Introduction to Cocoon, XML XSL

By Olivier Eymere

Table of Contents

Intro.....1

Getting the tools.....2

Installing Tomcat.....3

Installing Cocoon.....5

Defining your document.....8

Creating your xml file.....10

Viewing your document in an HTML browser.....12

Viewing your document in a WAP browser.....16

Viewing your file as a pdf.....19

Intro

As problems go, It grows every time you turn around. Just how are you going to deal with all of these new devices? Suddenly everything is internet ready and your site has to accommodate them all. Figuring out how to deal with Netscape and IE were nothing compared to what is coming up. People will soon be viewing your site with a text browser that only show 6 lines at a time. Forget graphics. Your site has to be reduced to the basics. Html is not good for these new browsers so you have to learn a new markup language. When web services and SOAP become mainstream you may not be presenting your data in a browser at all. The worst part is that you have to write all of these So, how do you support all of these new devices, manage to keep your data consistent and actually keep your sanity?

One of your largest problems is html. Html is is presentation oriented. When you write html the focus is on how the page will look. The actual data in the page is secondary to the layout. There are now many tools to help make web pages look nice but they do not help make sense of the data contained in the pages.

XML, in contrast, is data oriented. When you create an XML document you focus on what the data is without concern for layout or presentation. Once you have the data and its structure down, you then write stylesheets and focus on how the data will be presented. The great thing about this approach is that you can present the same data in different ways. XML can greatly reduce the amount of work you need to do ensure that your data stays consistent no matter what media your users are using. This may not be particularly useful for your home page but it can do amazing things to help manage volumes of company data, documentation, contacts, schedules or orders.

This tutorial will guide you through setting up Tomcat and Cocoon to serve XML pages, then you will create a DTD, XML file and three XSL stylesheets so that you can view your data in your desktop browser, a cell phone browser and a pdf file. Before getting started you should be warned that writing pages in XML requires more time up front than HTML. By the end of the tutorial you will see the value in taking the extra time.

At the end of this tutorial you will have:
1. Installed and configured Tomcat to serve up xml documents
2. Installed Cocoon to process xml documents and format it according to your xsl documents.
3. Created a dtd to define the structure of you xml document.
4. Created an xml document containing an address book entry.
5. Created three xsl files to format the xml document in HTML,WML and pdf formats

Getting the tools

If you do not already have java installed you will need jdk 1.1 or higher. You can get the Linux jdk from http://blackdown.org

For this tutorial you will need:

Tomcat 3.2.1
For this tutorial we will use the binary build. Get it at http://jakarta.apache.org/site/binindex.html. In a production environment you would also want to download the Apache web server and configure Apache and Tomcat to work together. In the interest of keeping this tutorial a reasonable length we will not be configuring Tomcat and Apache to work together. So you will be accessing Tomcat directly using port 8080 instead of the usual port 80. If you want to configure Apache to pass requests on to Tomcat see the documentation at the jakarta web site.

Cocoon 1.8.2
Get it at http://xml.apache.org/cocoon/dist/

UP SDK from phone.com (now openware.com)
Unfortunately you must register before downloading the sdk and it only works on Windows. If you are going through this tutorial in Linux you will need another Windows machine to use the phone browser. Get it at http://www.phone.com/products/index.html

Installing Tomcat

Tomcat is the Apache Group's Java Servlet and JavaServer Page server. We need Tomcat for the Cocoon servlet to run.

Installing Tomcat is straight forward: all you need to do is unpack the file you downloaded, set JAVA_HOME and TOMCAT_HOME variables and start Tomcat.

The first step is to unpack Tomcat:

```
$ cd /usr/local/
$ gunzip jakarta-tomcat-3.2.1.tar.gz
$ tar -xvf jakarta-tomcat-3.2.1.tar
```

Then set up your environment:

```
$ JAVA_HOME=/path/to/jdk
$ TOMCAT_HOME=/usr/local/jakarta-tomcat-3.2.1
$ export TOMCAT_HOME JAVA_HOME
```

You can add these lines to the file $TOMCAT_HOME/bin/tomcat.sh just above the first 'if' statement if you do not feel like setting these variable everytime.

That is all you need to do, Tomcat is ready to run. To start Tomcat cd to $TOMCAT_HOME/bin and run:

```
$ ./startup.sh
```

After a few moments you should see two lines that look like this:

```
2001-01-26 12:25:02 - PoolTcpConnector: Starting
HttpConnectionHandler
on 8080
2001-01-26 12:25:02 - PoolTcpConnector: Starting
Ajp12ConnectionHandler
on 8007
```

When you see these line Tomcat is running and ready to serve. Open up a browser and go to http://localhost:8080 and run a couple samples to make sure that Tomcat is working properly. Run at least one servlet and one jsp sample. If you get any errors check your environment and restart Tomcat.

Installing Cocoon

Cocoon is Apache's XML publishing engine. Cocoon will read your xml file and format it according to the layout defined in your XSL files. The Cocoon distribution includes the cocoon.jar file and a set of jar files that help cocoon read and format xml files. Installing Cocoon is not much more complicated than installing Tomcat. You need to copy all of the jar files to a place where Tomcat knows to find them and update Tomcat's configuration.

First unpack Cocoon and change to the Cocoon directory:

```
$ tar -xzf Cocoon-1.8.2.tar.gz
$ cd cocoon-1.8.2
```

Next copy all of the jar files in lib/ to $TOMCAT_HOME/lib

```
$ cp lib/*.jar $TOMCAT_HOME/lib
$ cp bin/cocoon.jar $TOMCAT_HOME/lib
```

Tomcat 3.2.1 will automatically load all of the jar files found in $TOMCAT_HOME/lib so you do not have to add these to your CLASSPATH. However, if you are using java 1.2 or higher you will also need to add $JAVA_HOME/lib/tools.jar to your CLASSPATH. Cocoon uses tools.jar for page compilation. If you do not want to add tools.jar to your profile add this line to $TOMCAT_HOME/bin/tomcat.sh:

export CLASSPATH=$CLASSPATH:$JAVA_HOME/lib/tools.jar

The next step is to add a Context for Cocoon to Tomcat's server.xml file. This tells Tomcat to create a context for Cocoon. It also tells Tomcat where to find the cocoon files and what url path cocoon should be launched from. Change to $TOMCAT_HOME/conf and use an editor to open server.xml

At the bottom of the file you will see this tag </ContextManager>. Above that tag add these two lines:

```
<Context path="/cocoon" docBase="webapps/cocoon" debug="0"
reloadable="true">
```

```
</Context>
```

This tells Tomcat to create a context for cocoon with the path /cocoon. Whenever /cocoon is called Tomcat will look for the files in $TOMCAT_HOME/webapps/cocoon. Of course at this point there is no webapps/cocoon directory so let's create that next.

```
$ mkdir $TOMCAT_HOME/webapps/cocoon
$ mkdir $TOMCAT_HOME/webapps/cocoon/WEB-INF
```

The WEB−INF directory is part of the servlet 2.2 specification. According to the specification the WEB−INF directory should contain a file called web.xml which defines your servlets and any parameters the serlvets need. Most servlet engines will not strictly enforce these standards but trying to work around it will usually give you more headaches than solutions.

The next step is to copy the cocoon web.xml file and cocoon.properties files to your WEB−INF directory and copy the sample files to the webapps directory.

```
$ cd /path/to/cocoon-1.8
$ cp src/WEB-INF/web.xml $TOMCAT_HOME/webapps/cocoon/WEB-INF/
$ cp conf/cocoon.properties
$TOMCAT_HOME/webapps/cocoon/WEB-INF/
$ cp -R samples/ $TOMCAT_HOME/webapps/cocoon/samples
```

The last thing you need to do is edit the web.xml file you just copied. Open $TOMCAT_HOME/webapps/cocoon/WEB−INF/web.xml in an editor. You will see the line:

```
<param-value>[path-to-cocoon]/conf/cocoon.properties</param-value>
```

Change it to:

```
<param-value>WEB-INF/cocoon.properties</param-value>
```

Now stop and start Tomcat.

```
$ $TOMCAT_HOME/bin/shutdown.sh
$ $TOMCAT_HOME/bin/startup.sh
```

While Tomcat is loading you will see:

```
2001-01-26 01:35:04 - ContextManager: Adding context Ctx(
/cocoon )
```

If you see this then everything is good. Open a browser and go to http://localhost:8080/cocoon and try some of the samples. Not all of the samples are meant to be viewed in a browser so don't worry if you get errors with some of them. If you can see hello-page.xml then everything is working.

Defining your document

Your server environment is ready to go. Now we can actually start creating xml files. The first step is to create a Document Type Definition (dtd). The dtd tells the xml parser what information is allowed in the xml file and defines the format of the data. If the xml file does not fit into the constraints of the dtd the xml parser will give you an error and stop parsing the document. In our example we are creating a 'contact' element which has a name, address, phone number and email address. The dtd will define which tags must be part of a contact element and what kind of data will be in each tag.

It is not actually necessary to create a dtd for such a simple xml file, but it is important to use dtds when you are creating more sophisticated documents. Advantages of using dtds are that they ensure that your xml files are valid and well−formed and that you have all of the required data in the document. Using dtds also helps create standardization. Once you have created a dtd for a 'contact' it can be used in any place where contact information is required. This helps ensure that your data is consistent through out the company. Dtds also make it easier to communicate your data structures to others. If someone wants to interact with your applications or web site, for example, you could use dtds to show them your data structures.

To create your dtd, change directories to $TOMCAT_HOME/webapps/cocoon and create a directory called 'address'. Then change directories to address, create a file called contact.dtd and open it up in a text editor

The first step is to define you root element:

```
<!ELEMENT contact (name,address,phone,e−mail)>
```

In the line above we defined an element called contact which contains the elements name, address, phone and e−mail. Any time we use this dtd we must have a contact element which contains all of the child elements.

Next we will define the name element:

```
<!ELEMENT name (first−name,last−name)>
<!ELEMENT first−name (#PCDATA)>
<!ELEMENT last−name (#PCDATA)>
```

The first line defines the name element as containing first−name and last−name elements. The next two lines define first−name and last−name elements as text elements.

Following the same logic we can fill in the address, phone and e−mail elements.

```
<!ELEMENT address (street,city,state,country)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT e−mail (#PCDATA)>
```

The entire dtd looks like this:

```
<!−− contact.dtd −−>
<!ELEMENT contact (name,address,phone,e−mail)>
<!ELEMENT name (first−name,last−name)>
<!ELEMENT first−name (#PCDATA)>
<!ELEMENT last−name (#PCDATA)>
<!ELEMENT address (street,city,state,country)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT e−mail (#PCDATA)>
```

Creating your xml file

Now that we know what our data should look like we create the xml file. According to the dtd our xml file must contain all of these elements in (more or less) this structure:

```
<contact>
    <name>
        <first-name></first-name>
        <last-name></last-name>
    </name>
<address>
    <street></street>
    <city></city>
    <state></state>
    <country></country>
</address>
<phone></phone>
<e-mail></e-mail>
</contact>
```

To begin create a file called homer.xml and open it in a text editor. The first line of the xml file is the XML declaration:

```
<?xml version="1.0"?>
```

Every xml and xsl document should begin with the XML declaration. This declares that this is an xml document and which version of xml you are using.

Tags of the form <?...?> are processing instructions (PI). Processing instructions are instructions that are passed to the application that will be using the xml document. The first word after the <? is called the target which is the application that the instructions will be passed to. The rest of the PI contains the instructions to be passed to the target. In this case the target is the xml application and we are telling it that we are using xml version 1.0.

The next is our document type declaration. This is where we associate our xml file with the dtd we created. All declarations use the tag <!...>. The document type declaration looks like:

```
<!DOCTYPE contact SYSTEM "contact.dtd">
```

Next we want to add a PI for cocoon so that Cocoon knows that we want it to perform stylesheet transformations for us:

```
<?cocoon-process type="xslt"?>
```

Finally we can populate our file with data. At this point filling in the data is pretty simple, just add data values between the tags:

```
<contact>
<name>
<first-name>Homer</first-name>
<last-name>Simpson</last-name>
</name>
<address>
<street>122 West 1st Avenue</street>
<city>Springfield</city>
<state>ZZ</state>
<country>USA</country>
</address>
<phone>1-555-555-1111</phone>
<e-mail>h.simpson@springfieldnuclear.com</e-mail>
</contact>
```

With properly defined dtds and an xml template filling in data should be a relatively painless task.

Viewing your document in an HTML browser

We now have our xml file and a dtd to verify the validity of the document. You could view the xml file in an HTML browser but it would be quite uninteresting. The browser will load the xml file exactly as it appears in the text file. Your data will be presented correctly but users will want it to be more readable. Our next step is to create a stylesheet that contains HTML formatting information so that you can view the xml document as an html document.

To begin create a file called 'address−html.xsl' and open it using a text editor. As always the first line of the file is:

```
<?xml version="1.0"?>
Next we declare our document element and namespace by adding
this line:
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
```

The first part of the tag (xsl:stylesheet) is called the document element. Basically it tells the xml processor that the following document is an xsl:stylesheet. To close the document element we must end our document with:

```
<./xsl:stylesheet>
```

All data in the document must be enclosed within these tags.

The next part of the tag (xmlns:xsl="http://www.w3.org/1999/XSL/Transform") declares the namespace. XMLNamespaces (xmlns) are used to provide a way for the xml parser to make sense out of colliding vocabularies. Trying to create a namespace that is guaranteed to be unique in any and every context would be impossible. Without xmlns it is likely that your tags will collide with other tags at some point. For example suppose in our contact book we wanted to add the contact's job title. To do this we would add:

```
<title>Nuclear Technician</title>
```

to our xml file and update the dtd. We now have a problem because HTML also uses the tags <title> </title> but it means something totally different. The XMLNamespace is used to sort out which tags are which. As an aside, the link ("http://www.w3.org/1999/XSL/Transform) does not actually point to anything. If you want to know what that is all about check out http://www.w3.org/lpt/a/1999/01/namespaces.html.

Next we create our first xsl template. A template is a set of instructions and literal results that are executed when certain conditions are met by the processor. The first template we will create lays out the HTML page:

```
<xsl:template match="contact">
    <html>
        <head>

<title>

<xsl:value-of select="name/last-name"/>

<xsl:text>, </xsl:text>

<xsl:value-of select="name/first-name"/>

</title>
        </head>
        <body bgcolor="#ffffff">

<xsl:apply-templates/>
        </body>
    </html>
</xsl:template>
```

The match="contact" attribute of the template tag tells the processor to use the following template when it comes across a contact element.

The statement <xsl:value−of select=" "> is where we actually select data from the xml document. In this case we select name/last−name, name/first−name and use that for the title of page. You must also use <xsl:text> some text </xsl:text> where you want regular text in the HTML. With this template the title of the page will be "Simpson, Homer". Otherwise the template contains standard HTML that will layout the page. In the body of the HTML page we use xsl:apply−templates to tell the processor to apply the templates for the other elements of the xml document.

To fill in the body of the html document we will create templates for each of the elements of the xml document.

```
<xsl:template match="name">
    <h1 align="left">
        <xsl:apply-templates/>
    </h1>
</xsl:template>
Matches the name elements and formats the name with an h1 tag.
<xsl:template match="address">
    <i><xsl:text>Address: </xsl:text></i><br/>
    <xsl:value-of select="street"/><br/>
    <xsl:value-of select="city"/><br/>
    <xsl:value-of select="state"/><br/>
    <xsl:value-of select="country"/><br/>
</xsl:template>
```

This simply matches the address element and prints out each address element line by line.

```
<xsl:template match="phone"><br/>
    <xsl:text>Phone number: </xsl:text>
    <xsl:apply-templates/>
</xsl:template>
<xsl:template match="e-mail"><br/>
    <xsl:text>Email: </xsl:text>
    <a>
        <xsl:attribute name="href">

<xsl:text>mailto:</xsl:text><xsl:apply-templates/>
        </xsl:attribute>
        <xsl:apply-templates/>
    </a>
</xsl:template>
```

The last two templates match phone and e-mail and print the data out. For each element we grab the data, add some text and add it to the result tree. In the template that matches e-mail we also add html formatting to add a mailto: link to the e-mail address. The user only has to click on the email address link to create an email addressed to the contact. Since this page will be viewed in a web browser it seems likely that the user will want to send an email to the contact so why not make is easier?

Now that we have an xml and xsl document we need to tell the xml document to use the xsl stylesheet. To do this open up homer.xml in a browser and add the following processing instruction under the cocoon−process PI (<?cocoon−process type="xslt"?>)

```
<?xml−stylesheet href="address−html.xsl? type="text/xsl"?>
```

Everything is ready for viewing your first xml/xsl page in a browser. If tomcat is not running start it up and point your browser to the following URL:

http://localhost:8080/cocoon/address/homer.xml

If you see any Cocoon errors read it carefully. The error messages are generally quite good at telling you what is wrong. Xml is much more strict than html. Among other things case if followed strictly and all tags must be properly terminated. If you have xml syntax problems, Cocoon will usually tell you what is wrong.

Viewing your document in a WAP browser

One of the key features of xml is that you can view the same data in a variety of formats. Once you have created a contact book wouldn't it be nice to be able to use the same contact book whenever and wherever you need to look up a contact? Rather than writing conversion tools to reformat your contact for each media you can use different stylesheets on the same data.

As the use of web enabled cell phone and PDAs grows this approach can save you a lot of headaches. The standard for cell phone browsers is (for better or worse) wireless markup language (wml). We want to make our contact accessible from a cell phone browser. To do this we must now create a stylesheet for wml browsers. Create a file called 'address−wml.xsl' and open it in an editor.

The xsl instructions are essentially the same but we are now using wml constructs. If you are not familiar with wml you should check out Vikram Vaswani and Harish Kamath's excellent tutorial at http://www.devshed.com/Client_Side/XML/DemystifyingWML1/.

The top of the stylesheet adds a processing−instruction for the type text/wml so the wml can be interpreted and formatted.

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="contact">
<xsl:processing−instruction name="cocoon−format">
    type="text/wml"
</xsl:processing−instruction>
```

The body is a mix of xsl and wml instead of html but is quite similar otherwise. Because users will be viewing this page on a cell phone browser we should rethink the presentation of the data. Taking the small screen and intended use of the page into consideration the first page will only show the name and phone number. If the cell phone user wants more contact information there is a link to the full contact info.

```
<wml>
    <card id="index" title="Your Contacts">
        <p align="center">

<a href="#contact">Phone Book</a><br/>
        </p>
    </card>
<card id="contact" title="Phone Book">
```

```
    <p>
        <b><xsl:value-of select="name/first-name"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="name/last-name"/></b><br/>
        <xsl:value-of select="phone"/>
        <do type="accept" label="More">

<go href="#Address"/>
        </do>
    </p>
</card>
<card id="Address" title="address">
    <p>
        <b>
        <xsl:value-of select="name/first-name"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="name/last-name"/>
        </b><br/>
        <xsl:value-of select="address/street"/><br/>
        <xsl:value-of select="address/city"/><br/>
        <xsl:value-of select="address/state"/><br/>
        <xsl:value-of select="address/country"/><br/>
        <a href="#index">Main</a>
        <do type="prev">

<prev/>
        </do>
    </p>
</card>
</wml>
</xsl:template>
</xsl:stylesheet>
```

As with the html stylesheet you need to tell the xml processor to use addres-wml.xsl when a phone browser accesses the page. To do this open up homer.xml in an editor and add the following line:

```
<?xml-stylesheet href="address-wml.xsl? type="text/xsl?
media="wap"?>
```

Simply put, this line tells cocoon to use address−wml.xsl for formatting instructions when the media requesting the page is a wap browser.

Open up your phone.com browser and point it to the same URL you used to view the html page.

The data is the same but we have changed the document layout to make it more useful in a cell phone browser. We do not have all of the information in one page. Instead the first page shows the contacts name and phone number only. Assuming that a cell phone user is most likely to look up a phone number lets show the phone number first and not clutter up the small screen.

Viewing your file as a pdf

The last thing we do in this tutorial is convert the xml to a pdf file. Creating a pdf file is great when your users might want to keep or print a copy of the data. Since pdfs are not editable they are a great way to provide users with a document that they will not be able to accidentally (or purposely) modify later.

For pdf generation we will use FOP from Apache. FOP takes in an xml document, an FO stylesheet and outputs a pdf. FOP is actually a separate project from cocoon but the FOP jar is included in the cocoon distribution so you do not need to add anything new to the project. There are a lot of similarities with html or wml stylesheets but you will notice that the formatting is much more specific. You need to create a file called address−pdf.xsl and open it in a text browser. Begin the file with the usual xsl:stylesheet PI but you also need to say that you will be using the xml namespaces for xml and fo:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format">
```

Use an xsl template to match the contact item and add a processing−instruction for the type text/xslfo.

```
<xsl:template match="contact">
<xsl:processing−instruction
name="cocoon−format">type="text/xslfo"</xsl:processing−instruction>
```

For any fo file you first need an fo:root element. The fo:root element is essentially the same as the root xsl:template. It defines page layouts, page sequences and host of optional text formatting instructions.

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
```

Every fo:root must have one fo:layout−master−set. The fo:layout−master−set defines the layout specification including page margins.

```
<fo:layout-master-set>
<fo:simple-page-master
page-master-name="single"
margin-top="2cm"
margin-bottom="2cm"
margin-left="2.5cm"
margin-right="2.5cm">
<fo:region-body margin-bottom="3cm"/>
<fo:region-after extent="1.5cm"/>
</fo:simple-page-master>
</fo:layout-master-set>
```

Next add an fo:page-sequence to define how to create the pages within the document. Your entire document can be within one fo:page-sequence or you can use different page-sequence for sections.

```
<fo:page-sequence>
<fo:sequence-specification>
<fo:sequence-specifier-alternating
page-master-first="single"
page-master-odd="single"
page-master-even="single"/>
</fo:sequence-specification>
```

Next you need to add fo:flow to output the text. In our file we are applying the templates from the xml result tree.

```
<fo:flow>
<xsl:apply-templates/>
</fo:flow>
</fo:page-sequence>
Close the fo:root and xsl:template
</fo:root>
</xsl:template>
```

Finally we create xsl:templates for the elements of our contact item. In each template we have an fo:block which is used to define font, font size, alignment, etc.

```
<xsl:template match="name">
<fo:block font-size="12pt" text-align="justified">
<xsl:apply-templates/>
</fo:block>
</xsl:template>
<xsl:template match="address">
<fo:block font-size="12pt" text-align="justified">
<xsl:apply-templates/>
</fo:block>
</xsl:template>
<xsl:template match="phone">
<fo:block font-size="12pt" text-align-last="justified">
<xsl:apply-templates/>
</fo:block>
</xsl:template>
<xsl:template match="e-mail">
<fo:block font-size="12pt" space-before.optimum="12pt"
text-align="justified">
<xsl:apply-templates/>
</fo:block>
</xsl:template>
</xsl:stylesheet>
```

You will notice that the xsl syntax is similar to previous xml files but with fo formatting instruction. Now all that needs to be done is to tell your xml file to use address-pdf.xsl. For simplicity open homer.xml and replace:

```
<?xml-stylesheet href="address-html.xsl" type="text/xsl"?>
```

with:

```
<?xml-stylesheet href="address-pdf.xsl" type="text/xsl"?>
```

It is possible to have both the pdf and html file generated at the same time and have the pdf downloaded from a link in the html page but it is not a straight forward as you might think. I will save that for another day. All that is left for you now is to point your browser to http://localhost:8080/address/homer.xml. If you have an Acrobat plugin the pdf should load automatically. If you do not have a plugin you should be prompted to download the file. Once downloaded open Acrobat to view the file.

That covers this tutorial. If everything has gone well you have a good idea of how powerful xml can be. Hopefully it can help you manage your data and make your life a little easier