on black, a dark",
"currentLord": "Extinct"
},
{

```

```
"region": "Westerlands",
"unid": "EABEB1601EF7469F88257E000014F3A2",
"seat": "Casterly Rock",
"heir": "Tommen Baratheon",
"title": "King of the Rock (formerly), Lord of Casterly Rock Shield of Lannisport Warden of the West",
"overlord": "House Baratheon",
"words": "Hear Me Roar!",
"name": "Lannister",
"description": "House Lannister of Casterly Rock is one of the Great Houses of Seven Kingdoms, and the principal
"coatOfArms": "Gules, a lion or",
"currentLord": "Queen Regent Cersei Lannister"
},
{
"region": "Dorne",
"unid": "A84FDD689561713588257E000016B577",
"seat": "Old Palace within Sunspear",
"heir": "Lord of the Sandship Lord of Sunspear Prince of Dorne",
"title": "Lord of the Sandship, Lord of Sunspear, Prince of Dorne",
"overlord": "House Baratheon of King's Landing",
"words": "Unbowed, Unbent, Unbroken",
"name": "Martell",
"description": "House Nymeros Martell of Sunspear is one of the Great Houses of Westeros and is the ruling house of
"coatOfArms": "Tenny, a sun in splendour gules transfixed by a spear bendwise Or",
"currentLord": "Old Palace within Sunspear"
},
{
"region": "The North",
"unid": "896AA1D0286E4FE088257E0000123C29",
"seat": "Winterfell",
"heir": "Rickon Stark",
"title": "King in the North, Lord of Winterfell, Warden of the North, King of the Trident",
"overlord": "None (formerly House Baratheon)",
"words": "Winter is Coming",
"name": "Stark",
"description": "House Stark of Winterfell is one of the Great Houses of Westeros and the principal noble house of
"coatOfArms": "A running grey direwolf, on an ice-white field",
"currentLord": "Brandon Stark"
},
{
"region": "Crownlands, Valyria",
"unid": "0103FD4EB458904B88257E00000FE3E6",
"seat": "Red Keep, Dragonstone, Summerhall",
"heir": "",
"title": "King of the Andals, the Rhoynar and the First Men Lord of the Seven Kingdoms",
"overlord": "None",
"words": "Fire and Blood",
"name": "Targaryen",
"description": "House Targaryen is a noble family of Valyrian descent that escaped the Doom. They lived for centu
"coatOfArms": "Sable, a dragon thrice-headed",
"currentLord": "Queen Daenerys Targaryen"
},
{
"region": "The Reach",
"unid": "57FFA05DC92F9CCF88257E000015CF43",
"seat": "Highgarden",
"heir": "Willas Tyrell",
"title": "Lord of Highgarden Defender of the Marches High Marshal of the Reach Warden of the South Lord Paramount
"overlord": "House Baratheon",
"words": "Growing Strong",
"name": "Tyrell",
"description": "House Tyrell of Highgarden is one of the Great Houses of the Seven Kingdoms, being liege lords of
"coatOfArms": "Vert, a rose Or",
"currentLord": "Mace Tyrell"
}
}
]
```

A Note on Domino Data Service

As the `@unid` property won't register as a valid `json` object property, you may need to transform that to `unid`. My `HTTPServlet` was already using `unid`, so it works out quite nicely 😊.

Json-Server

So, you may be wondering what `json-server` is. [Json-server](#) describes itself as:

```
...a full fake REST API with zero coding in less than 30 seconds (seriously)
```

I would call it an `application/json` mock back-end service. No matter how you slice it, it will take the contents of my `db.json` file and provide an [endpoint](#) for it to have the various CRUD operations performed against it. It respects well-formed CRUD operations, in `application/json` format, with `GET`, `POST`, `PUT`, `PATCH`, and `DELETE` operations.

To install it, you will need to have [Node.js \(and npm!\)](#) installed on your machine. I recommend installing this globally, so you won't need to maintain a copy of it in some project folder. To do so, from your command line, run:

```
npm install -g json-server
```

This will install the current version of `json-server`. What makes this all so exciting to me is that after [issue \(feature request\) 103 was completed](#), the `id` property is now configurable. So, assuming a relative path to the `db.json` file, running the following gives you a functional back-end mock with the file above:

```
json-server --id unid --watch db.json
```

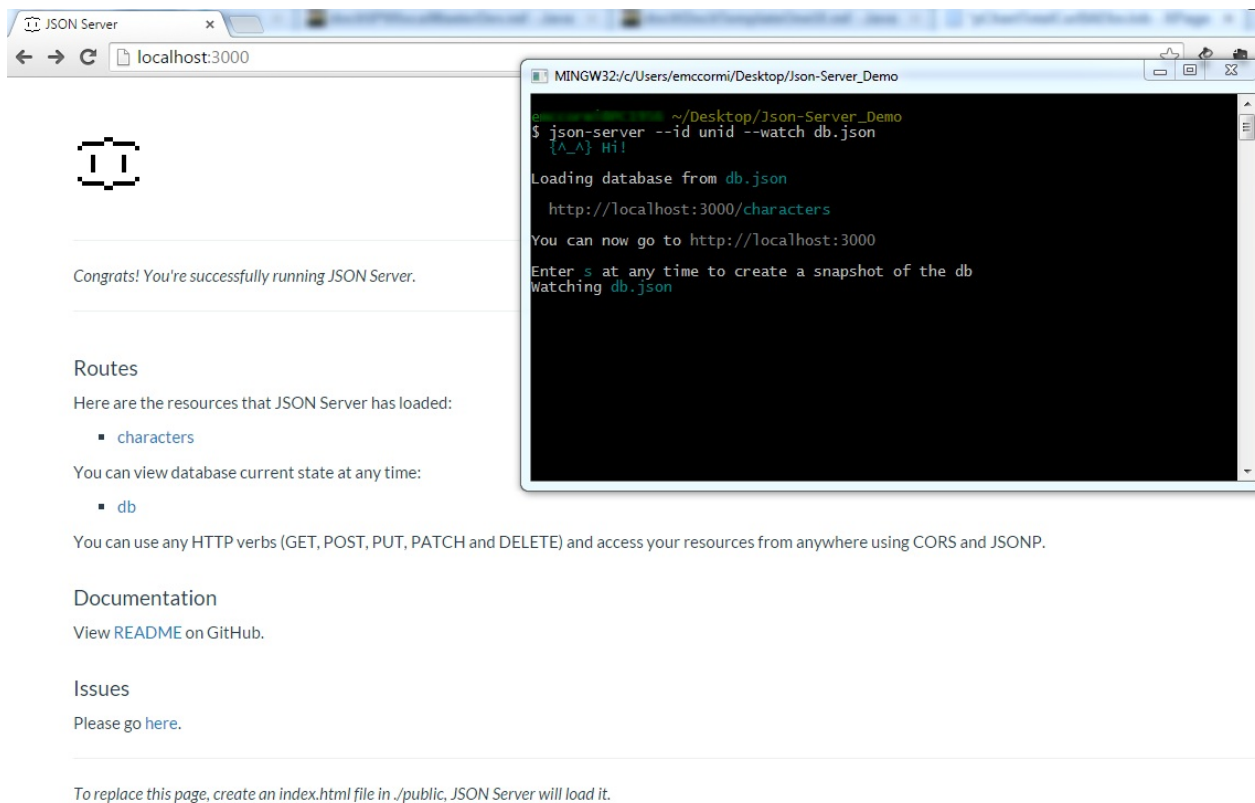
We're invoking the `json-server` command (that's right, npm installed it into our PATH, making it available as a global command), we're configuring the `id` to key off of the `unid` property, we'll be watching the file for changes, and it's pointing at the `db.json` file. By default, it will load on port 3000, but that's configurable as well. See the `json-server` read me on [GitHub](#), or run it with no params or with `-h` for a listing of what's available (port is set by `--port` or `-p`).

Examples

Here come a few examples, in all the glory an animated GIF of a [REST API](#) client can give.

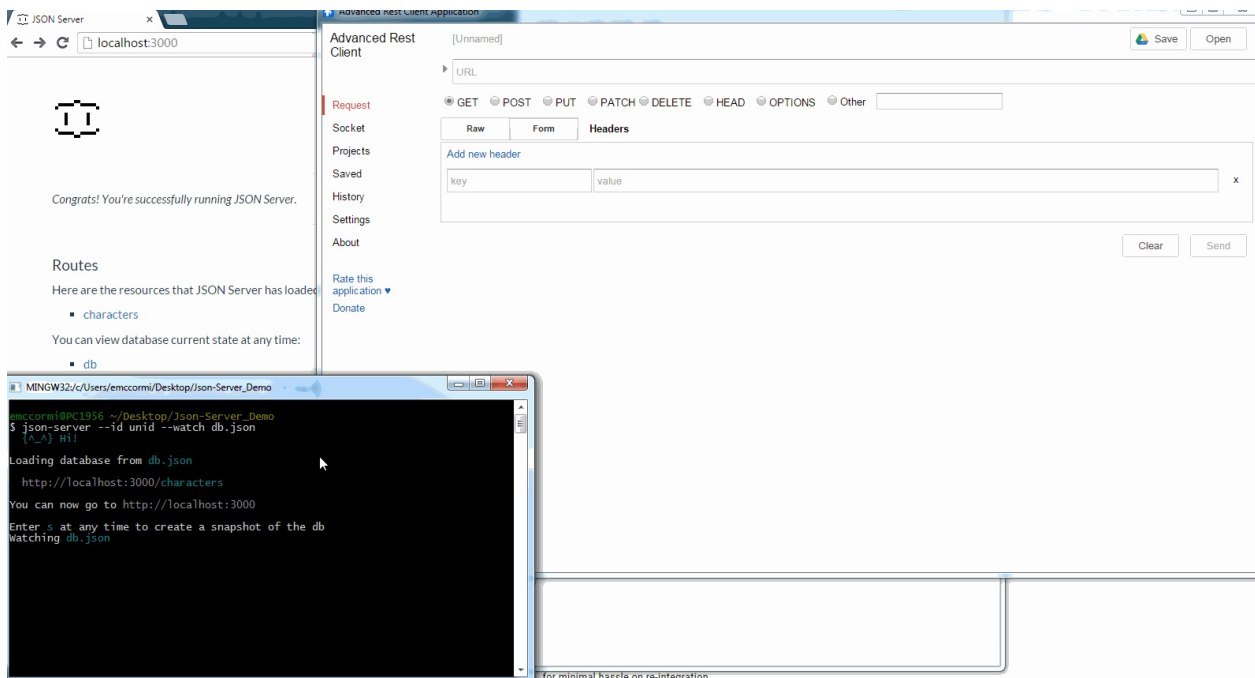
Getting Started

You'll see that the default page at the port `json-server` is serving on, that there are a couple things, such as a hyperlink to the collections, overall "db", and link back to the readme on [GitHub](#). In the console, we can see that we can even take a "snap shot" of the "db"; this will save the data at that point in time to a separate `.json` file.

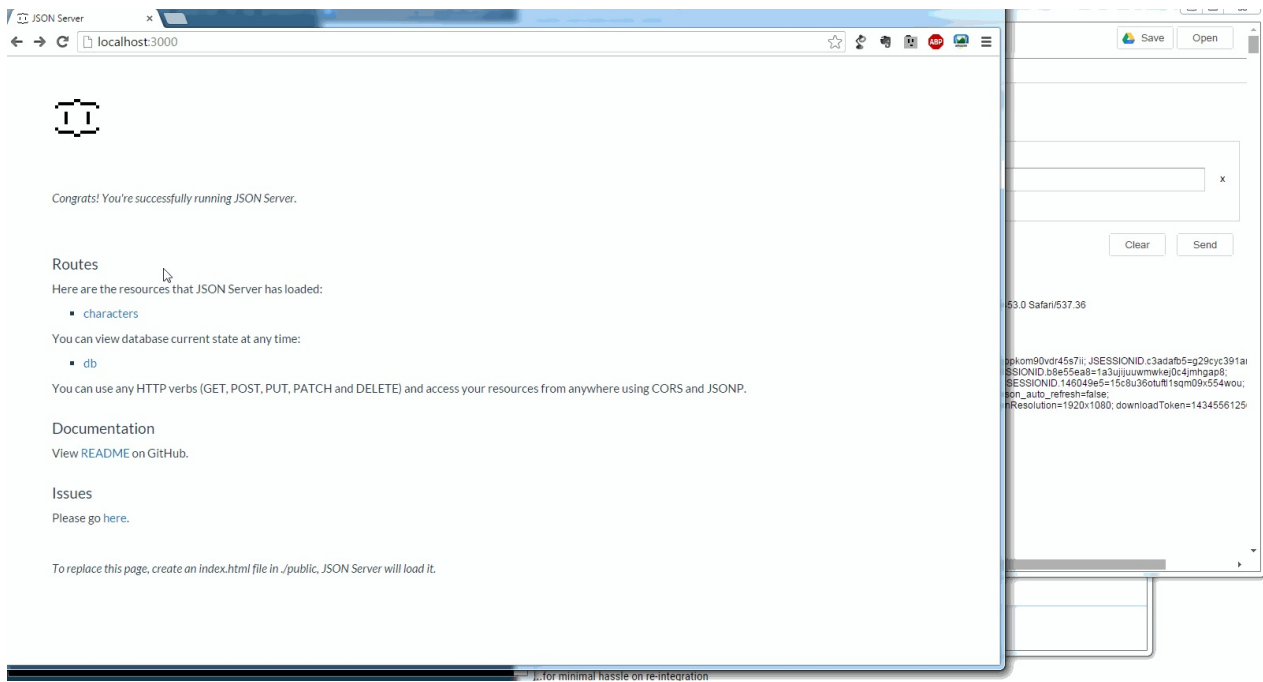


GET Collection

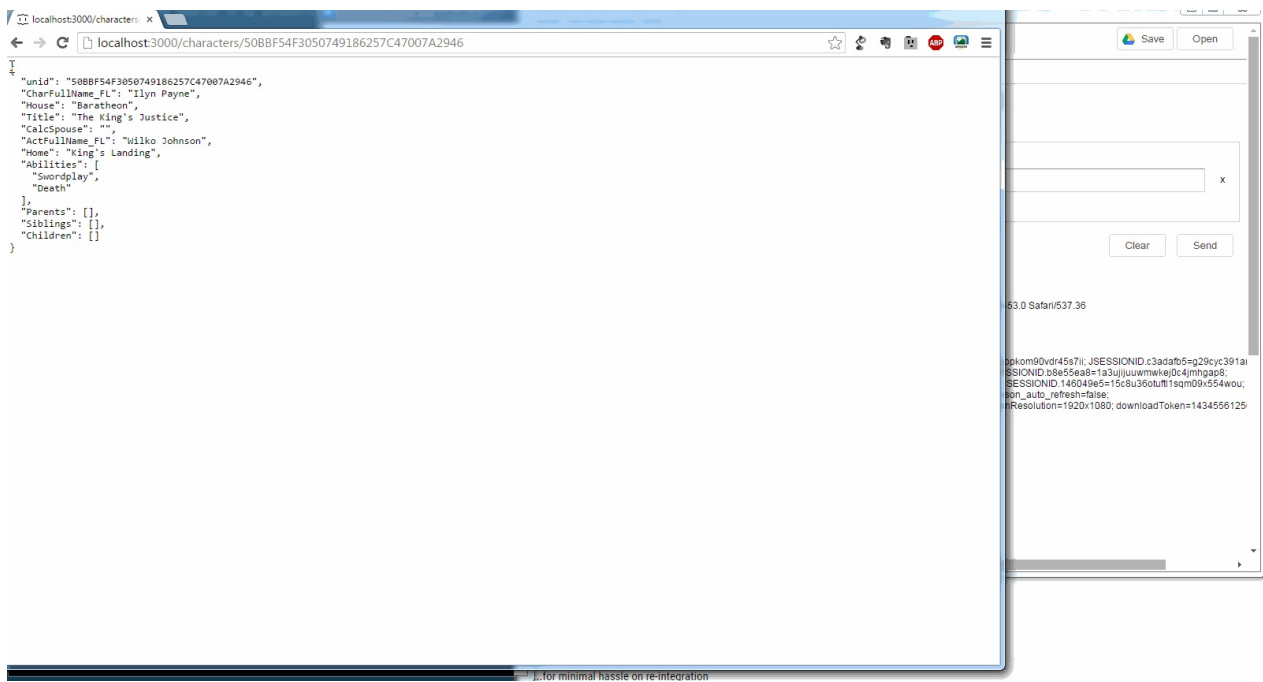
I'm using the [Advanced REST API Client](#) for Chrome, as I'm used to it. You may wish to check out [Postman](#), another Chrome app, or you can load from your JS console or, as I'll show later, [actual web content](#).



GET Record



PUT Record



ad nauseum

One Last Thing

The part that makes this all so awesome, is how extensible this all is. For example, if I were to create a folder called "public" in the same path I'm running `json-server` from (with my `db.json`), `json-server` will pick up on that and display that set of contents instead of the default helper page. How is this useful? Well, check out this nifty example from my "App of Ice and Fire". You'll notice that the type of dataset is changing slightly (to be the same from that app), but otherwise it's the same.

Create Public Folder

I'm going to demonstrate this last bit with my 'App of Ice and Fire' app. I'm setting the new `db.json` file in the root of the project folder and am symlink-ing a `public` folder to that app's `NSF/WebContent/` path (where my static files reside). To do this on a *nix machine, change directory to the project folder and type `ln -s public NSF/WebContent/` and for a Windows command prompt,

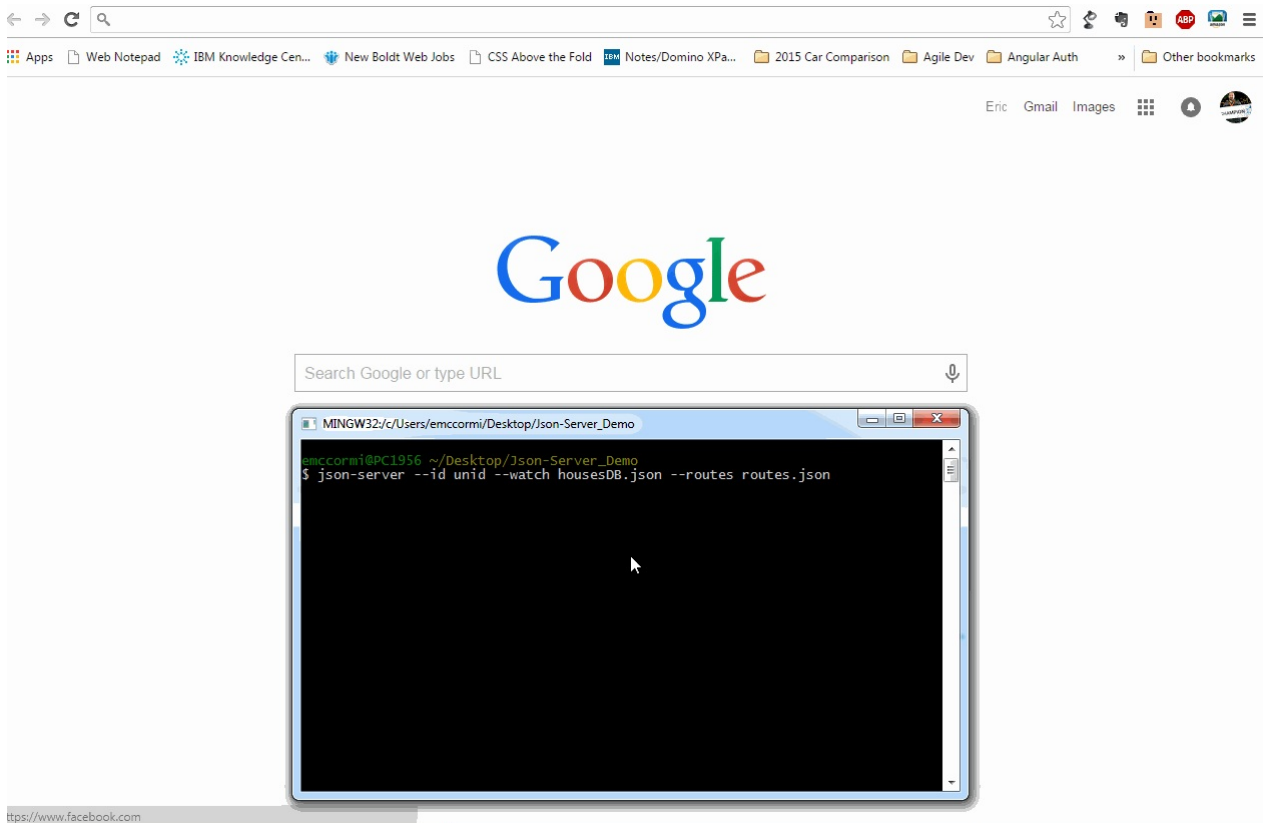
`mklink /d public NSF\WebContent` (if using the default Windows command prompt, you may need to "run as administrator" to get it to work). I added a symlink to my repository, which [GitHub](#) *ought* to respect, but I don't want to duplicate the contents so it's in the `.gitignore` file; just know that when using a [Git](#) repository, all symbolic links should be relative.

With that in place, the only thing standing between myself and a working, non-[Domino](#) server, local copy of my static assets is to map the following into a `routes.json` file and add the parameter when I call `json-server`, like so:

```
{ "/xsp/houses": "/houses", "/xpp/houses/:id": "/houses/:id" }
```

This will make my NSF-relative calls to `/xsp/houses` resolve to the `houses` that `json-server` is providing. Check it out. Our final command to start things up is:

```
json-server --id unid --watch housesDB.json --routes routes.json
```



Summary

I hope you can see the benefit of being able to work on your front-end independent of the server. With a little tweaking (I have some code in development that I would remove for production, checking to verify the formatting of my response in my `houseApp.js` to forcibly wrap my collection and document respectively), I'm now able to focus on all the ui-level application without needing to even touch my development server. To cross apply, I only have to paste into my `WebContent` path, without worrying if I'll break anything on the server. All in all, it's another good tool for the toolbox.

You can find these updates in my [App of Ice and Fire](#) repository on [GitHub](#). Please feel free to check it out and play with it. Until next time.



Java

Posts which are generally based on the topic of [Java](#) development.

Introduction

Many of the [XPages](#) Managed Bean demonstrations point to your ability to populate an `xp:comboBox` with a custom defined List of Select Items. One thing that seems to happen to me is that I wind up having to re-sort such lists to work off of their Label, as opposed to their value; so as to look sorted, at least to human eyes.

A Brief ComboBox Anatomy Lesson

An `xp:comboBox` lets us build out a list (preferably somewhat short) of values with their labels, which are selected from a "drop down" like interface. More specifically, from MDN,

The HTML select (`<select>`) element represents a control that presents a menu of options. The options within the menu are represented by `<option>` elements, which can be grouped by `<optgroup>` elements. Options can be pre-selected for the user.

```
<select>
  <option value="1">One</option>
  <option value="2">Two</option>
  <option value="3">Three</option>
</select>
```

But you're here for the code. Here's an incredibly simple select tag implemented with three options. If you switch to the HTML pane, you'll see that the value (which is what can be data bound for value in the `xp:comboBox` control) is 1, 2, or 3 while the labels are their English equivalent of One, Two, or Three. In classic [Notes](#), we would achieve this by the usual list (line separated) by passing in sets of `Label | Value`, separated by the pipe character. You can still do this in [XPages](#), but if you're defining the source for one in a bean, you'll want to build out your List. My sample class below shows this, but the meat and potatoes here is the Comparator.

A Comparator

Enter [java.util.Comparator](#). It's a member of the Collections Framework, making it ideal for sorting Collections (which a List is). So, to begin, we'll define a class (you can nest it in another class, as I have, a stand-alone class, or a member of another, utility class). This class contains a single, public compare method, which returns an int. It returns an int, as that's what's returned by the [compareToIgnoreCase method of java.lang.String](#)). All the compare method is doing is comparing whether the first string is before or after the second string.

Code

Here's my super simple sample bean, with the `selectOptionsList` being read-only (no setter method) as it's just the `selectedOption` being what the value to be stored is.

```
package com.eric.test;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

import javax.faces.model.SelectItem;

public class SampleComparatorUse implements Serializable {

    private static final long serialVersionUID = 1L;

    private String selectedOption;
    private List<SelectItem> selectOptionsList;

    public SampleComparatorUse() {}

    /**
     * Custom Comparator, for use with sorting (ascending) a List<SelectItem>.
     */
}
```

```

private static class LabelAscComparator implements Comparator<SelectedItem> {
    //uses a one-off comparison which returns comparison boolean, as int
    public int compare(SelectedItem s1, SelectedItem s2) {
        //you can also do a case sensitive via s1.getLabel().compareTo(s2.getLabel())
        return s1.getLabel().compareToIgnoreCase(s2.getLabel());
    }
}

/**
 * Getter for Combo Box options, sorted alphabetically ascending
 * by the label. Read-only, as it's a computed value, so no setter.
 */
public List<SelectedItem> getSelectOptionsList() {

    if( this.selectOptionsList == null ) {
        List<SelectedItem> options = new ArrayList<SelectedItem>();
        //normally I compute this by pulling in values from another source, iterated
        //these are statically added for demonstrative purposes
        options.add(new SelectedItem( "value3", "label3" ));
        options.add(new SelectedItem( "value1", "label1" ));
        options.add(new SelectedItem( "value2", "label2" ));

        //auto-magic sorting! otherwise the order would be label3, label1, label2
        //results, based on the label, in label1, label2, label3
        Collections.sort( options, new LabelAscComparator() );

        selectOptionsList = options;
    }

    return selectOptionsList;
}

/**
 * @param selectedOption String value being set via the EL binding;
 * this is the data field for what has been selected, standard setter.
 */
public void setSelectedOption( String selectedOption ) {
    this.selectedOption = selectedOption;
}

/**
 * Standard getter for the selectedOption property of the bean.
 */
public String getSelectedOption() {
    return this.selectedOption;
}
}

```

The XPage control implementation is a standard `xp:comboBox` implemented with the value and select items (options) bound via [EL](#). The value which the user selects is bound to the bean's property of `selectedOption` while the list of `SelectedItem`s (options list, with both value and labels populated) is the `selectOptionsList` property.

```
xmlns:xp="http://www.ibm.com/xsp/core">
```

Lastly, the `faces-config.xml` to demonstrate the management of the bean into scope.

```
xml version="1.0" encoding="UTF-8"?>
```

```
<faces-config>
  <managed-bean>
    <managed-bean-name>myBean</managed-bean-name>
    <managed-bean-class>com.eric.test.SampleComparatorUse</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>

  <!--AUTOGEN-END-BUILDER: End of automatically generated section-->
</faces-config>
```

Intro

JSON, as [previously mentioned](#), is a data format which has been exploding in web development since it was [first introduced in the early 2000s](#). And whether or not you as a developer prefer [XML](#) (it's okay, they're just formats), there are some [good reasons](#) to use JSON data. Ultimately, I don't really care about the "[XML vs JSON](#)" *debate*, because some services use [XML](#) and some use [JSON](#), neither are going away anytime soon, and [XML is more flexible than most people give it credit for](#).

*Note: I **am** more of a [JSON fan](#), but that should be immaterial to relevance. The biggest argument I see in favor of [JSON](#) as opposed to [XML](#) is [file size](#).*

JSON

To date, when I've shown examples on this blog of how to build [JSON](#), I've generally used Google's [GSON](#) library. I've also only shown it in a fashion (for simplicity's sake) that I refer to as the "old" way (below), because it maps easily to converting to using the [IBM Commons JSON](#) library (more below). I try to add [Gson](#) to the server when possible, but often will end up importing the JAR to an NSF, should I not have administrator blessing. [Gson](#) is supported from [Java](#) 1.5 through 1.8, according to their [pom file](#).

This is in contrast to the provided [com.ibm.commons.util.io.json](#) package, which is included and makes it a convenient option for many/most.

Be forewarned! To use [com.google.gson](#), you will need to grant permission for it in your [java.policy](#) file; [you can run into trouble if you don't](#). This is probably the second best argument against using [com.google.Gson](#), but I'm still a fan.

"Old" Way

Part of the reason [com.ibm.commons.util.io.json](#) is popular (aside that it comes with the server, a big plus) is that it maps well to how we think. Streaming in elements into an object tends to make sense to us, but there's another way. Here's what I'll refer to as the "old" way (it works, it's valid, but not ideal as I'll show).

```
//...
private void buildJsonData() {
    JsonObject myData = new JsonObject();
    myData.putJsonProperty("hello", "world");
    JsonArray dataAr = new JsonArray();
    for( int i=0; i<5; i++ ) {
        JsonObject subObject = new JsonObject();
        subObject.putJsonProperty("_id", i+1);
        subObject.putJsonProperty("someOtherKey", "someOtherValue");
    }
    myData.putArray("data", dataAr);
    myData.putJsonProperty("error", false);
}
//...
```

This will generate a resulting [JSON](#) string with an object, represented as such:

```
{
  "hello": "world",
  "dataAr": [
    { "_id": 1, "someOtherKey": "someOtherValue" },
    { "_id": 2, "someOtherKey": "someOtherValue" },
    { "_id": 3, "someOtherKey": "someOtherValue" },
    { "_id": 4, "someOtherKey": "someOtherValue" },
    { "_id": 5, "someOtherKey": "someOtherValue" }
  ],
  "error": false
}
```

It may not be very exciting, but it sure gets the job done. Here's what I'm excited about.

"New" Way

I first saw a technique in which a person used a [Gson](#) instance to generate, on the fly, *application/json* by merely calling the the [Gson.toJson](#) method). I thought this was cool, but it made good sense. The [Java](#) object already existed and inherited from a proper class, which can loosely map to the [JavaScript](#) prototypal elements (string, boolean, array, object, integer, etc.). [Gson](#) is not unique in this, as the [IBM Commons JSON](#) library can achieve the same thing, using a [JsonGenerator](#)). That's the easy side of things, the tricky part is going backwards, consuming [JSON](#) into a [Java](#) Object (or just creating it from existing [Java](#) objects without being so linear in re-iterating properties just to change the format they're stored in).

IBM Commons JSON

Using [JsonParser](#), you can use [fromJson](#)), which returns a [java.lang.Object](#). In other words, you need to do your tests and transforms to get a handle on its structure. This works, but takes more effort (I would be glad for someone to show me how to map the [IBM Commons](#) library to the approach I'll show next).

Google Gson

The [Gson](#) approach is to take in a class definition (or type) as the second parameter in their [fromJson](#) method, immediately mapping your object to a well structured object that you can invoke for its properties. Here's a quick demonstration.

```
...
/*
 * the main data object, we've read the API docs and know what to expect ;- )
 * assuming that the previously output JSON data is what we're pulling off of
 */
class SomeNiftyDataObject {
    private String hello;
    private List<SomeNiftySubObject> dataAr = new ArrayList<SomeNiftySubObject>();
    private boolean error;

    /*
     * the sub-object, in the dataAr
     * since we need to define the sub-object's
     * structure as well
     */
    class SomeNiftySubObject {
        private String _id;
        private String someOtherKey;

        /*
         * Getter / Setter pairs
         */
        public String get_id() { return _id; }
        public String getSomeOtherKey() { return someOtherKey; }

        public void set_id( String id ) { this._id = id; }
        public void setSomeOtherKey( String someOtherKey ) { this.someOtherKey = someOtherKey; }
    }

    /*
     * Getter / Setter pairs
     */
    public String getHello() { return hello; }
    public List<SomeNiftySubObject> getDataAr() { return dataAr; }
    public boolean getError() { return error; }

    public void setHello( String hello ) { this.hello = hello; }
    public void setDataAr( List<SomeNiftySubObject> dataAr ) { this.dataAr = dataAr; }
    public void setError( boolean error ) { this.error = error; }
}
}
```

```
...
/*
 * we're building data, from a received set of JSON, into
 * a Java object, so we can do normal Java things with it
 */
private void buildMyNewJsonData () {
    // assuming that the JSON of the data is set in a string called rawData
    Gson g = new Gson();
    SomeNiftyDataObject nwData = g.fromJson( rawData, SomeNiftyDataObject.class );
    //SomeNiftyDataObject is now instantiated with the data set according to our class above!
}
...
```

Why The "New" Way?

It's obviously more verbose up front, but done the "old" way, I didn't show all the type checks and conversions I would have to do to keep things working as expected. The "new" way defines the data format and ensures consistently well-formed objects; they are POJO instances after all (beans, except for the implementing [java.util.Serializable](#) bit, as we are using getter/setter methods).

You've defined the format and instantiated data object, meaning that now all you need to do is use the standard [EL](#) get/set to interact with the data. That's it, you're done!

JavaScript

Posts which are generally based on the topic of [JavaScript](#) development.

Consistent Multi-Value Formatting

The [Notes/Domino API](#) is, to be polite, nuanced. It produces interesting results when a sane person might expect a more reasoned approach. For example, one of the staples of [Notes/Domino API](#) is the ability to have multi-value fields. Approaching [Domino/XPages](#) as a novice a couple years ago, I found it odd that performing a `(NotesDocument)getItemValue` on a field with multiple values checked in the field properties of the Form, from which the given document was computed (making it effectively a programmatic schema), would still yield a `java.lang.String` (or its respective object type) when a single value. When the field has multiple values, it returns a `java.util.Vector` containing the respective objects for its values. To account for this sort of situation, a developer then needs to account for the different types of returned values. This makes an otherwise simple call a bit tedious.

Unbeknownst to me, Mark Leusink must have felt the same, as he posted a helper function to convert any value to an `Array` in his [\\$U.toArray XSnippet from December 2, 2011](#). Since I didn't find XSnippets (somehow, I'm not certain how), I created my own version working directly with `java.util.Vector`s. I believe there is still merit to this, as when it performs the typeof, if it's already a `java.util.Vector`, it does no conversion, as opposed to invoking an additional `toArray()` call. My version also makes use of a switch block, which means that it handles unexpected results, in my opinion, somewhat gracefully. Have a look.

```
var util = {
  /**
   * @author Eric McCormick
   * src: http://edm00se.github.io/DevBlog/xpages/consistent-multivalue-formatting/
   * @param java.util.Object to examine
   * @return java.util.Vector of values from originating Object
   */
  asVec: function(obj){
    switch(typeof obj){
      case "java.util.Vector": //it's already a Vector, just return it
        return obj;
        break;
      case "java.util.ArrayList": //it's an ArrayList, return it as a Vector
      case "Array": //it's an Array prototype, return it as a Vector
        var x:java.util.Vector = new java.util.Vector();
        var s = obj.size()||obj.length;
        for(var i=0; i<s; i++){
          x.add(obj[i]);
        }
        return x;
        break;
      case "java.lang.String":
      default: //it's most likely a String, return it as a Vector
        var x:java.util.Vector = new java.util.Vector();
        x.add(obj);
        return x;
        break;
    }
  }
};
```

The first case executes and, knowing that it's in the end format, merely returns it immediately. The second and third case are handled the same, regardless of the differences between a `java.util.ArrayList` and `Array`, their values are still accessible via bracket notation, making the operations performed, with the exception of `.size()` versus `.length` call, the same.

Lastly, the `java.lang.String`, or unexpected results, are wrapped into a `java.util.Vector` and returned. The bottom line is, no matter what happens, you get back exactly what you expect.

To me, this embodies what we strive for as developers; the need to write functional code which, [as with the Unix philosophy](#), does "one thing and does it well". The building blocks of our applications need to be sound, consistent, and perform well under pressure. This builds out a temporary variable *only if necessary* and provides the functionality I had expected in the first place. It's easily built into a helper function library, which is exactly how I use it. Your mileage may vary, but I'm a fan. If anyone has a better way of doing things, I wouldn't mind hearing it.

CLARIFICATION

This all stems from an issue with [Dojo](#) less than 1.6.2 in Chrome 29 and any browser that used the same child [node](#) reference. This post specifically covers how to fix this (as it occurs with [Domino](#) 8.5.3) and get one of the most popular web standards compliant browsers back in the game with [Domino](#) and the ExtLib/UP1 controls.

Fixing Dojo 1.6.1 in Domino 8.5.3

I ran into a situation recently that required a bit of determination to fix. The BLUF: my implementation of the [Dojo Enhanced DataGrid](#) was breaking when applying the [dojox.grid.enhanced.plugins.Filter](#) due to an issue with the [Dojo](#) queries of elements rooted at the specified element. For example:

```
dojo.query('> *', dojo.byId('container'))
```

Thankfully, that [doesn't keep a good developer down](#).

Domino 9, 8.5.3 UP1, and my Woes

[IBM Domino](#) 9 brings a great many changes to the [Domino](#) server and has been fairly well received. Stuck, for now, with 8.5.3, I was at least able to get Upgrade Pack 1 applied, giving me the basic level of the majority of the same new controls.

That being said, I still had the controls I wanted to play with, so I still tried the "play at home" version of [Brad Balassaitis'](#) excellent series on [Dojo Grids in XPages](#). When I hit [Part 14: Enhanced Filtering with No Coding](#), I found that the Filtering plugin would cause my control to break in a rather unexpected fashion.

Dojo 1.6.1

[Domino](#) 8.5.3 has [Dojo](#) 1.6.1. The culprit in question, as I found out from attempting to use Chrome with the Enhanced Grid, the issue was with the child selector call.



After finding what the issue was, followed by some intense Google searching, I had found [the fix for this in Dojo 1.6.2](#). The fix was small enough, I thought, "why can't I implement this?" So I did.

Doing Something About It

1. Extract the Source Dojo from the JAR

Starting about [Domino](#) 8.5.3, the [Dojo](#) library inclusion migrated from the usual source path in `\data\domino\js\dojo-1.x.x` folder structure on the file system to a JAR deployment. The 1.6.1 source can be found in `\osgi\shared\eclipse\plugins\com.ibm.xsp.dojo_8.5.3.20131121-1400.jar`. To begin, we need to extract the 1.6.1 source files out of the JAR, I recommend using 7zip, though any method of un-zipping the JAR will suffice. The folder structure is in the resources\dojo-version directory. Extract that to the older format js directory in the [Domino\data](#) path and you now have a working version of the 1.6.1 [Dojo](#) library. I recommend giving your extract [Dojo](#) library a better name than my very boring 1.6.1.source; like `.modified` ;-)

2. Apply the Fix

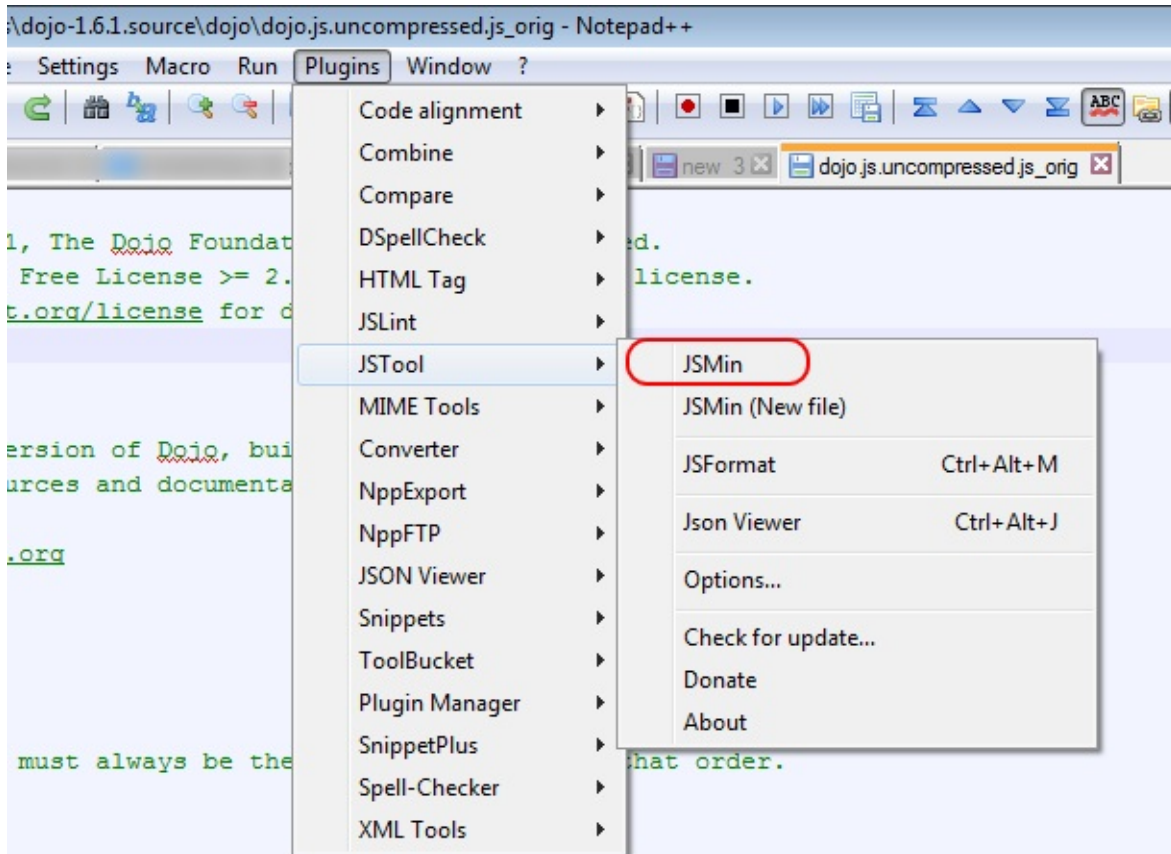
Per the relevant commit in [Dojo 1.6.2](#), which addressed this issue, we need to make our change to the `dojo.js` file in two locations. I recommend making a backup copy of:

- `dojo.js`
- `dojo.js.gz`
- `dojo.js.uncompressed.js`

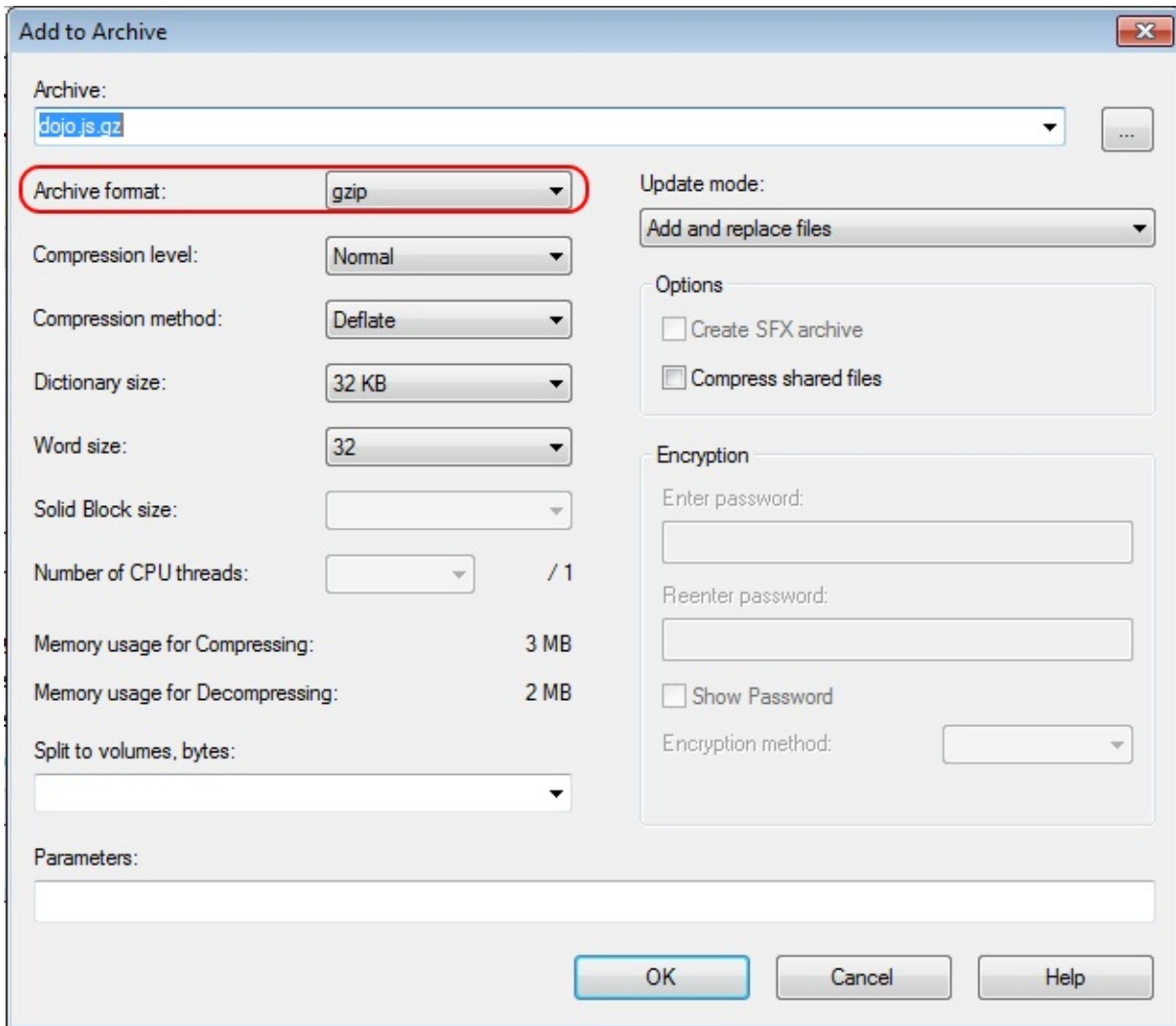
You can see there are actually three files, one minified, one minified and gzip'd, and one un-minified. Per the description in the [commit message](#), we need to find the `root[childNodesName]` references and replace them with `root.children || root.childNodes`, found in two locations. They specified two line numbers, but mine turned out to be lines 8644 and 8925. I'm chalking up the variance to our 1.6.1 version coming to us via [IBM](#) (I'm guessing comments). Since the lines we change are those directly dealing with our issue, the child dependency handling, I know we're good. Save your file over the `dojo.js.uncompressed.js` file.

3. Apply Again

Now that the original, un-compressed version has its updates, it's time for those minified and minified and gzip'd versions. If you're using an editor like [Notepad++](#), you can use a plugin such as [JSTool](#) to perform the JSMIn operation.



Save that file over the `dojo.js` file. Now for the gzip'd version. With 7zip, you can perform a right-click on the `dojo.js` file and select 7zip > Add to Archive. For the settings, select the correct archive format, ensuring the `.gz` extension gets applied and overwrite the existing `dojo.js.gz` file.



4. Restart and Use

Now you only need to restart the [Domino](#) server and you can start using it. Just restarting the http task doesn't quite do it, as [Domino](#) needs to fully re-register all its known libraries (not just re-initialize the handling of http connections). If you don't, but you set the library in the Xsp Properties, you will not be able to load your new version, as the selected library will return a runtime error, as the server hasn't registered it yet.



To use it in an NSF, open the Xsp Properties file and specify your modified name in the [Dojo](#) version field. If you prefer the source, it's applied by `xsp.client.script.dojo.version=1.6.1.modified` (or `.source`, whichever you call it). Note to leave off the `dojo-` prefix.

XPage Properties

Theme Defaults

Application theme: Platform default ▼

Override on Web: Application default ▼

Override on Notes: Application default ▼

Use mobile theme for XPages with the prefix: m_

Mobile theme: Mobile default ▼

Override on iOS: Application default ▼

Override on Android: Application default ▼

Debug user agent: ▼

Window Behavior for Navigation and Links

(Notes client only)

Server default

Open in same window and tab

Open in new window or tab (per client preference)

Minimum Supported Release

Compile this application to run on:

Minimum release required by the XPage features used ▼

Use 8.5.0 style and styleClass property

Ignore errors for unknown namespace URIs

Dojo

Dojo version: 1.6.1.source

Dojo parameters:

Time Zone

Time zone: Server default Browser Server

Round trip time zone

Error Handling

Display XPage runtime error page

Error page: Standard server error page ▼

Timeouts

Application timeout: (minutes)

Session timeout: (minutes)

General | Persistence | Page Generation | Source

Rejoice

You are now able to use the [Dojo Enhanced DataGrid](#) with the `dojox.grid.enhanced.plugins.Filter!`

← → ↻ https:// Eric/Sandbox.nsf/awesomeGoTgrid.xsp

CharFullName_FL	House
10 of 54 items shown. Clear filter	
Arya Stark	Stark
Bran Stark	Stark
Eddard Stark	Stark
Hodor	Stark
Luwin	Stark
Rickon Stark	Stark
Robb Stark	Stark
Rodrik Cassel	Stark
Sansa Stark	Stark
Talisa Maegyr	Stark

The Original

The stock [XPages](#) date picker control leaves more room for user error than I prefer. Mark Roden [originally came up with this excellent script in jQuery](#). I love [jQuery](#), but it's not the (client-side) JS library I always have available to me in my [XPages](#) work, and I'm not about to load yet another library after [Dojo](#) in an existing, [Dojo](#)-centric application for a comparatively trivial, one-off function. Since the function can be written in vanilla JS, or any decent JS library, I decided to re-write it into [Dojo](#).

The Dojo Version

I have this in production on a few applications, and both I and the users love it. So once again, I'm glad to be privy to the fruits of a great community of fellow XPage developers. They've made my life easier on so many occasions and I'm hoping I can give back in a small way. Now, since I'm using [Dojo](#), apparently this can be achieved in more-or-less the same way with the [dijit.form.DateTextBox](#) [Dojo](#) module (just set the field's [DojoType](#) after specifying the resource), as [highlighted in the comments on Mark Roden's blog post](#). While this is certainly functional, it *does* achieve the picker launch on entering the field, it isn't consistent with the native [XPages](#) date picker control, which is already all over *every single XPage'd application in my company*. So, for consistency and UX as decided by others in my organization, I rolled this [Dojo](#) version of Mark Roden's script.

Here's the code:

```
/*
 * Dojo version of the improved behavior of the XPages calendar picker.
 * Adapted from the jQuery version, originally by Marky Roden.
 * credit: http://xomino.com/2012/03/14/improving-user-interaction-with-xpages-date-picker/
 * Adapted by Eric McCormick, @edm00se, http://about.me/EricMcCormick
 */
dojo.addOnLoad(function(){
  //id has _Container and is class of xspInputFieldDateTimePicker
  var myAr = dojo.query("[id$=_Container].xspInputFieldDateTimePicker");
  //iterate over each element to apply affect
  myAr.forEach(function(node, index, arr){
    //current root node, based on id$=_Container
    var curNode = node;
    //span for the button to fire the picker
    var myBtn = dojo.query('> span > span > span.dijitButtonContents', curNode)[0];
    //actual <input> element into which is focused/typed
    var myInputFld = dojo.query('> div > div.dijitInputField > input.dijitInputInner', curNode)[0];
    //connect the focus event to the picker click event
    dojo.connect(myInputFld, "onfocus", function(){myBtn.click()});
    //provide an onkeypress preventDefault
    dojo.connect(myInputFld, "onkeypress", function(evt){evt.preventDefault()});
  });
});
```

Breaking It Down

We [bootstrap](#) the function via the [dojo addOnLoad](#) call and start by creating an array of the fields with an ID attribute containing `'_Container'` and the class `xspInputFieldDateTimePicker`. We then iterate over these DOM nodes, getting a handle on their button, and connecting the `click()` event call for that button to the respective field during the `onfocus` and/or `onkeypress` events. I'm sure a more advanced user could improve the performance of my version of the function, and I welcome them to post a forked version; after all, why use [GitHub/Gist](#) if you don't want your code improved via the aid of others?

Note, for this to work in [Dojo 1.8](#), the `dojo.connect` and `dojo.query` calls I establish, which work perfectly fine in 1.6, **must** be converted to `dojo.on` and `query` calls, respectively. For more, I do recommend reading the Usage section of the [Dojo 1.8 docs](#).

Other

Development-adjacent and related topics.

Intro

This is a brief intro to nginx, the reverse proxying web server I've fallen in love with every time I've used it. I'm by far [not the first person to blog on the subject](#), but this may be a good starting point for some people.

While setting myself up for editing the [AngularJS](#) version of my app for [my Java servlet series](#), I set up my [git](#) repo to be accessible both inside and outside of my DDE vm, fired up local web preview, and realized that my connection to said local web preview was denying my connections, as I was accessing it from another IP. On top of all this, unless I'm hosting my HTML, JS, or CSS files (my static content) from within the design elements of Pages, Scripts, or Style Sheets, I wasn't going to get any *gzip* response benefits, regardless of the XSP Properties setting.

Wanted to use local web preview w/ [#XPages](#) outside my vm, access denied. Now, my [#nginx](#) reverse proxy has solved that problem in ~5 minutes.

— Eric McCormick (@edm00se) [March 22, 2015](#)

Nginx: the 'What' and 'Why'

Nginx (pronounced "engine X") is an open source HTTP [reverse proxy](#) web server. It also does normal file serving, etc., but its primary goal is to be a reverse proxy. This has many benefits and comes up very commonly as being a front-end server for [Node.js](#) applications; so serve the static content, offload cached response handling to something other than a [Node.js REST API](#) (e.g.- if the content doesn't change, don't re-build it), and other front-end things like minification or gzipping responses or more complex tasks like load balancing. These all have very obvious advantages, I'll just fill you in on the few I've used for this situation.

Aside: I've used Nginx as a front-end server for a couple [Node.js](#) apps at work and have impressed my admins with the ability to make their lives easier with their managing of a web server's SSL certificates and other admin-y thngs, all independent of the application server, it's been a hit. In fact, if we weren't running all our [Domino](#) server traffic through a Citrix NetScaler, we would be running an Nginx reverse proxy in front of every [Domino](#) server serving web content, after this past year's POODLE scare.

Setup and Config

In order to access a server hosted within a vm (guest), for development purposes from the host OS, which is restricted to same origin / localhost only requests, I set up a siple nginx reverse proxy to forward my requests.

Steps

1. To install in a Windows VM, download and install [nginx](#) from the current, stable release; I installed to C:\nginx\
2. Edit the `<install path>/conf/nginx.conf` file with the marked changes in the file of the same name in this gist.
3. Start the nginx executable, located in your install path. There are service wrappers for Windows, or you can just kill the process to stop the nginx instance.

Commands for Windows

More information on the implementation of nginx in Windows can be found on [the corresponding docs page](#). Here's the basic breakdown of commands, form within the nginx install directory:

Command	
start nginx	starts the process
nginx -s stop	fast shutdown
nginx -s quit	graceful shutdown
nginx -s reload	config change, graceful shutdown of existing worker proc, starts new
nginx -s reopen	re-open log files

Description

The config file contains a `server` block, inside which is a `location /` block. Inside that location block, we need to replace the root and index assignment with a `proxy_pass http://127.0.0.1:8080`; line and a `proxy_http_version 1.1`; line.

Sample Nginx.conf

```
...
server {
    listen      80;
    server_name localhost;

    # adds gzip options
    gzip on;
    gzip_types  text/css text/plain text/xml application/xml application/javascript application/x-javascript text/j;
    gzip_proxied no-store no-cache private expired auth;
    #gzip_min_length 1000;
    gzip_disable "MSIE [1-6]\.";

    ...

    location / {
        # Backend server to forward requests to/from
        proxy_pass      http://127.0.0.1:8080;
        proxy_http_version 1.1;

        # adds gzip
        gzip_static on;
    }
    ...
}
```

My Speed Claim

I tweeted a pretty strong sounding result. In fact, I believe that my DDE local web preview being freshly restarted was part of the ridiculously long response for my data set, but there was still a significant improvement of around 400-500ms (down from just over a full second to just over half of one); which shows the improvements gained from *gzing* the static elements.

Enabling gzip, minification, and caching for static assets in [#nginx](#) and watching my load time drop from 2.5s to 550ms makes me a happy guy.

— Eric McCormick (@edm00se) [March 22, 2015](#)

Summary

You don't always need a reverse proxying server in front of your application server, but what it can add to your immediately accessible capabilities, and the segregation between admin-y tedium and application development, is pretty awesome.

Intro

[Source Control](#) has become a rallying point of sorts for me in the last few years. It has saved my bacon on a few occasions and [source control](#) in general (and [git](#), specifically) is near and dear to me. I'm always on the quest for the best application development workflow, which can sometimes be difficult to achieve, depending on varying development team sizes.

I've gone through a couple rounds of preferences when it comes to self-hosted [source control](#) servers in the past couple years. I'm going to break down what I like about externally hosted solutions and then get into the benefits and trade-offs of a few of the (freely available) self-hosted solutions.

Externally Hosted

There are two major players in the (free) externally hosted realm. There are more, but the two big ones of late are [GitHub](#) and [Bitbucket](#).

GitHub

I was going to start with a succinct description of [what GitHub is](#), straight from their page. Oddly enough, there isn't an easy one I could find. They talk a fair amount about enterprise solutions and some sales lingo about enabling teams, so instead I'll drop in the lead paragraph from [the Wikipedia page on GitHub](#):

"[GitHub](#) is a web-based [Git](#) repository hosting service, which offers all of the distributed revision control and source code management (SCM) functionality of [Git](#) as well as adding its own features. Unlike [Git](#), which is strictly a command-line tool, [GitHub](#) provides a web-based graphical interface and desktop as well as mobile integration. It also provides access control and several collaboration features such as wikis, task management, and bug tracking and feature requests for every project."

So, now that we've established that [GitHub](#) is a [git](#) repository server, which provides additional features (issue tracking, wikis, forking, etc.), let's get into some of the differentiating specifics:

- only hosts [git](#) repositories
- any private repos / enterprise hosting costs 💰📄
- generally the de facto solution for the Open Source community
- free static site (or generated via [Jekyll](#)) via [GitHub Pages](#) (any *gh-pages* branch for a [GitHub](#) repo will have a space at [.github.io/](#)), like this one (which uses a custom domain now)

I like [GitHub](#), but it's not perfect. There are plenty of upsides due to its popularity though, like [Travis CI](#), which gives CI testing for free to all [GitHub](#) repositories, or [Gitter](#) which gives "discussions" to [GitHub](#) teams and repositories, across Pull Requests and Issues.

Bitbucket

As with [GitHub](#), [Bitbucket](#) (from Atlassian) offers repository hosting (for both [Git](#) and [Mercurial](#)) along with some extras. Here's their description:

"[Bitbucket](#) is a web-based hosting service for projects that use either the [Mercurial](#) (since launch) or [Git](#) (since October 2011) revision control systems."

- hosts both [Git](#) and [Mercurial](#) repositories
- free accounts get an unlimited number of public or private repositories
- private repositories with a free account can have up to 5 collaborators
- additional enterprise hosted options with different products for different aspects of code management (issue tracking with Jira, Stash for self-hosted [Bitbucket](#) server, Bamboo for CI, and more); also costing 💰📄
- strong articles and guides (Atlassian blog)

Atlassian is also the creator of SourceTree, a free [Mercurial](#) and [Git](#) client for Mac and Windows platforms. SourceTree bakes in the [Git/Hg-Flow](#) processes, making it easier to implement [feature branching](#) workflows (or others).

Winner?

What are you looking for? I could claim one as superior to the other, and my inclination is that [Bitbucket](#) is technically superior, but I believe we're better off for having both. They bring us a lot of goodies in an attempt to gain our business with increasingly better tools. This is a good thing.

Self-Hosting

Why self host? I could a few specific reasons, but ultimately, I've found that self-hosting is really only going to depend on imposed business requirements. Depending on the size of your "shop", this can be a rabbit hole, as you'll see from my experiences with Redmine.

Redmine

[Redmine](#) was one of the early kids on the block, is written with Ruby on Rails, and supports [Git](#), [Mercurial](#), SVN, CVS, and others. It works and supports quite possibly the highest number of protocols, but can be a bit cumbersome, especially compared to how easy some of the others are to operate. I set up a Redmine instance at my day job and it's been sufficient, though doesn't always sport things we've grown to expect, like a link to clone (HTTPS or SSH) at the top.

GitLab CE

In all honesty, [GitLab](#) very nearly replicates all the beauty and ease of use you would come to expect from [GitHub](#) or [Bitbucket](#) and is freely available. It's slick. If I hadn't run into trouble, I would still be using it. It's worth a look and I **highly recommend extensive backups before upgrading versions**. Its biggest potential downside for some is that it's [Git-only](#).

I fell in love with GitLab's Community Edition... after I got it set up the first time, which was before the omnibus package would install correctly on my home Ubuntu box. Eventually, I switched to the omnibus package, but that eventually failed hard for me on an upgrade (via the recommended steps) and I lost my db. I had enough hassling at that point, as [you can probably tell from the issue](#) I had open for a while.

Gogs

When I went looking for alternates, I found [Gogs](#), which says it's a "self-hosted [Git](#) service written in Go". That's [Go](#) as in [GoLang](#), not that that really matters much. Gogs runs rather light and can [run on a Raspberry Pi](#).

I've been using it for a while, and [even submitted a PR](#). I thought it odd that I was directed to a different repo for the PR, but it seems to be a bit of a community effort.

Summary

All in all, if you're looking to go your own way, most of the self-hosted options focus on [Git](#) with no real regard for other protocols. If [Mercurial](#) is a must, I recommend using a free account with [Bitbucket](#) and if your needs change, roll with the punches. As for those of us who have bought into [Git](#), I thoroughly enjoy the simplicity of Gogs, but will admit to being up for a ~~rematch~~ revisit to GitLab, after we've both had our time to see other ~~people~~ solutions.

Recommended

- Fredrik Norling's blog post on how to [Setup a Free Git Server with Domino Credentials in a Few Minutes](#), which uses [GitBlit](#)
- [Notes in 9 #131: Use SourceTree for Better XPages Source Control](#)
- [Notes in 9 #140: SourceTree Deep Dive](#)

[Edit] You should also check out [Show 103: The Show & Tell on Source Control - An End to End Solution](#) presentation from IBM Connect 2014 by Paul Withers and Declan Lynch. It's worth a read as these two go into some excellent detail on both the hosting solutions, SourceTree, DORA, and some specifics in using them all together. [/Edit]

Syntax Highlighting in Redmine Project Repository

Redmine is a great way to track projects and their code repositories in a "one stop shopping" locale. The only issue is that, while Redmine makes use of its syntax highlighter of choice (CodeRay, by default), it doesn't know about the custom file extensions used by [Domino Designer](#). This can be easily remedied with a few quick edits.

Updating Redmine CodeRay to Syntax Highlight (most) [Domino/XPages](#) Design Elements

Get into the correct file that pertains to file extensions and syntax highlighting definitions.

- go to your Redmine directory (ex- `/var/www/redmine`)
- enter your vendor library path for CodeRay (ex- `vendor/bundle/ruby/1.9.1/gems/coderay-1.0.9/`)
- edit the `file_type.rb` file (which defines the language syntax associations, ex- `lib/coderay/helpers/file_type.rb`) with your preferred editor

Now that you're there, we need to associate the design elements accordingly. Scroll down to the section defining the array of `TypeFromExt` (or search, in nano CTRL+W, for something like `xml`), and add in the following:

- `'fa' => :xml,`
- `'form' => :xml,`
- `'frameset' => :xml,`
- `'jss' => :java_script,`
- `'page' => :xml,`
- `'properties' => :xml,`
- `'view' => :xml,`
- `'xsp' => :xml,`

The [Notes](#) (classic) design elements, especially Form and View, are best viewed in [source control](#) if you have un-checked the 'Use Binary DXL for [source control](#) operations' in [Domino Designer](#) ([Domino Designer](#) > [Source Control](#)) and will read like a semi-decent [XML](#) structure. If you're just concerned about the XPage'd elements, the 'xsp' file extension and 'jss' are the only necessary definitions.

Glossary

AJAX

AJAX (short for Asynchronous JavaScript and XML) is a set of web development techniques utilizing many web technologies used on the client-side to create asynchronous Web applications. With Ajax, web applications can send data to and retrieve from a server asynchronously (in the background) without interfering with the display and behavior of the existing page. By decoupling the data interchange layer from the presentation layer, Ajax allows for web pages, and by extension web applications, to change content dynamically without the need to reload the entire page. Data can be retrieved using the XMLHttpRequest object. Despite the name, the use of XML is not required (JSON is often used in the AJAJ variant), and the requests do not need to be asynchronous.

[3.1. Unraveling the M-V-C Mysteries](#) [3.2. REST is Best](#) [1.6. Servlet Handling Data, A Round House Kick](#)
[3.3. More on HTTP and AJAX Requests](#)

Angular

AngularJS (commonly referred to as "Angular" or "Angular.js") is an open-source web application framework mainly maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

[3.2. REST is Best](#) [3.4. What is a Single Page Application\(SPA\)?](#) [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
[1.8. Building a Front-End pt.2: An App with AngularJS](#)

AngularJS

AngularJS (commonly referred to as "Angular" or "Angular.js") is an open-source web application framework mainly maintained by Google and by a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. It aims to simplify both the development and the testing of such applications by providing a framework for client-side model–view–controller (MVC) and model–view–viewmodel (MVVM) architectures, along with components commonly used in rich Internet applications.

[2.3. Custom JSON with Java-ized XAgent](#) [3.2. REST is Best](#) [3.4. What is a Single Page Application\(SPA\)?](#)
[3. Application Logic](#) [1.6. Servlet Handling Data, A Round House Kick](#) [7.1. A Brief Introduction to Nginx](#)
[1.7. Building a Front-End pt.1 Plus a Quick Review](#) [1.8. Building a Front-End pt.2: An App with AngularJS](#)
[1.9. Series Review](#) [3.3. More on HTTP and AJAX Requests](#) [1. A Saga of Servlets](#)

API

In computer programming, an application programming interface (API) is a set of routines, protocols, and tools for building software applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types. An API defines functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface. A good API makes it easier to develop a program by providing all the building blocks. The programmers then put the blocks together.

[2.2. Server REST Consumption with Authentication](#) [2.3. Custom JSON with Java-ized XAgent](#)
[5.2. Building Java Objects from JSON](#) [1.2. Servlet Intro and Flavors](#) [1.4. Servlet Handling of Requests](#)

- [1.6. Servlet Handling Data, A Round House Kick](#) [7.1. A Brief Introduction to Nginx](#)
[6.1. Consistent Multi-Value Formatting in SSJS for Domino/XPages](#) [4.1. "Replacing" an XAgent](#)
[1.7. Building a Front-End pt.1 Plus a Quick Review](#) [4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)

Application Programming Interface

In computer programming, an application programming interface (API) is a set of routines, protocols, and tools for building software applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types. An API defines functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising the interface. A good API makes it easier to develop a program by providing all the building blocks. The programmers then put the blocks together.

[1.4. Servlet Handling of Requests](#)

Bitbucket

Bitbucket is a web-based hosting service for projects that use either the Mercurial (since launch) or Git (since October 2011) revision control systems. Bitbucket offers both commercial plans and free accounts. It offers free accounts with an unlimited number of private repositories (which can have up to five users in the case of free accounts) as of September 2010, but by inviting three users to join Bitbucket, three more users can be added, for eight users in total. Bitbucket is written in Python using the Django web framework. It is similar to GitHub, which primarily uses Git. In a 2008 blog post, Bruce Eckel compared Bitbucket favorably to Launchpad, which uses Bazaar.

[7.2. Self-Hosting SCM Server](#)

CORS

Cross-origin resource sharing (CORS) is a mechanism that allows restricted resources (e.g. fonts) on a web page to be requested from another domain outside the domain from which the resource originated. A web page may freely embed images, stylesheets, scripts, iframes, videos and some plugin content (such as Adobe Flash) from any other domain. However embedded web fonts and AJAX (XMLHttpRequest) requests have traditionally been limited to accessing the same domain as the parent web page (as per the same-origin security policy). "Cross-domain" AJAX requests are forbidden by default because of their ability to perform advanced requests (POST, PUT, DELETE and other types of HTTP requests, along with specifying custom HTTP headers) that introduce many cross-site scripting security issues.

[1.3. Basic Servlet Implementation](#)

CSJS

Client-Side JavaScript specifically refers to JavaScript code which executes in the client (aka- the web browser) as opposed to Server-Side JavaScript.

[3.2. REST is Best](#) [3.3. More on HTTP and AJAX Requests](#)

Dojo

Dojo Toolkit (stylized as dōjō toolkit) is an open source modular JavaScript library (or more specifically JavaScript toolkit) designed to ease the rapid development of cross-platform, JavaScript/Ajax-based applications and web sites.

- [6.3. An Dojo Implementation of the Calendar Picker Script](#)
- [6.2. Fixing an Older Version of Dojo \(1.6.1\)](#)
- [3.2. REST is Best](#)
- [3.4. What is a Single Page Application\(SPA\)?](#)
- [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
- [3.3. More on HTTP and AJAX Requests](#)

Domino

IBM Domino (formerly Lotus Domino) is the server of a collaborative client-server software platform sold by IBM.

- [0. Intro](#)
- [7.3. Domino/XPages Design Element Syntax Highlighting on Redmine](#)
- [2.1. RESTful API Consumption on the Server \(Java\)](#)
- [2.2. Server REST Consumption with Authentication](#)
- [2.3. Custom JSON with Java-ized XAgent](#)
- [6.2. Fixing an Older Version of Dojo \(1.6.1\)](#)
- [3.1. Unraveling the M-V-C Mysteries](#)
- [3.2. REST is Best](#)
- [3.4. What is a Single Page Application\(SPA\)?](#)
- [3. Application Logic](#)
- [5.1. When You Need a Comparator](#)
- [1.1. A Quick Note on JARs](#)
- [1.2. Servlet Intro and Flavors](#)
- [1.3. Basic Servlet Implementation](#)
- [1.4. Servlet Handling of Requests](#)
- [1.6. Servlet Handling Data, A Round House Kick](#)
- [7.1. A Brief Introduction to Nginx](#)
- [6.1. Consistent Multi-Value Formatting in SSJS for Domino/XPages](#)
- [4.1. "Replacing" an XAgent](#)
- [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
- [1.9. Series Review](#)
- [4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)
- [1. A Saga of Servlets](#)

EL

Expression Language (EL) provides an important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (managed beans). The EL is used by both JavaServer Faces technology and JavaServer Pages (JSP) technology. The EL represents a union of the expression languages offered by JavaServer Faces technology and JSP technology.

- [3.1. Unraveling the M-V-C Mysteries](#)
- [5.1. When You Need a Comparator](#)
- [5.2. Building Java Objects from JSON](#)

Endpoint

Endpoint, the entry point to a service, a process, or a queue or topic destination in service-oriented architecture

- [2.1. RESTful API Consumption on the Server \(Java\)](#)
- [2.2. Server REST Consumption with Authentication](#)
- [1.3. Basic Servlet Implementation](#)
- [1.4. Servlet Handling of Requests](#)
- [1.6. Servlet Handling Data, A Round House Kick](#)
- [4.1. "Replacing" an XAgent](#)
- [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
- [1.9. Series Review](#)
- [4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)

Expression Language

Expression Language (EL) provides an important mechanism for enabling the presentation layer (web pages) to communicate with the application logic (managed beans). The EL is used by both JavaServer Faces technology and JavaServer Pages (JSP) technology. The EL represents a union of the expression languages offered by JavaServer Faces technology and JSP technology.

- [3.1. Unraveling the M-V-C Mysteries](#)

Git

Git is a widely used version control system for software development. It is a distributed revision control system with an emphasis on speed,[6] data integrity, and support for distributed, non-linear workflows. Git was initially designed and developed by Linus Torvalds for Linux kernel development in 2005.

[1.1. A Quick Note on JARs](#) [7.1. A Brief Introduction to Nginx](#) [1.8. Building a Front-End pt.2: An App with AngularJS](#)
[7.2. Self-Hosting SCM Server](#) [4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)

GitHub

GitHub is a Web-based Git repository hosting service. It offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. Unlike Git, which is strictly a command-line tool, GitHub provides a Web-based graphical interface and desktop as well as mobile integration. It also provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

[0. Intro](#) [6.3. An Dojo Implementation of the Calendar Picker Script](#) [1.6. Servlet Handling Data, A Round House Kick](#)
[6.1. Consistent Multi-Value Formatting in SSJS for Domino/XPages](#) [7.2. Self-Hosting SCM Server](#)
[4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)

GSON

Gson (also known as Google Gson) is an open source Java library to serialize and deserialize Java objects to (and from) JSON.

[2.1. RESTful API Consumption on the Server \(Java\)](#) [2.2. Server REST Consumption with Authentication](#)
[2.3. Custom JSON with Java-ized XAgent](#) [5.2. Building Java Objects from JSON](#) [1.1. A Quick Note on JARs](#)
[1.6. Servlet Handling Data, A Round House Kick](#) [4.1. "Replacing" an XAgent](#) [1.9. Series Review](#) [1. A Saga of Servlets](#)

HATEOAS

HATEOAS, an abbreviation for Hypermedia as the Engine of Application State, is a constraint of the REST application architecture that distinguishes it from most other network application architectures. The principle is that a client interacts with a network application entirely through hypermedia provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia. By contrast, in some service-oriented architectures (SOA), clients and servers interact through a fixed interface shared through documentation or an interface description language (IDL).

[1.7. Building a Front-End pt.1 Plus a Quick Review](#)

Hg

Mercurial is a cross-platform, distributed revision control tool for software developers. It is mainly implemented using the Python programming language, but includes a binary diff implementation written in C. It is supported on MS Windows and Unix-like systems, such as FreeBSD, Mac OS X and Linux. Mercurial is primarily a command line program but graphical user interface extensions are available. All of Mercurial's operations are invoked as arguments to its driver program hg, a reference to the chemical symbol of the element mercury.

[7.2. Self-Hosting SCM Server](#)

IBM

International Business Machines Corporation (commonly referred to as IBM) is an American multinational technology and consulting corporation, with headquarters in Armonk, New York. IBM manufactures and markets computer hardware, middleware and software, and offers infrastructure, hosting and consulting services in areas ranging from mainframe computers to nanotechnology.

[0. Intro](#) [2.1. RESTful API Consumption on the Server \(Java\)](#) [2.3. Custom JSON with Java-ized XAgent](#)
[6.2. Fixing an Older Version of Dojo \(1.6.1\)](#) [3.1. Unraveling the M-V-C Mysteries](#)
[3.4. What is a Single Page Application\(SPA\)?](#) [5.1. When You Need a Comparator](#) [5.2. Building Java Objects from JSON](#)
[1.1. A Quick Note on JARs](#) [1.3. Basic Servlet Implementation](#) [1.4. Servlet Handling of Requests](#)
[1.6. Servlet Handling Data, A Round House Kick](#) [4.1. "Replacing" an XAgent](#) [1.9. Series Review](#)
[7.2. Self-Hosting SCM Server](#) [1. A Saga of Servlets](#)

Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere" (WORA), meaning that compiled Java code can run on all platforms that support Java without the need for recompilation. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture. As of 2015, Java is one of the most popular programming languages in use, particularly for client-server web applications, with a reported 9 million developers.

[2.1. RESTful API Consumption on the Server \(Java\)](#) [2.2. Server REST Consumption with Authentication](#)
[2.3. Custom JSON with Java-ized XAgent](#) [3.1. Unraveling the M-V-C Mysteries](#) [3. Application Logic](#)
[5.2. Building Java Objects from JSON](#) [1.1. A Quick Note on JARs](#) [1.2. Servlet Intro and Flavors](#)
[1.3. Basic Servlet Implementation](#) [1.4. Servlet Handling of Requests](#) [1.5. Interlude and Announcement](#)
[1.6. Servlet Handling Data, A Round House Kick](#) [6.1. Consistent Multi-Value Formatting in SSJS for Domino/XPages](#)
[4.1. "Replacing" an XAgent](#) [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
[4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#) [3.3. More on HTTP and AJAX Requests](#) [5. Java](#)
[1. A Saga of Servlets](#)

Java Server Faces

JavaServer Faces (JSF) is a Java specification for building component-based user interfaces for web applications. It was formalized as a standard through the Java Community Process and is part of the Java Platform, Enterprise Edition.

[3.1. Unraveling the M-V-C Mysteries](#)

Java Server Pages

JavaServer Pages (JSP) is a technology that helps software developers create dynamically generated web pages based on HTML, XML, or other document types. Released in 1999 by Sun Microsystems, JSP is similar to PHP and ASP, but it uses the Java programming language.

Java Virtual Machine

A Java virtual machine (JVM) is an abstract computing machine that enables a computer to run a Java program.

JavaScript

JavaScript is a high-level, dynamic, untyped, and interpreted programming language. It has been standardized in the ECMAScript language specification. Alongside HTML and CSS, it is one of the three essential technologies of World Wide Web content production; the majority of websites employ it and it is supported by all modern web browsers without plug-ins. JavaScript is prototype-based with

first-class functions, making it a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles. It has an API for working with text, arrays, dates and regular expressions, but does not include any I/O, such as networking, storage or graphics facilities, relying for these upon the host environment in which it is embedded.

[2.1. RESTful API Consumption on the Server \(Java\)](#) [2.3. Custom JSON with Java-ized XAgent](#)
[3.1. Unraveling the M-V-C Mysteries](#) [3.2. REST is Best](#) [3.4. What is a Single Page Application\(SPA\)?](#)
[3. Application Logic](#) [5.2. Building Java Objects from JSON](#) [4.1. "Replacing" an XAgent](#)
[1.7. Building a Front-End pt.1 Plus a Quick Review](#) [1.8. Building a Front-End pt.2: An App with AngularJS](#) [6. JavaScript](#)

JavaServer Faces

JavaServer Faces (JSF) is a Java specification for building component-based user interfaces for web applications. It was formalized as a standard through the Java Community Process and is part of the Java Platform, Enterprise Edition.

JavaServer Pages

JavaServer Pages (JSP) is a technology that helps software developers create dynamically generated web pages based on HTML, XML, or other document types. Released in 1999 by Sun Microsystems, JSP is similar to PHP and ASP, but it uses the Java programming language.

Jenkins CI

Jenkins is an open source continuous integration tool written in Java. The project was forked from Hudson after a dispute with Oracle.

jQuery

jQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML. jQuery is the most popular JavaScript library in use today, with installation on 65% of the top 10 million highest-trafficked sites on the Web. jQuery is free, open-source software licensed under the MIT License.

[6.3. An Dojo Implementation of the Calendar Picker Script](#) [1.6. Servlet Handling Data, A Round House Kick](#)
[1.7. Building a Front-End pt.1 Plus a Quick Review](#)

JSF

JavaServer Faces (JSF) is a Java specification for building component-based user interfaces for web applications. It was formalized as a standard through the Java Community Process and is part of the Java Platform, Enterprise Edition.

[2.1. RESTful API Consumption on the Server \(Java\)](#) [3.1. Unraveling the M-V-C Mysteries](#) [1.2. Servlet Intro and Flavors](#)
[4.1. "Replacing" an XAgent](#) [3.3. More on HTTP and AJAX Requests](#)

JSON

JSON (JavaScript Object Notation) is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is the primary data format used for asynchronous browser/server communication, largely replacing XML (used by AJAX).

- [2.1. RESTful API Consumption on the Server \(Java\)](#)
- [2.2. Server REST Consumption with Authentication](#)
- [2.3. Custom JSON with Java-ized XAgent](#)
- [3.2. REST is Best](#)
- [3.4. What is a Single Page Application\(SPA\)?](#)
- [5.2. Building Java Objects from JSON](#)
- [1.1. A Quick Note on JARs](#)
- [1.2. Servlet Intro and Flavors](#)
- [1.4. Servlet Handling of Requests](#)
- [1.6. Servlet Handling Data, A Round House Kick](#)
- [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
- [1.8. Building a Front-End pt.2: An App with AngularJS](#)
- [4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)
- [1. A Saga of Servlets](#)

JSON-RPC

JSON-RPC is a remote procedure call protocol encoded in JSON. It is a very simple protocol (and very similar to XML-RPC), defining only a handful of data types and commands. JSON-RPC allows for notifications (data sent to the server that does not require a response) and for multiple calls to be sent to the server which may be answered out of order.

- [3.3. More on HTTP and AJAX Requests](#)

JSONP

JSONP (or JSON with Padding) is a technique used by web developers to overcome the cross-domain restrictions imposed by browsers to allow data to be retrieved from systems other than the one the page was served by.

- [2.1. RESTful API Consumption on the Server \(Java\)](#)

JSP

JavaServer Pages (JSP) is a technology that helps software developers create dynamically generated web pages based on HTML, XML, or other document types. Released in 1999 by Sun Microsystems, JSP is similar to PHP and ASP, but it uses the Java programming language.

- [3.1. Unraveling the M-V-C Mysteries](#)

JVM

A Java virtual machine (JVM) is an abstract computing machine that enables a computer to run a Java program.

- [1.1. A Quick Note on JARs](#)
- [1.5. Interlude and Announcement](#)
- [1.7. Building a Front-End pt.1 Plus a Quick Review](#)

Mercurial

Mercurial is a cross-platform, distributed revision control tool for software developers. It is mainly implemented using the Python programming language, but includes a binary diff implementation written in C. It is supported on MS Windows and Unix-like systems, such as FreeBSD, Mac OS X and Linux. Mercurial is primarily a command line program but graphical user interface extensions are available. All of Mercurial's operations are invoked as arguments to its driver program hg, a reference to the chemical symbol of the element mercury.

- [7.2. Self-Hosting SCM Server](#)
- [4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)

Node

Node.js is an open-source, cross-platform runtime environment for developing server-side web applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux, FreeBSD, NonStop, IBM AIX, IBM System z and IBM i. Its work is hosted and supported by the Node.js Foundation, a collaborative project at Linux Foundation.

[6.3. An Dojo Implementation of the Calendar Picker Script](#) [6.2. Fixing an Older Version of Dojo \(1.6.1\)](#)
[7.1. A Brief Introduction to Nginx](#)

NodeJS

Node.js is an open-source, cross-platform runtime environment for developing server-side web applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux, FreeBSD, NonStop, IBM AIX, IBM System z and IBM i. Its work is hosted and supported by the Node.js Foundation, a collaborative project at Linux Foundation.

[1.4. Servlet Handling of Requests](#)

Notes

IBM Notes (formerly Lotus Notes) is a client of a collaborative client-server software platform sold by IBM.

[7.3. Domino/XPages Design Element Syntax Highlighting on Redmine](#) [2.1. RESTful API Consumption on the Server \(Java\)](#)
[2.2. Server REST Consumption with Authentication](#) [5.1. When You Need a Comparator](#) [1.1. A Quick Note on JARs](#)
[1.4. Servlet Handling of Requests](#) [1.5. Interlude and Announcement](#) [1.6. Servlet Handling Data, A Round House Kick](#)
[6.1. Consistent Multi-Value Formatting in SSJS for Domino/XPages](#) [7.2. Self-Hosting SCM Server](#) [1. A Saga of Servlets](#)

Remote Procedure Call

In computer science, a remote procedure call (RPC) is client/server system in which a computer program causes a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC might be called remote invocation or remote method invocation (RMI). Many different (often incompatible) technologies have been used to implement the concept.

REST

In computing, Representational State Transfer (REST) is the software architectural style of the World Wide Web. REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher-performing and more maintainable architecture. To the extent that systems conform to the constraints of REST they can be called RESTful. RESTful systems typically, but not always, communicate over HTTP with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve web pages and to send data to remote servers. REST interfaces with external systems using resources identified by URI, for example /people/tom, which can be operated upon using standard verbs, such as DELETE /people/tom.

[2.1. RESTful API Consumption on the Server \(Java\)](#) [2.2. Server REST Consumption with Authentication](#)
[3.1. Unraveling the M-V-C Mysteries](#) [3.2. REST is Best](#) [3. Application Logic](#) [1.3. Basic Servlet Implementation](#)
[1.4. Servlet Handling of Requests](#) [7.1. A Brief Introduction to Nginx](#) [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
[4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)

RESTful API

An API which provides access via RESTful principles, as in HATEOAS.

[2.2. Server REST Consumption with Authentication](#) [3.1. Unraveling the M-V-C Mysteries](#) [3.2. REST is Best](#)
[1.4. Servlet Handling of Requests](#) [4.1. "Replacing" an XAgent](#) [1.7. Building a Front-End pt.1 Plus a Quick Review](#)
[1.9. Series Review](#) [3.3. More on HTTP and AJAX Requests](#) [1. A Saga of Servlets](#)

RPC

In computer science, a remote procedure call (RPC) is client/server system in which a computer program causes a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC might be called remote invocation or remote method invocation (RMI). Many different (often incompatible) technologies have been used to implement the concept.

SCM

A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

[7.2. Self-Hosting SCM Server](#)

Server-Side JavaScript

Server-Side JavaScript (SSJS) is an implementation of JavaScript which executes on the server and has access to certain server-side APIs. XPages includes an SSJS as a scriptable language, parse inside `{javascript: }` or `{javascript: }` blocks and in SSJS script libraries, making its implementation quite convenient as it plugs in equivalently to Expression Language (with an interpretation prefix).

Service Orientated Architecture

A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any vendor, product or technology. A service is a self-contained unit of functionality, such as retrieving an online bank statement. By that definition, a service is a discretely invocable operation. However, in the Web Services Definition Language (WSDL), a service is an interface definition that may list several discrete services/operations. And elsewhere, the term service is used for a component that is encapsulated behind an interface. This widespread ambiguity is reflected in what follows.

SOA

A service-oriented architecture (SOA) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network. The principles of service-orientation are independent of any vendor, product or technology. A service is a self-contained unit of functionality, such as retrieving an online bank statement. By that definition, a service is a discretely invocable

operation. However, in the Web Services Definition Language (WSDL), a service is an interface definition that may list several discrete services/operations. And elsewhere, the term service is used for a component that is encapsulated behind an interface. This widespread ambiguity is reflected in what follows.

SOAP

SOAP, originally an acronym for Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML Information Set for its message format, and relies on other application layer protocols, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission.

[2.1. RESTful API Consumption on the Server \(Java\)](#) [2.2. Server REST Consumption with Authentication](#)

Source Control

A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

[7.3. Domino/XPages Design Element Syntax Highlighting on Redmine](#) [7.2. Self-Hosting SCM Server](#)
[4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#)

Source Control Management

A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information. Changes are usually identified by a number or letter code, termed the "revision number," "revision level," or simply "revision." For example, an initial set of files is "revision 1." When the first change is made, the resulting set is "revision 2," and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

SSJS

Server-Side JavaScript (SSJS) is an implementation of JavaScript which executes on the server and has access to certain server-side APIs. XPages includes an SSJS as a scriptable language, parse inside ``#{javascript: }`` or ``${javascript: }`` blocks and in SSJS script libraries, making its implementation quite convenient as it plugs in equivalently to Expression Language (with an interpretation prefix).

[2.1. RESTful API Consumption on the Server \(Java\)](#) [2.2. Server REST Consumption with Authentication](#)
[2.3. Custom JSON with Java-ized XAgent](#) [3.1. Unraveling the M-V-C Mysteries](#) [3. Application Logic](#) [4.1. "Replacing" an XAgent](#)
[1.7. Building a Front-End pt.1 Plus a Quick Review](#) [3.3. More on HTTP and AJAX Requests](#)

WSDL

The Web Services Description Language (WSDL) is an XML-based interface definition language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service (also referred to as a WSDL file), which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. It thus serves a purpose that corresponds roughly to that of a method signature in a programming language.

[2.2. Server REST Consumption with Authentication](#)

XHR

XMLHttpRequest (XHR) is an API available to web browser scripting languages such as JavaScript. It is used to send HTTP or HTTPS requests to a web server and load the server response data back into the script. Development versions of all major browsers support URI schemes beyond http and https, in particular, blob URLs are supported.

[3.1. Unraveling the M-V-C Mysteries](#) [1.6. Servlet Handling Data, A Round House Kick](#)

XML

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format which is both human-readable and machine-readable. It is defined by the W3C's XML 1.0 Specification and by several other related specifications, all of which are free open standards.

[7.3. Domino/XPages Design Element Syntax Highlighting on Redmine](#) [2.1. RESTful API Consumption on the Server \(Java\)](#)
[2.3. Custom JSON with Java-ized XAgent](#) [3.1. Unraveling the M-V-C Mysteries](#) [3.2. REST is Best](#)
[5.1. When You Need a Comparator](#) [5.2. Building Java Objects from JSON](#) [1.2. Servlet Intro and Flavors](#)

XML-RPC

XML-RPC is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism. "XML-RPC" also refers generically to the use of XML for remote procedure call, independently of the specific protocol. This article is about the protocol named "XML-RPC".

[2.1. RESTful API Consumption on the Server \(Java\)](#)

XMLHttpRequest

XMLHttpRequest (XHR) is an API available to web browser scripting languages such as JavaScript. It is used to send HTTP or HTTPS requests to a web server and load the server response data back into the script. Development versions of all major browsers support URI schemes beyond http and https, in particular, blob URLs are supported.

[3.1. Unraveling the M-V-C Mysteries](#)

XPages

XPages is a rapid web and mobile application development technology. It allows data from IBM Notes and Relational Databases to be displayed to browser clients on all platforms. The programming model is based on web development languages and standards including JavaScript, Ajax, Java, the Dojo Toolkit, Server-side JavaScript and JavaServer Faces. XPages uses IBM Domino, IBM's rapid application development platform, including functionality such as the document-oriented database.

[0. Intro](#) [7.3. Domino/XPages Design Element Syntax Highlighting on Redmine](#)
[6.3. An Dojo Implementation of the Calendar Picker Script](#) [2.2. Server REST Consumption with Authentication](#)
[2.3. Custom JSON with Java-ized XAgent](#) [3.1. Unraveling the M-V-C Mysteries](#) [3.2. REST is Best](#)
[3.4. What is a Single Page Application\(SPA\)?](#) [3. Application Logic](#) [5.1. When You Need a Comparator](#)
[1.3. Basic Servlet Implementation](#) [1.4. Servlet Handling of Requests](#) [1.5. Interlude and Announcement](#)
[1.6. Servlet Handling Data, A Round House Kick](#) [6.1. Consistent Multi-Value Formatting in SSJS for Domino/XPages](#)
[4.1. "Replacing" an XAgent](#) [1.7. Building a Front-End pt.1 Plus a Quick Review](#) [1.9. Series Review](#)
[4.2. Alternate Front-End Development: Mocking Your Domino Back-end](#) [3.3. More on HTTP and AJAX Requests](#) [1. A Saga of Servlets](#)