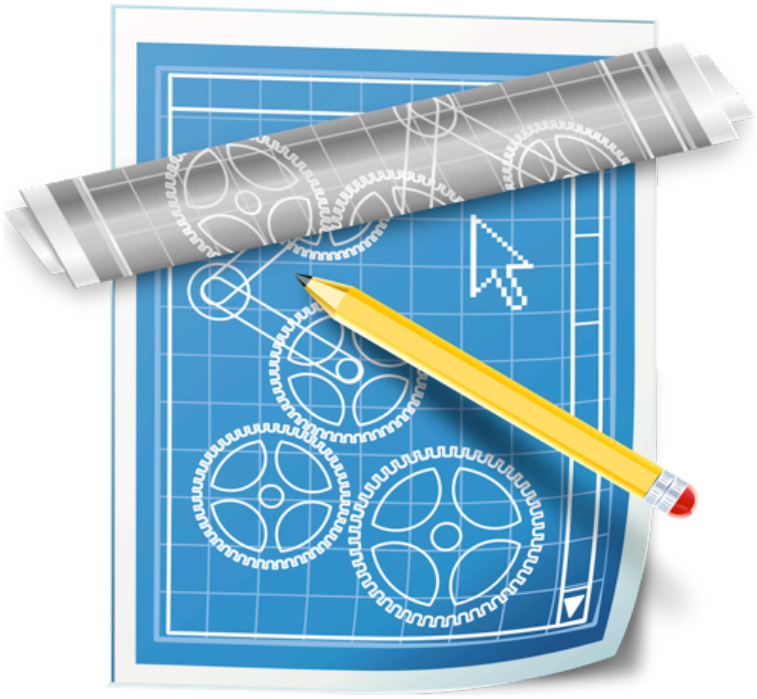
Bombax

Developer's Guide



Version 2.0

www.bombaxtic.com

Visit us on the web at http://www.bombaxtic.com/

Please consider the environment before printing this guide.

Document version history:

2.beta.0 — 2010/04/19
1.0.0 — 2010/01/26
1.beta.3 — 2010/01/23
1.beta.2 — 2010/01/18
1.beta.1 — 2010/01/13
1.beta.0 — 2010/01/10

Table of Contents

1 - Bombax Platform Overview **6**

1.1 - The Benefits of Bombax *6*

1.2 - How Bombax Works *8*

2 - Getting Started **10**

2.1 - Requirements *10*

2.2 - BxApp Tutorial *10*

3 - BXML Files **22**

3.1 Creating a BXML Handler *22*

3.2 - How BXML is Processed *24*

3.3 - Non-Enclosed Text *25*

3.4 - BXML Tags *25*

3.4.1 The <? Tag *25*

3.4.2 The <?setup Tag *25*

3.4.3 The <?import Tag *26*

3.4.4 The <?base Tag *26*

3.4.5 The <?paste Tag *26*

3.4.6 The <?static Tag *26*

3.4.7 The <?name Tag *27*

3.4.8 The <?-- Tag *27*

4 - Debugging BxApps **28**

4.1 - Debugging in Xcode *28*

4.2 - Profiling and Analyzing Your BxApp *28*

4.3 - Handling Exceptions and Crashes *29*

5 - Interfacing **30**

5.1 - Interfacing with Other Frameworks *30*

5.2 - Interfacing with C and C++ *31*

5.3 - Interfacing with Other Languages *33*

6 - Persisting Data **35**

6.1 - The state Property *35*

6.2 - Core Data *35*

6.3 - Using Databases *37*

6.4 - Configurators and NSUserDefaults *37*

7 - Packaging and Running your BxApp **40**

7.1 - Info.plist Properties *40*

7.2 - The Next Steps *41*

1 - Bombax Platform Overview

Bombax is an advanced web application platform for Mac OS X that uses the Objective-C language for development. If you have developed an iPhone or OS X application, you already have most of the experience necessary to create Bombax web applications. Unlike other web application platforms, Bombax is written especially for OS X developers and uses the highly regarded Cocoa framework.

If you are new to OS X development, Bombax can be an excellent way to learn Cocoa and OS X development, especially as Bombax developers usually do not need to use Interface Builder which can be complex. If this is your first use of Cocoa, we recommend that you begin by familiarizing yourself with the basic ideas of Cocoa and Objective-C development by visiting the Apple Development Center at http://developer.apple.com/ and reading the introductory documents.

When you write a Bombax web application you create a BxApp, which is a fully compiled Cocoa application that uses the Bombaxtic framework. When you deploy your BxApp, the Bombax server connects to your BxApp allowing it to handle web browser requests. Bombax and the Bombaxtic framework are specifically designed to make handling these requests as easy, secure, and efficient as possible.

1.1 - The Benefits of Bombax

Bombax makes web development faster and more powerful than ever before. Compared with other web application platforms such as Java EE, ASP.NET, and LAMP, Bombax has many significant advantages:

- Bombax uses Apple's Cocoa framework, one of the most advanced, modern, and refined frameworks available on any system. BxApps have access to the full library of Cocoa classes and functions, saving a great deal of infrastructure development time.

- Bombax's Bombaxtic framework makes many common dynamic website functions very easy such as sending e-mails, handling uploaded files, accessing static resources, localizing content, and connecting to databases.

- Bombax includes extensive database connection functionality, allowing sophisticated access to Oracle, PostgreSQL, MySQL, and SQLite in a pure Objective-C environment. Bombax servers include all necessary database drivers and have out-of-the-box support for BxApp database access to guarantee connectivity.

- The Bombax server core is based on Nginx, an extremely efficient and reliable HTTP server used by millions of high demand websites and outperforming nearly all other servers including Apache, Glassfish, and IIS.

- BxApps are more memory and CPU efficient than any other widely used type of web application. Because they are fully compiled Objective-C applications, performance intensive code can run as much as several orders of magnitude faster than PHP and other commonly used languages. BxApps can even utilize low-level optimizations such as multicore and SIMD programming.

- BxApps use FastCGI to communicate with the Bombax server. Because of this, BxApps support application persistence between client requests, multithreaded and multiprocess concurrency, and more secure operation.

- BxApps are automatically monitored and restarted using the watchdog functionality of the Bombax servers to ensure high reliability. Exceptions and low level errors are trapped through the Bombaxtic framework to allow easy handling through your BxApp.

- Bombax is easy to administer, using a complete GUI application to configure and operate the server. Virtual hosting, SSL, IPv6, FastCGI, and more is supported out of the box and installing a BxApp is as simple as drag and drop.

- BxApps allows you to write and use any Objective-C, C, or C++ code supported on OS X including frameworks and libraries without any kludgey mapping code like JNI. Literally thousands of libraries for all kinds of application are available to you.

- BXML (BombaX Markup Language) makes it very easy to embed Objective-C code directly into HTML documents in the style of JSP or PHP.

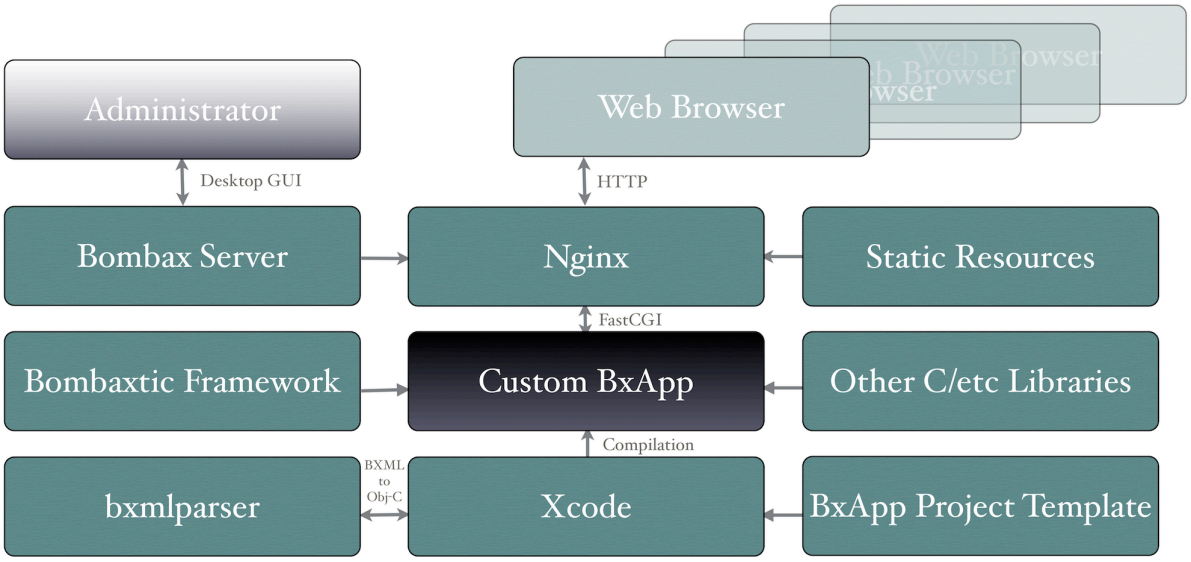
- Bombax web application development is fully integrated into Xcode through BxApp projects, BXML syntax highlighting, complete debugging support, and more.

- BxApps can provide full featured OS X GUI windows for application configuration. These GUI components can be developed in Xcode as part of the BxApp project.

- Bombax and BxApps work on Intel and PowerPC computers running OS X 10.5 and higher, with extra optimizations for 64-bit Intel processors. Your BxApps are easily distributable as a single bundle file and can be installed in seconds.

1.2 - How Bombax Works

Bombax development involves several key components as shown in this figure:



The Bombax GUI application provides a high level interface to configure, start, and stop instances and handlers. The server GUI automatically controls operation of a collection of server processes based on Nginx. When the server is started, these processes begin to listen for connections from clients according to the configuration.

When a connection is received and maps to your BxApp, a FastCGI channel communicates information about the client request which is passed to the `handlerForPath` method of your `BxApp` subclass which looks to see if a `BxHandler` is appropriate to handle the request. If the `BxHandler` has not been used before, it is instantiated and `setup` is called. The `BxHandler`'s `renderWithTransport:` method is then called, passing a `BxTransport` specific to the client request.

The `BxTransport` instance allows your handler to find out all available information about the request, including query variables, cookies, uploaded files, and more. `BxTransport` also provides buffered output to the client, making it easy to create HTML pages in your handler. You aren't limited to HTML though; you can output any kind of response to the client.

Within the `renderWithTransport:` method you can do anything possible in any other Objective-C method, including creating objects, calling other Objective-C methods or C functions, and accessing Cocoa or other frameworks. When

`renderWithTransport:` returns, the FastCGI channel is flushed and your output is sent to the client.

In addition to using BxApps, the Bombax server can also serve static data which takes place extremely quickly. This frees up your BxApp to focus on dynamic content and not piping JPEGs and other (normally) static files. Bombax supports generic FastCGI connections to generic FastCGI processes for legacy communication to e.g. PHP and Python web applications. Of course, once you are used to developing on the Bombax platform, you'll probably want to convert those applications to BxApps!

2 - Getting Started

2.1 - Requirements

To develop a BxApp, your computer must meet the following requirements:

— *OS X 10.5 or higher*
— *Xcode 3.0 or higher*
— *Intel or PowerPC G4/G5 Processor*
— *Bombax installed (with development options)*

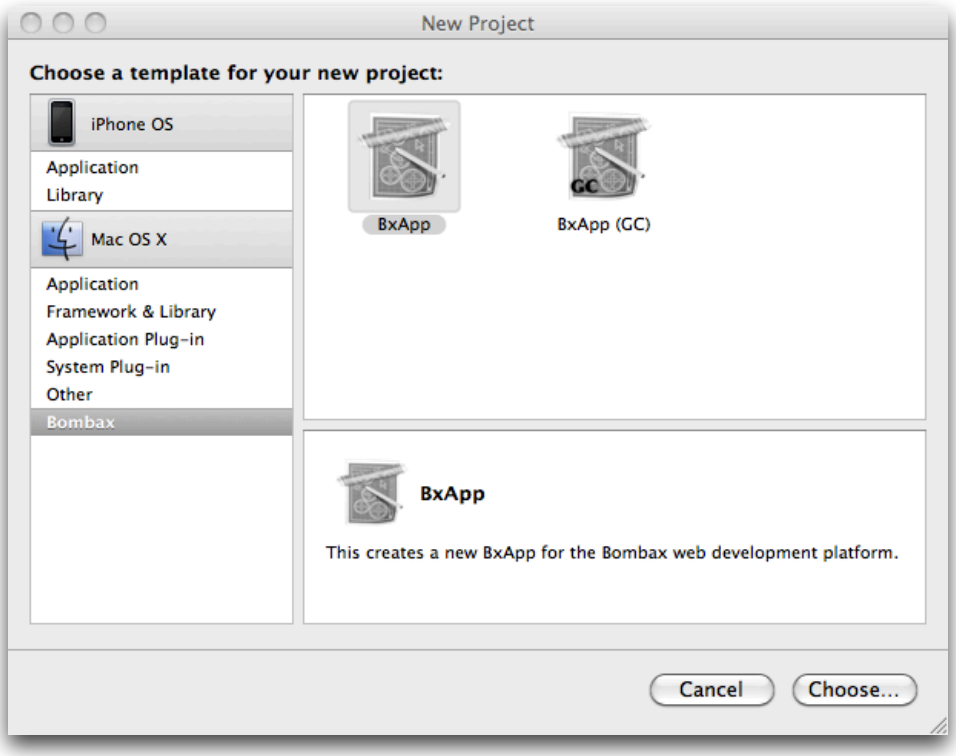
Xcode may be downloaded for free from http://developer.apple.com/. After installing Xcode run or rerun the Bombax installer and ensure that the all development options are selected.

2.2 - BxApp Tutorial

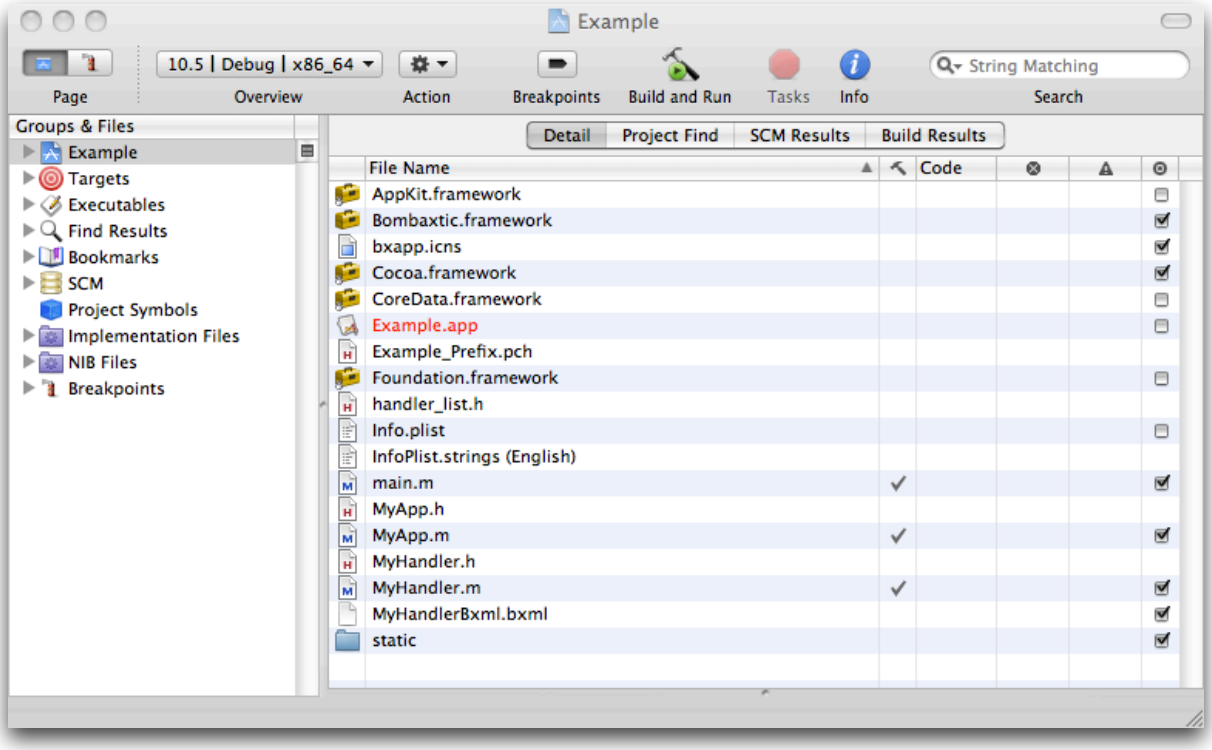
BxApps are a type of Xcode project and need to be create and built with Xcode. To create a BxApp, follow the following steps.

1: Open Xcode.

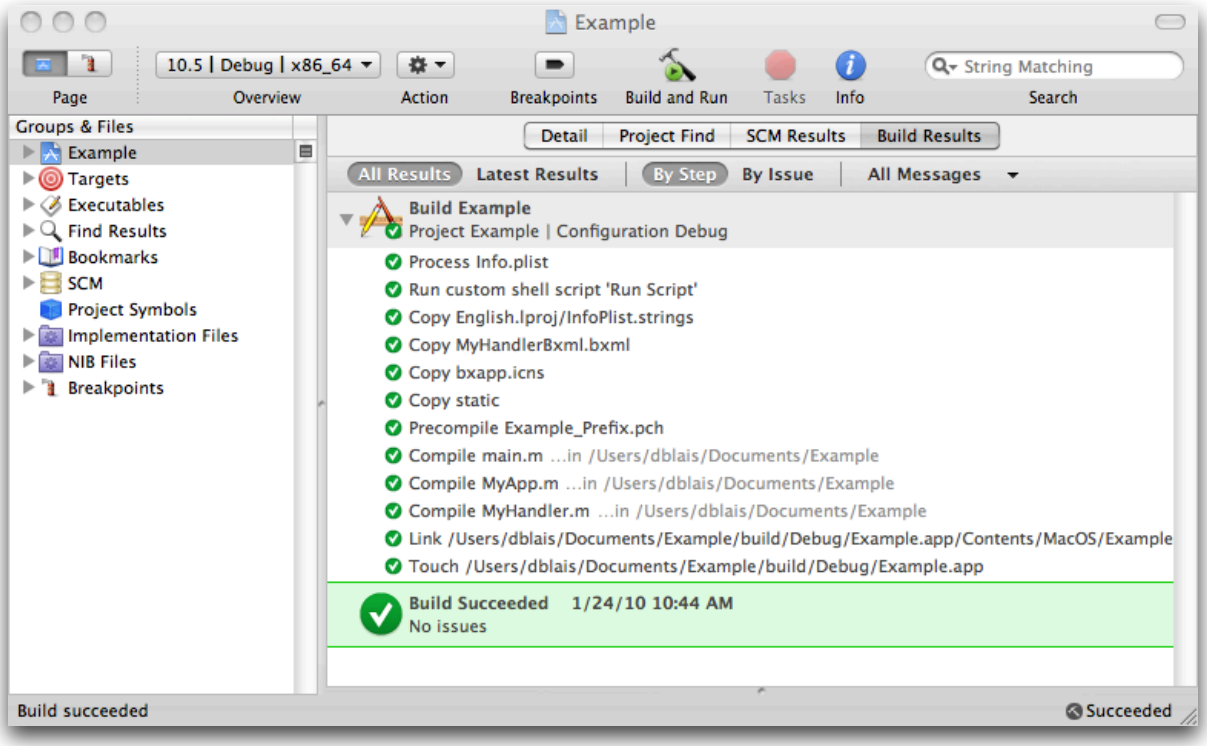
2: Select *New Project...* from the *File* menu, select "BxApp" or "BxApp (GC)" in the Bombax category, and press the *Choose...* button to select the location and name.



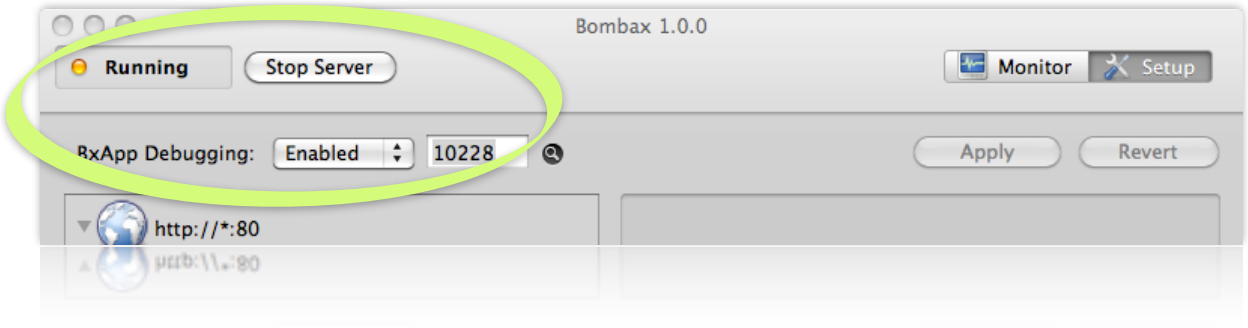
3: Your newly created BxApp project should look like the following:



From *Build* menu select the *Build* menu item. Your BxApp should build without any errors or warnings. In your project window, click the *Build Results* tab. It should look similar to this:

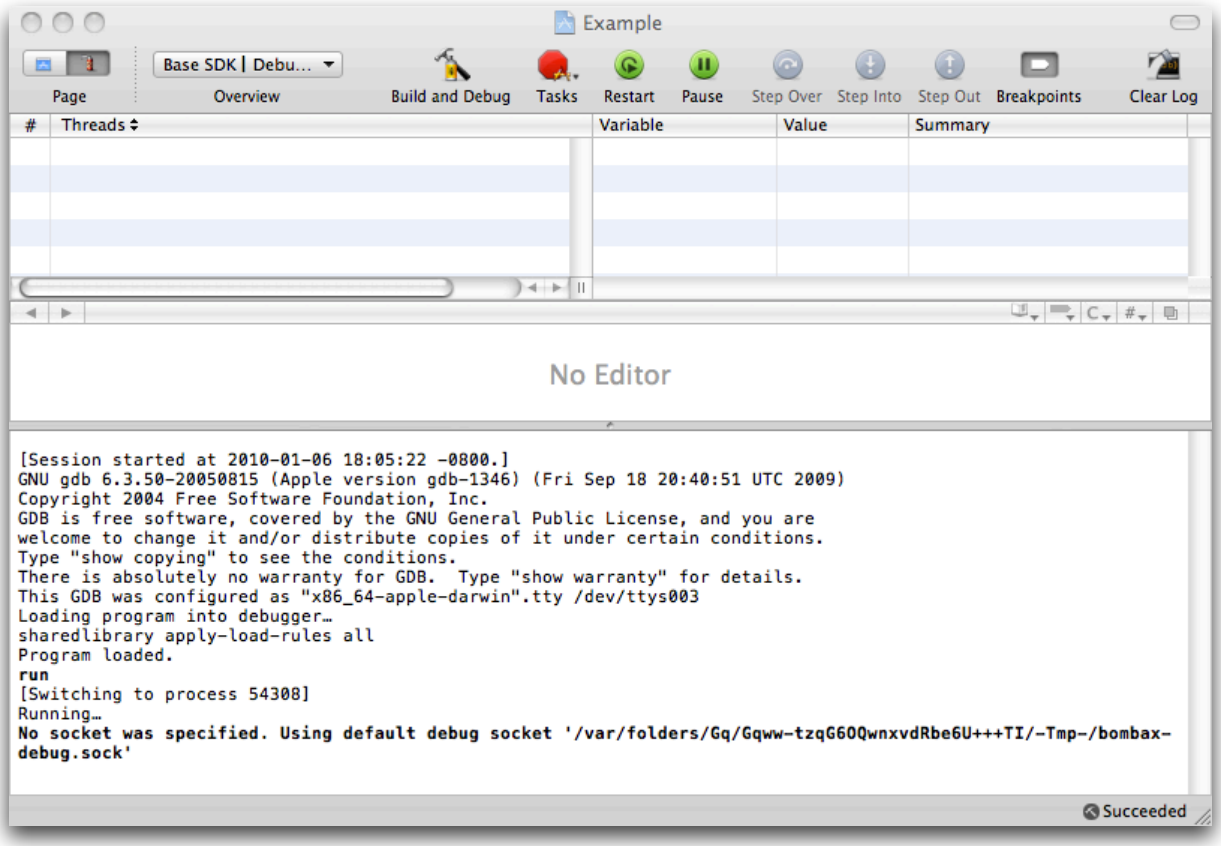


4: Make sure the Bombax server is running and debug is enabled on port 10228. For example:

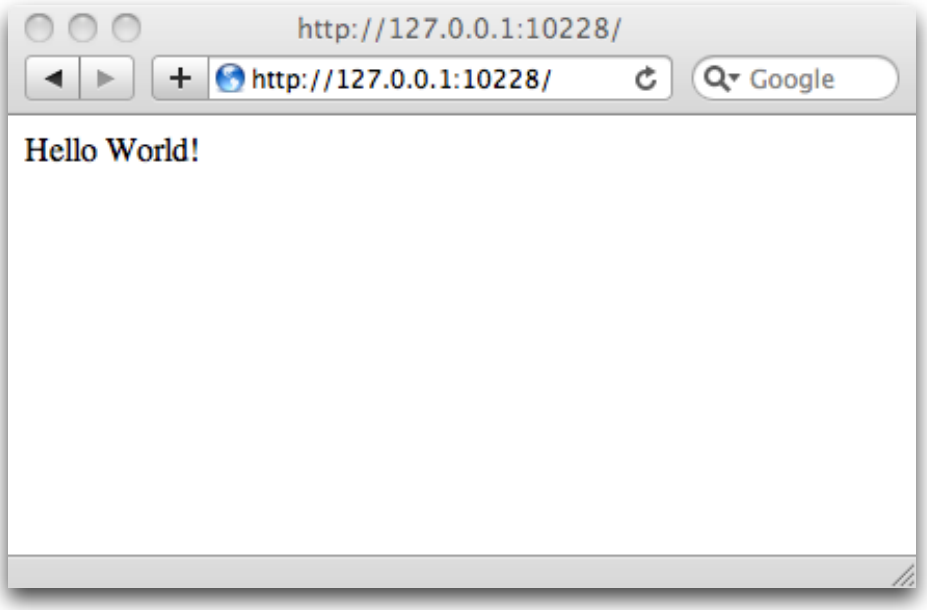


Note that the Bombax GUI application does not need to be running for the server to operate, only to adjust configuration and start or stop the server.

In your BxApp project, go to the *Run* menu and select the *Debug* menu item. Switch to the *Debug* page view in your project window and verify that the console output looks similar to the following:



5: Now that your BxApp is running, you can view it at the standard debug location of http://127.0.0.1:10228/. It should look like this:



Congratulations! You've just created and run your first BxApp.

6: Now that you are able to build, run, and test your application, let's take a look at how it works. Normally once the Bombax server gets a connection from a client, it checks its configuration and passes the connection to the BxApp that is registered for that pattern of path, hostname, and so on. If the debug checkbox is enabled in Bombax, it also matches any connections to http://127.0.0.1:10228/ to whatever BxApp is currently running in Xcode. So because debug is enabled, when you started your BxApp and visited http://127.0.0.1:10228/ the connection was passed by Bombax to your BxApp. Note that only one BxApp may be debugged at a time without additional configuration.

When your BxApp first starts, the `main` function in `main.m` is called, just as with any other Objective-C program. Let's take a look at the default contents of this file:

```
#import <Cocoa/Cocoa.h>
#import <Bombaxtic/Bombaxtic.h>
#import "handler_list.h"

int main(int argc, char *argv[])
{
    // If the name of the BxApp subclass changes, you must change this to match
    return BxMain("MyApp");
}
```

The import headers add the Cocoa and Bombaxtic frameworks as well as the special handler_list.h file. handler_list.h is automatically generated from your BXML files (see Section 3) and should not be modified directly. While Cocoa applications normally call NSApplicationMain as their entry point, BxApps call BxMain, which is defined in Bombaxtic.h.

BxMain sets up all of infrastructure necessary for your BxApp to work with the Bombax server and passes incoming client requests to your custom subclass of BxApp (note that that is the BxApp class, a part of the Bombaxtic framework), which in this case is MyApp. If your BxApp subclass changes, you need to make sure that the classname (a char*, not a NSString*) passed to BxMain is also changed, as the comment indicates.

Your BxApp subclass, MyApp, is defined in two files: MyApp.h and MyApp.m. Normally only one method in BxApp is overridden: setup. When a client request is received by MyApp, it needs to be passed to a subclass of BxHandler which provides whatever custom application logic is appropriate. BxApp's setup method is what configures the mapping of BxHandler subclasses to incoming requests. For example, the MyApp.m initially contains the following:

```
#import "MyApp.h"

@implementation MyApp

- (id)setup {
    [self setHandler:@"MyHandlerBxml" forMatch:@"/example"];
    [self setDefaultHandler:@"MyHandler"];
    return self;
}
@end
```

This sets up two BxHandler subclasses for incoming requests by referring to their classnames. MyHandlerBxml will handle paths that exactly match /example (you can see this at http://127.0.0.1:10228/example) and any other request is handled by the default handler, MyHandler.

These handlers are defined by other files in the project; MyHandler is defined by MyHandler.h and MyHandler.m while MyHandlerBxml is defined by MyHandlerBxml.bxml. If you were to add another handler, you'd need to add additional mapping code to MyApp so that Bombax knows when to pass client requests to it.

Now let's take a look at the `MyHandler` class and how it handles incoming requests. The code for the `MyHandler` implementation looks like this:

```
#import "MyHandler.h"

@implementation MyHandler

- (id)renderWithTransport:(BxTransport *)transport {
    [transport write:@"Hello World!"];
    return self;
}

@end
```

The key overridden method of `BxHandler` subclasses is `renderWithTransport:`. When a client request is handed to the handler, all the information about the request and the communications channel is passed to the handler through the `BxTransport` variable. In `MyHandler`, the handler simply writes 'Hello World!' to the client.

7: Now that we've seen how the request handling mechanism works, let's take change what MyHandler does when it responds to the client. Modify MyHandler.m to look like the following:

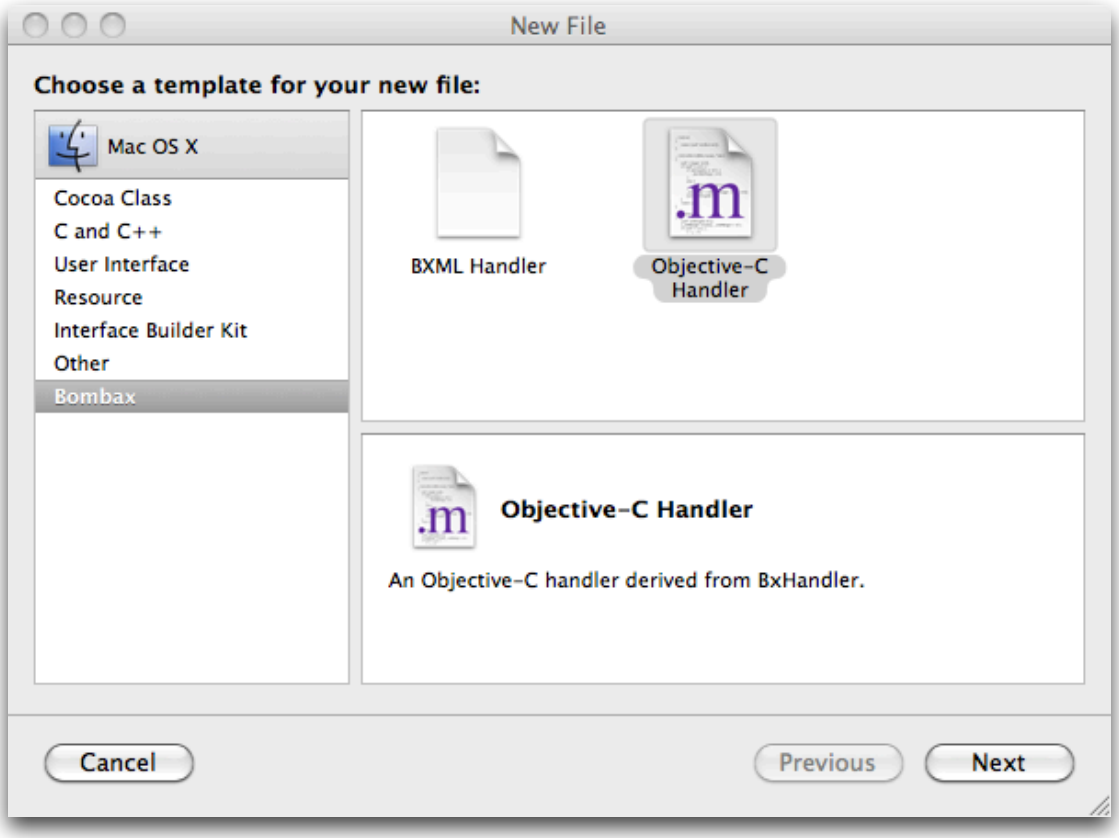
```
#import "MyHandler.h"

@implementation MyHandler

- (id)renderWithTransport:(BxTransport *)transport {
    [transport writeFormat:@"Query variables: %@", transport.queryVars];
    return self;
}

@end
```

Now restart your BxApp by selecting *Build and Debug* from the *Build* menu (or simply by pressing ⌘Y). If you visit http://127.0.0.1:10228/ it should now look like this:

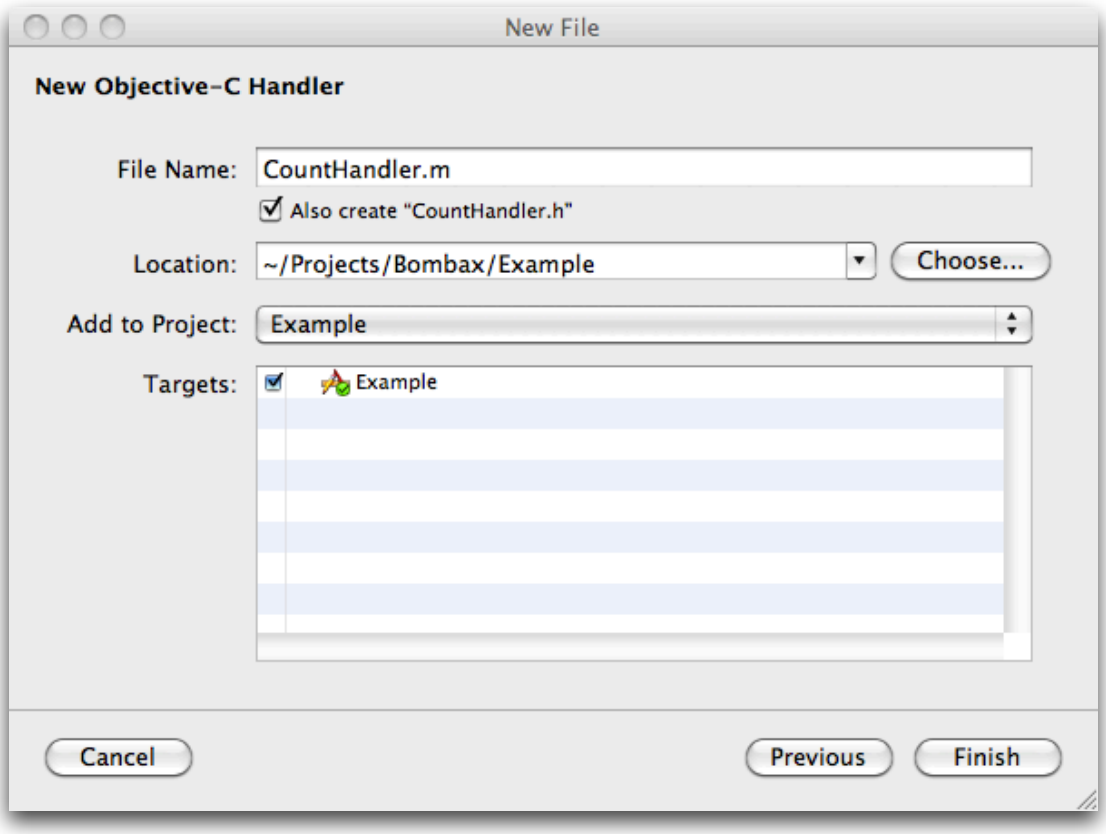
Note that you don't need to restart the Bombax server when you update your BxApp; just restart the BxApp from Xcode just like when testing a desktop application.

When the client request is processed, Bombaxtic assembles all of the query variables into a `NSDictionary*` property of `BxTransport` named `queryVars`. You've modified MyHandler to write out the contents of the dictionary when it responds to a request. Now let's access the same handler but add a query variable as with http://127.0.0.1:10228/?flavor=sweet :

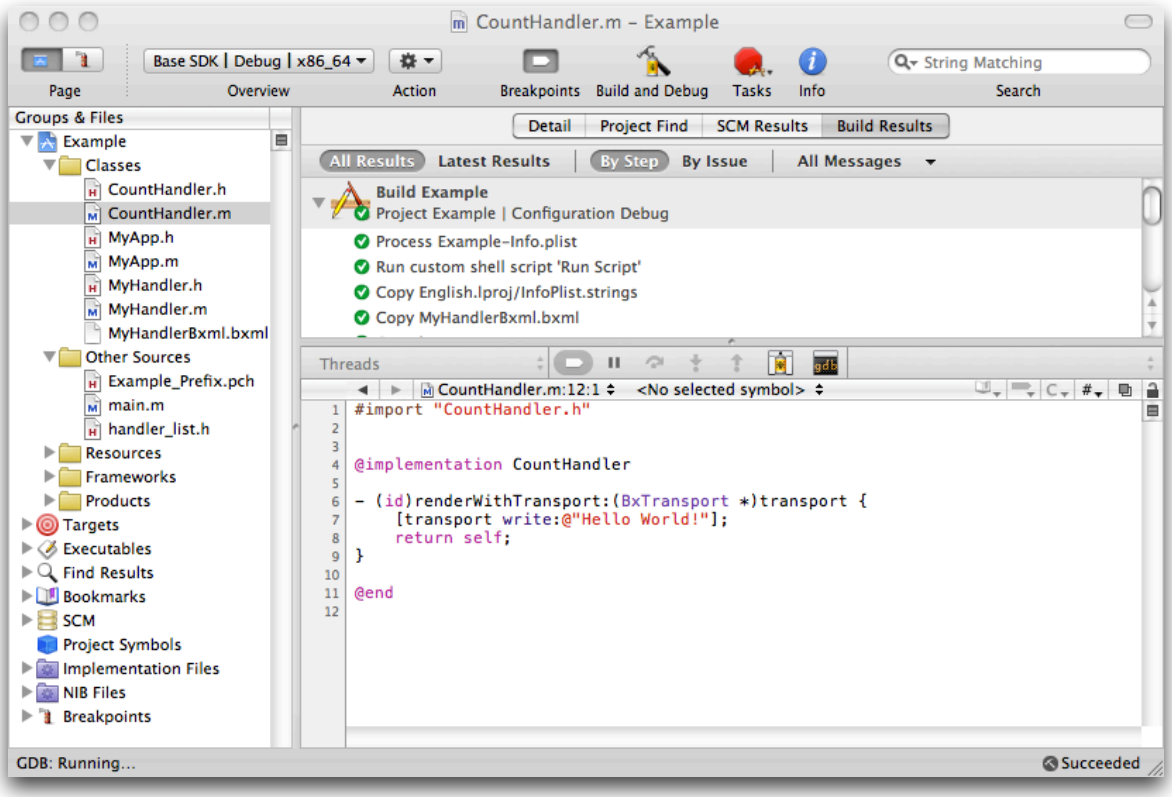
8: Let's add a new handler now by going to the *File* menu and selecting *New File...*, then selecting the *Bombax* category and the *Objective-C Handler* template:



Click *Next* and give the handler the name "CountHandler", then press *Finish*:



You should now have two new files in your project, `CountHandler.h` and `CountHandler.m`, appearing like this:



Now let's add an instance variable named `_counter` to `CountHandler.h`. While:

```
#import <Cocoa/Cocoa.h>
#import <Bombaxtic/BxHandler.h>

@interface CountHandler : BxHandler {
    int _counter;
}

@end
```

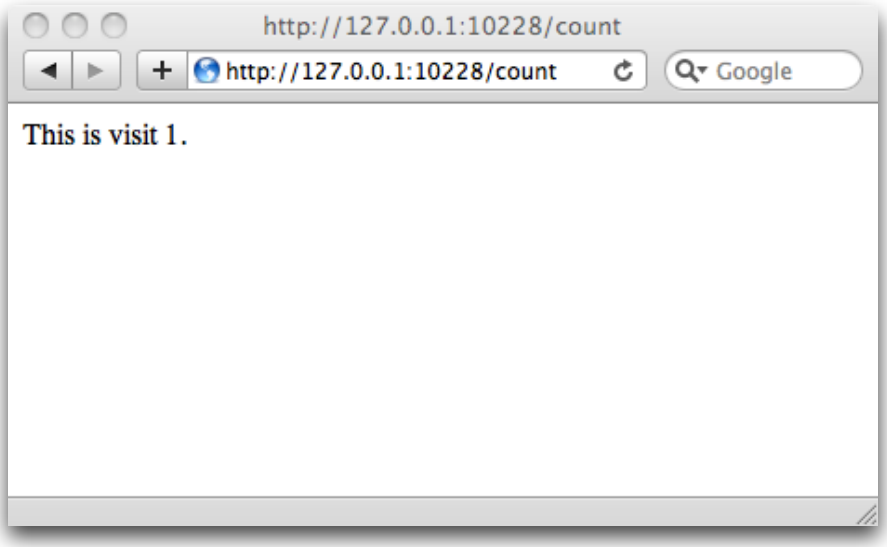
In `CountHandler.m` let's also override the `setup` method to initialize `_counter`. This method is ran once during the lifetime of the BxApp: right before your `BxHandler` subclass first handles a client request. In addition to adding the `setup` method, let's also change `renderWithTransport:` to increment `_counter` on each request and show the result:

```
#import "CountHandler.h"

@implementation CountHandler

- (id)setup {
    _counter = 0;
    return self;
}

- (id)renderWithTransport:(BxTransport *)transport {
    _counter++;
    [transport writeFormat:@"This is visit %d.", _counter];
    return self;
}

@end
```

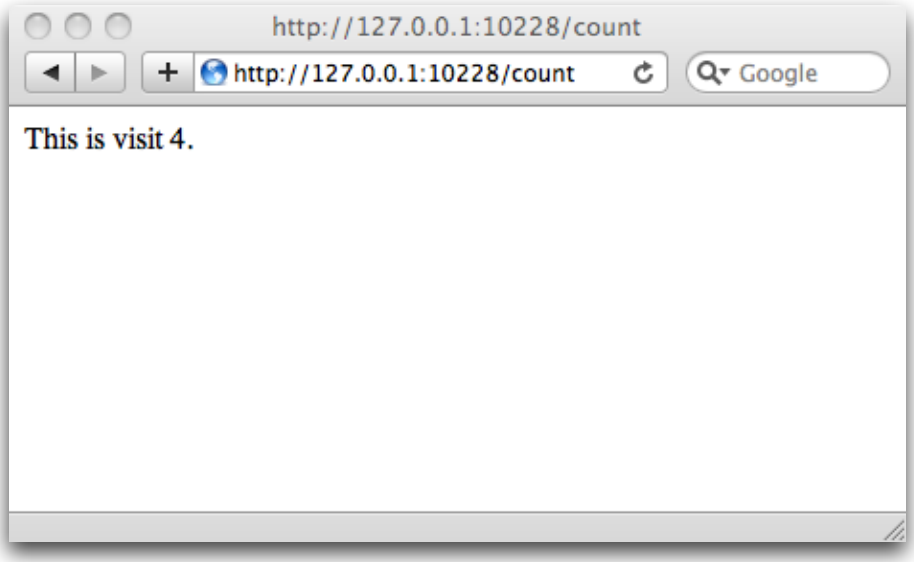
Since we've added a new handler, we need to make sure the BxApp knows how and when to use it for requests so let's modify `MyApp.m` to include a mapping to `CountHandler`:

```
#import "MyApp.h"

@implementation MyApp

- (id)setup {
    [self setHandler:@"CountHandler" forMatch:@"/count"];
    [self setHandler:@"MyHandlerBxml" forMatch:@"/example"];
    [self setDefaultHandler:@"MyHandler"];
    return self;
}

@end
```

Now restart the BxApp and visit it at http://127.0.0.1:10228/count. It should look like this on the first visit:



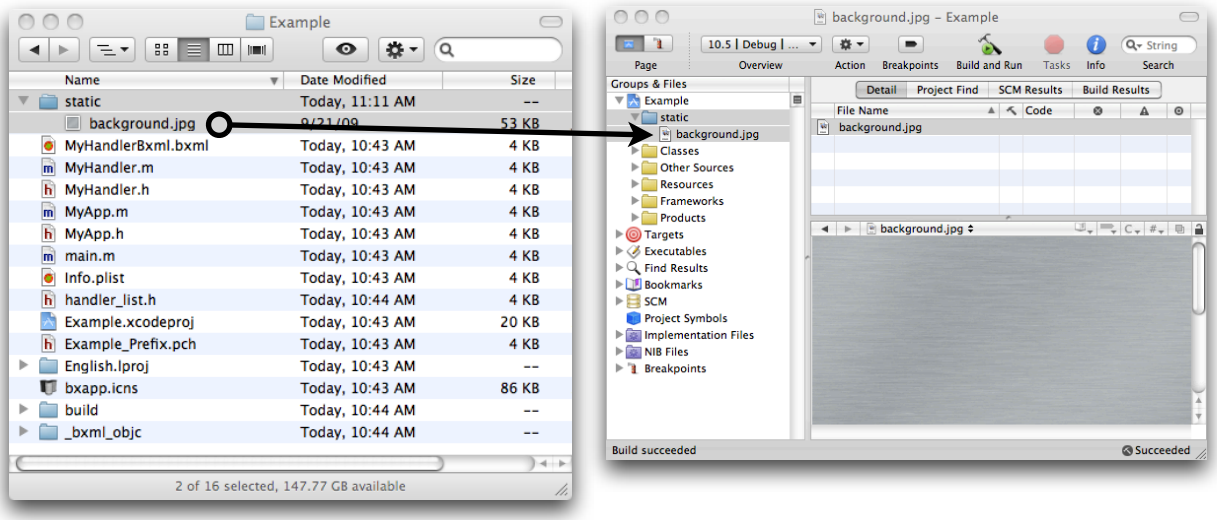
Try visiting it again. Each time you visit it, `_counter` should be incremented and the page should reflect its new value:



Note that many browsers request a `favicon.ico` file when visiting web page. This can lead to spurious counter increments. You can use `BxTransport`'s `requestPath` property to ignore these requests:

```
- (id)renderWithTransport:(BxTransport *)transport {
    if ([transport.requestPath isEqualToString:@"/favicon.ico"]) {
        [transport setHttpStatusCode:404];
    } else {
        _counter++;
        [transport writeFormat:@"This is visit %d.", _counter];
    }
    return self;
}
```

9: As our last step in this tutorial, let's add an image in the special `static` resource folder. This is a folder reference in your BxApp project named `static` that corresponds to an actual folder in your project's folder. Any file that resides in this folder will be automatically included and accessible when your BxApp is built. For example, if we add a file named `background.jpg` it appears in our BxApp project.



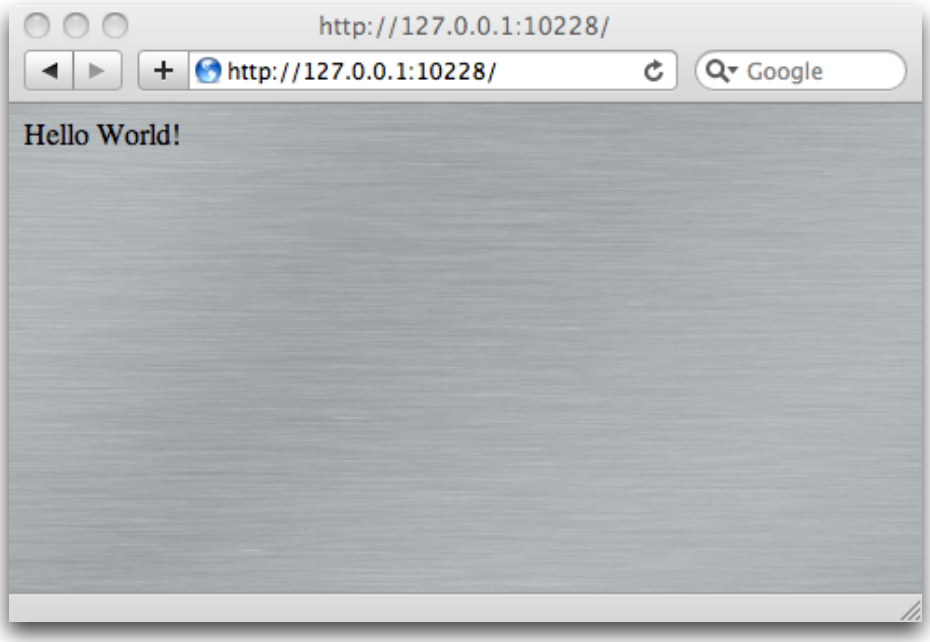
These static files are available in your application by calling `BxApp`'s `staticWebPath:` method. This method returns a web-accessible URL that will resolve to the file specified regardless of where the BxApp is hosted. For example, the following will work whether the BxApp's location is at the root / or a path such as /apps/Example:

```
#import "MyHandler.h"

@implementation MyHandler

- (id)renderWithTransport:(BxTransport *)transport {
    [transport writeFormat:@"<html><body style='background-image:url(%@)'>Hello
World!</body></html>",
     [self.app staticWebPath:@"background.jpg"]];
    return self;
}

@end
```

You can also use the `<?static ?>` tag to have the same effect in your BXML handlers (see Section 3 for more information on using BXML):

```
<html>
 <body style="background-image:url(<?static background.jpg ?>)">
  Hello World!
 </body>
</html>
```

No matter where your BxApp is installed or if you are debugging it, the static resource is located and rapidly served to the client:



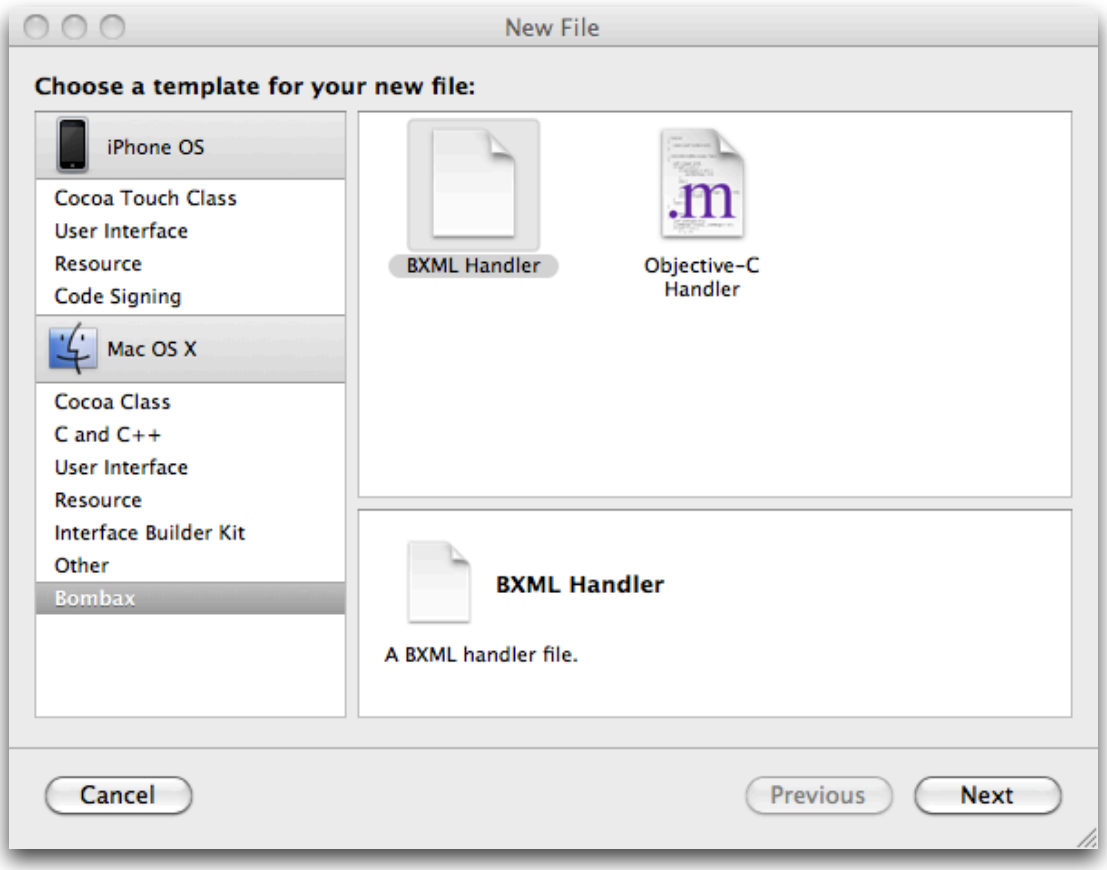
Congratulations! You've just created a BxApp and your own custom handler! The Bombaxtic framework contains many features to help you in the creation of your web application. Please see the *Bombaxtic Framework Reference* for detailed information about the classes included in the Bombaxtic framework. The rest of the guide will introduce you to more advanced topics in developing on the Bombax platform.

3 - BXML Files

As we saw in Section 2, when Bombax receives an incoming client connection and passes it your BxApp, your BxApp uses custom `BxHandler` subclasses to handle the incoming client requests. These `BxHandler`s can be standard Objective-C subclasses comprised by `.h` (header) and `.m` (method) class files. Alternatively, the Bombax platform allows you to write your `BxHandler` as BXML file containing special BXML tags inserted within another file format such as HTML or XML. This work very similarly to other tag-based template systems such as JSP (Java Server Pages) and PHP.

3.1 Creating a BXML Handler

To create a BXML page, you need to add a BXML file to your project. You can do so by going to the *File* menu, clicking on the *New File...* menu item, and then the *BXML Handler* within the *Bombax* category:



This will create a BXML file that operates as `BxHandler` subclass. As when you create an Objective-C `BxHandler`, you'll need to add the classname of the handler to your `BxApp` subclass. Normally, the name you give your BXML handler is also its classname.

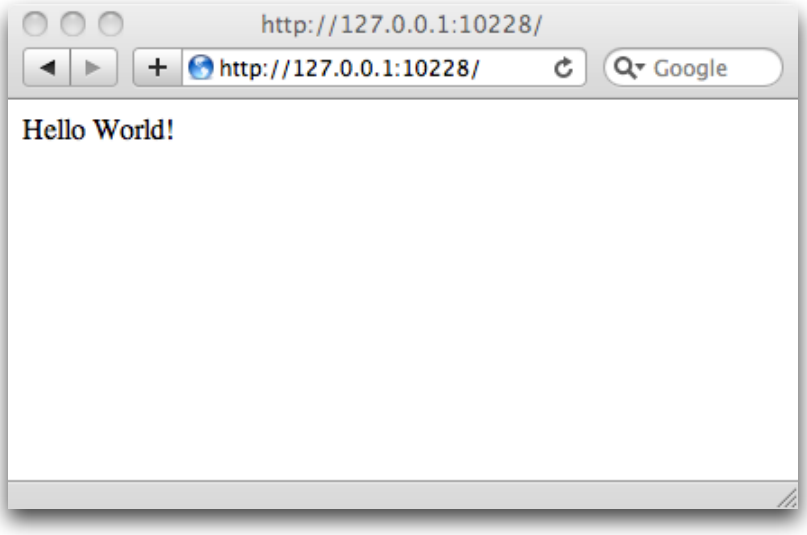
For example, if you create a BXML file named `TestHandler.bxml`, your `BxApp` subclass's `setup` method might look this:

```
#import "MyApp.h"

@implementation MyApp

- (id)setup {
    [self setDefaultHandler:@"TestHandler"];
    return self;
}

@end
```

Let's take a look at the contents of your BXML file when it is first created. It should look something like this:

```
<html>
 <body>
  <?
    [_ write:@"Hello World!"];
  ?>
 </body>
</html>
```

Most of this file is comprised by typical HTML tags, such as `<html>` and `<body>`. Within this HTML code, there is a BXML tag that begins with `<?` and ends with `?>`. Within these brackets, you may insert any Objective-C code that you would like just as if you were writing it within the `renderWithTransport:` method of a `BxHandler` subclass. In fact, the `BxTransport*` variable is available to you with the identifier '`_`' and just as when you wish to output to the client using an Objective-C handler you can use the `write:`, `writeFormat:`, and `writeData:` methods to create your response. Our BXML file currently it writes out "Hello World!" to the client so if we start the debugger and visit our BxApp, the response looks like this:



If we look at the raw HTML source that the web browser renders for this page you can see that when the BXML handler was called, it inserted the `write:` method directly between the surrounding HTML tags:

```
<html>
 <body>
  Hello World!
 </body>
</html>
```

3.2 - How BXML is Processed

When you build a BxApp with BXML files in it, several things happen behind the scenes in order to create a working handler that can respond to client requests. You don't need to do anything special in order to initiate the processing of BXML files beyond building the BxApp as your BxApp project was automatically set up to do so when you first created it.

For each `.bxml` file included in your project, a command-line tool called `bxmlparser` is invoked to convert the BXML into an Objective-C `BxHandler` subclass. `bxmlparser` scans the `.bxml` file for the special BXML tags like `<?` and `<?setup` in order to build up the resulting Objective-C class.

The newly generated Objective-C files are not visible in your project and should not be modified directly as they will be overwritten using the `.bxml` file upon each build. This is important to remember as any compilation errors they will show up in the auto-generated files and Xcode will allow you to make edits to them. Instead, make the changes in the `.bxml` files.

Once `bxmlparser` has finished generating the Objective-C classes, the resulting method files are then added to a special collecting header file header file named `handler_list.h`. This ensures that all of the BXML-generated Objective-C classes are compiled and available when the build takes place.

Although BXML is usually written to insert into HTML, it can also be used with other kinds of text-based file formats, such as XML, plain text, and RDF. In these cases, it is often necessary to specify the `Content-Type` header at the beginning of the file (see Section 3.4.1).

Because BXML is not interpreted at runtime but is converted to a compiled Objective-C class, there is no performance penalty for using BXML.

3.3 - Non-Enclosed Text

Sections of the BXML file that are not within BXML tags are inserted in the order they appear (interwoven with `<? ?>` code blocks) as `write:` calls within the `renderWithTransport:` method. Whitespace character such as newlines are not trimmed.

3.4 - BXML Tags

Several different BXML tags are available for you to use in your handler. All tags except `<?--` (which is closed with `--?>`) are closed with `?>`.

3.4.1 The `<?` Tag

The `<?` tag is used to define a section of Objective-C source code within the `renderWithTransport:` method. It is inserted into `renderWithTransport:` in the order it appears in your BXML file and is interwoven with any non-enclosed text. Because any non-enclosed text will cause a `write:` to be called and all headers to be written out (see the *Bombaxtic Framework Reference*), it is important to write any calls to set headers or cookies at the very beginning of your BXML file. For example this will simply output "This is now a plain text file." with a `Content-Type` of `text/plain`:

```
<? [_ setHeader:@"Content-Type" value:@"text/plain"]; ?>
This is now a <? [_ write:@"plain text"] ?> file.
```

Because `<?` contents are inserted directly as Objective-C code, you can create conditional blocks and loops. For example:

```
<html>
 <body>
  <? if ([_.queryVars objectForKey:@"name"] != nil) { ?>
    Your name is <? [_ write:[_.queryVars objectForKey:@"name"]]; ?>
  <? } else { ?>
    What's your name again?
  <? } ?>
 </body>
</html>
```

3.4.2 The `<?setup` Tag

The `<?setup` tag, which may occur anywhere in the BXML file, inserts its contents into the setup: method of the resulting BxHandler subclass.

3.4.3 The <?import Tag

The <?import tag, why may occur anywhere in the BXML file, contains a single header file for #import usage. For example, <?import Quartz/Quartz.h ?> and <?import MyDatabase.h ?>.

3.4.4 The <?base Tag

Your BXML handler may have another BxHandler subclass as its superclass. Because the generated renderWithTransport: calls [super renderWithTransport:_] as its first operation, this is very useful as you can define methods, properties, and member variables in the parent class which can be accessed in your BXML code. The <?base tag contains a single classname like <?base MySuperClass ?>.

Using the <?base tag can be a good way to template data through BXML with your business logic occurring upstream. The other major technique for this is to have your controller BxHandler call the appropriate view BXML with a call like:

```
[[self.app handlerInstanceForClassName:@"MyBxmlTemplate"] renderWithTransport:_];
```

3.4.5 The <?paste Tag

The <?paste tag calls the renderWithTransport: method of the provided BxHandler classname causing all writes to be inserted at that point in your BXML file. For example, <?paste MyHeader ?>.

3.4.6 The <?static Tag

The <?static tag calls BxApp's staticWebPath: with the provided file name to insert a relocatable URL that is mapped to your BxApp's static resource folder. method of the provided BxHandler classname causing all writes to be inserted at that point in your BXML file. For example:

```
<html>
 <head>
  <script type="text/javascript" src="<?static main.js ?>"></script>
 </head>
 <body>
  <img src="<?static images/header.jpg ?>" />
 </body>
</html>
```

3.4.7 The `<?name` Tag

The `<?name` assigns a specific classname to the Objective-C class generated from your BXML. If you do not use the `<?name` tag, the classname will be the BXML file's name without the `.bxml` extension.

3.4.8 The `<?--` Tag

The `<?--` tag comments out a BXML file until it finds `--?>`. All commented out BXML tags and non-enclosed text is ignored by bxmlparser and not included in the resulting Objective-C file.

4 - Debugging BxApps

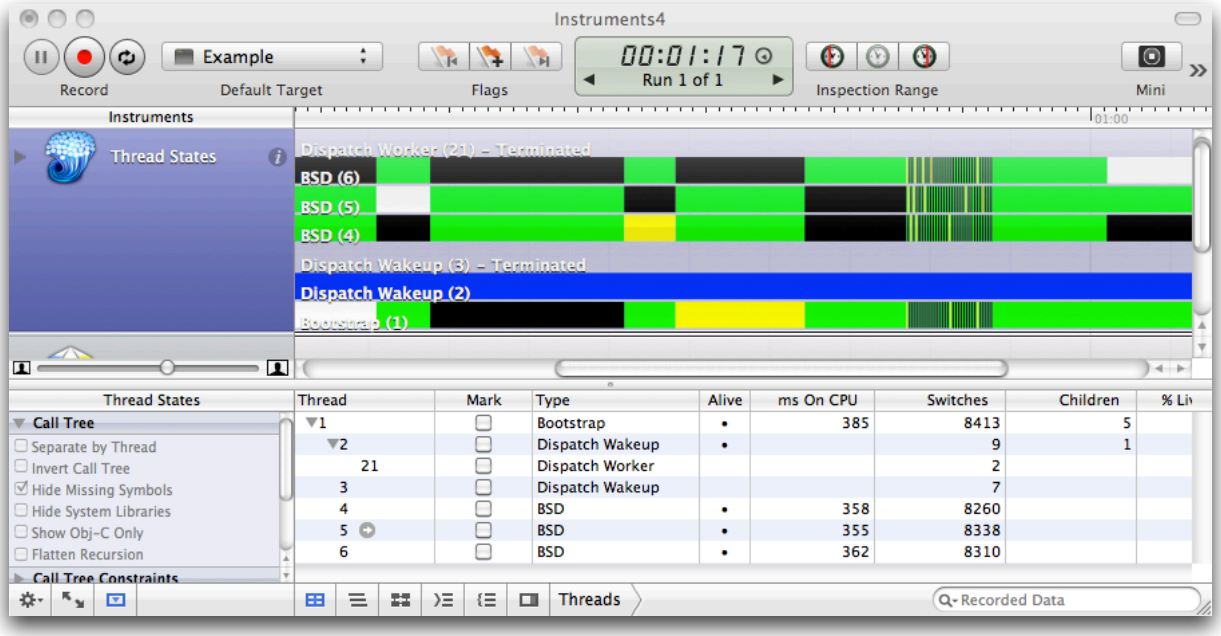
Bombax supports debugging during development through the Xcode IDE and has built-in error recovery features. Debugging web applications is often fairly difficult and can involve so-called 'printf' debugging, peppering code with `echo` and `die` commands that all too often are left in after finding the bug. Bombax changes this dramatically by making debugging a BxApp as easy as debugging a standard Cocoa application.

4.1 - Debugging in Xcode

To debug a BxApp application in Xcode, simply set a breakpoint, start in the debugger, and visit a page that will trigger the breakpoint. You will be able to step through the code line by line, watch expressions, and do other typical Xcode debugging tasks. Normally, only one BxApp may be debugged at a time. For more information about using the debugger in Xcode, please see Apple's *Xcode Debugging Guide*.

4.2 - Profiling and Analyzing Your BxApp

In addition to debugging, BxApps support the use of the profiling and analysis tools included with Xcode. For example, you can diagnosis thread usage through the Instruments application:



For more information, please see Apple's *Instruments User Guide*.

4.3 - Handling Exceptions and Crashes

When the incoming client request is ready to be passed to the `renderWithTransport:` method of your `BxHandler` subclass, it is wrapped within an Objective-C exception handler. If an exception is thrown and not caught in `renderWithTransport:`, this exception handler will invoke `BxApp`'s `exit:` method, causing the BxApp to terminate. Similarly, if a runtime or system error is encountered, such as by dividing by zero, your BxApp will automatically catch the error and call `exit:` on your `BxApp` subclass.

You can override `exit:` to perform any critical operations before the BxApp terminates. Because this method is also called when the Bombax server is requesting BxApps to close, it is imperative that this method returns as quickly as possible. Specifically, If an error triggers the `exit:` method and the Bombax server subsequently requests the BxApp exits while the `exit:` method is still in operation, the BxApp will be terminated without waiting for `exit:` to return. `exit`: is guaranteed to be called only by a single thread, but should be treated as having the same restrictions as a signal handler (see the `man` page for `sigaction` for more information).

If an unexpected error occurs and your BxApp is exited the Bombax server will automatically restart the BxApp if the *Watchdog Restart* checkbox has been checked for the BxApp location in the Bombax server administration application. Note that BxApps being debugged using the debug port are not automatically restarted.

5 - Interfacing

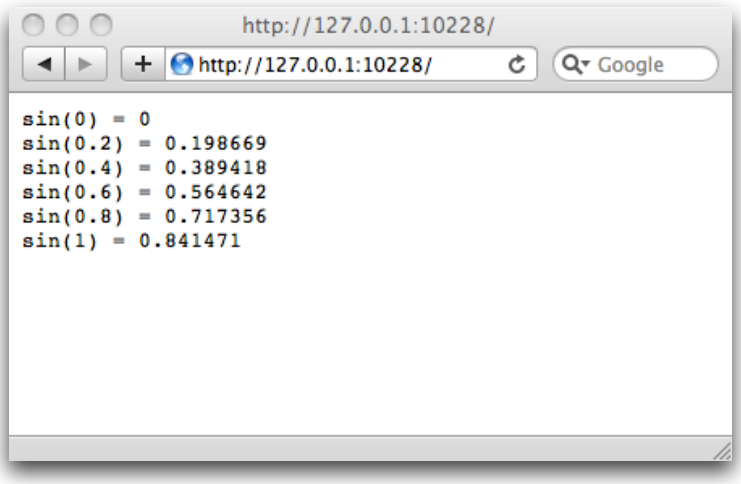
BxApps have the same kind of ability to interface with libraries and other programs that standard Cocoa applications do. Just like with standard Cocoa applications, if you link to a framework or library that is not included with the standard OS X distribution, it is important to ensure that these resources are installed or otherwise available when distributing your BxApp. However, with the large number of excellent frameworks and libraries included with OS X, the vast majority of BxApps already have everything they need available to them.

5.1 - Interfacing with Other Frameworks

To interface with a framework, simply add the framework to your BxApp project to access the framework's headers and libraries. For example, if you have added the Accelerate framework and write a handler to calculate the sine of six doubles like this:

```
#import "MyHandler.h"
#import <Accelerate/Accelerate.h>

@implementation MyHandler

- (id)renderWithTransport:(BxTransport *)transport {
    double x[6] = { 0, 0.2, 0.4, 0.6, 0.8, 1 };
    double y[6];
    int n = 6;
    vvsin(y, x, &n);
    [transport setHeader:@"Content-Type"
                   value:@"text/plain"];
    for (int i = 0; i < 6; i++) {
        [transport writeFormat:@"sin(%g) = %g\n", x[i], y[i]];
    }
    return self;
}

@end
```

The handler will create this output:



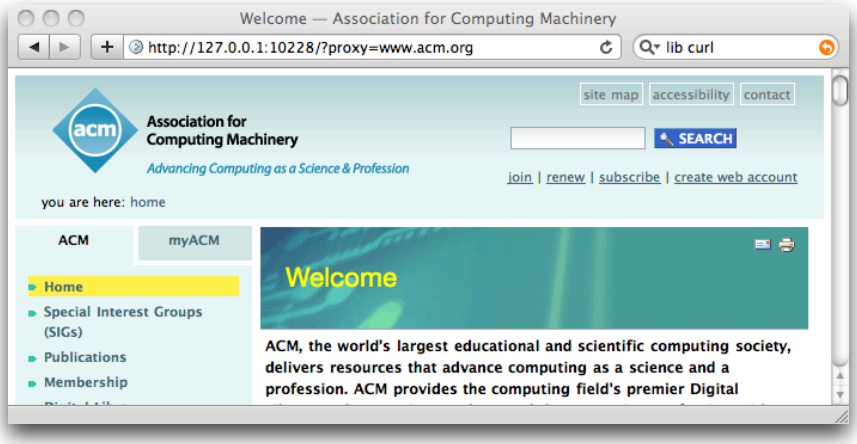
See Apple's *Framework Programming Guide* for more information about including frameworks.

5.2 - Interfacing with C and C++

Just as with standard Objective-C program, you can very easily use any C code either by adding or writing the `.c` or `.h` files directly or by linking to a C library. Note that if you link to a library it is still necessary to include any headers that aren't in the standard include path. The same caveats that apply to frameworks about preinstalling any non-standard libraries apply to dynamic C libraries as well. For example, in this handler `libcurl` is used to fetch the URL passed as the `proxy` query variable.

```
#import "MyHandler.h"
#import <curl/curl.h>

@implementation MyHandler

size_t writeCurlPage(void *ptr, size_t size, size_t nmemb, void *stream) {
    BxTransport *transport = (BxTransport *) stream;
    size_t len = size * nmemb;
    [transport writeData:[NSData dataWithBytes:ptr
                                        length:len]];
    return len;
}

- (id)renderWithTransport:(BxTransport *)transport {
    NSString *proxyUrl = [transport.queryVars objectForKey:@"proxy"];
    if (proxyUrl) {
        CURL *curl = curl_easy_init();
        curl_easy_setopt(curl, CURLOPT_URL, [proxyUrl UTF8String]);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, transport);
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, writeCurlPage);
        curl_easy_perform(curl);
        curl_easy_cleanup(curl);
    }
    return self;
}

@end
```

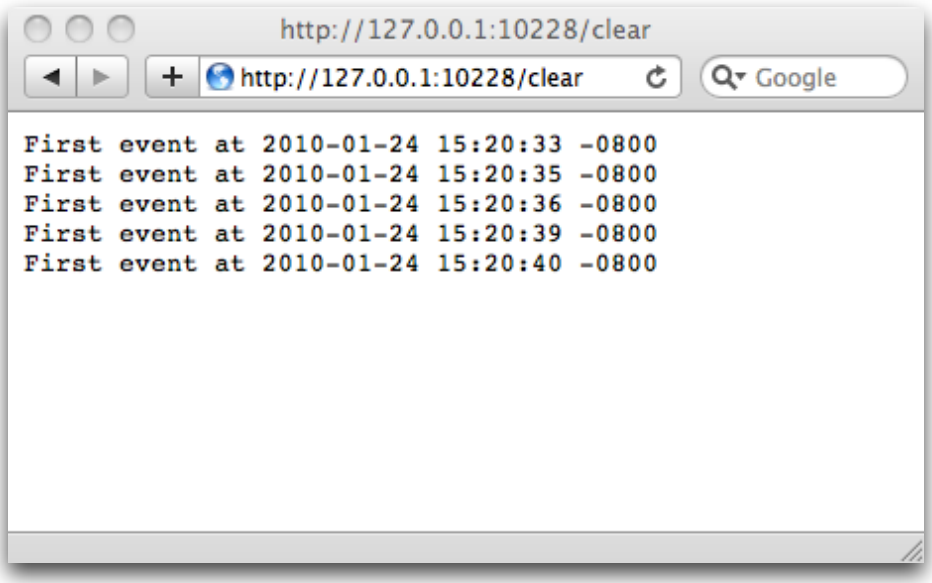
Thus accessing http://127.0.0.1:10228/?proxy=www.acm.org returns a page like this:



Using C++ (i.e. Objective-C++) is also possible in your BxApp. Simply rename your handler implementation's extension from `.m` to `.mm`. You will then be able to access and write C++ in your handler (C++ in BXML files is not supported; use a backing or helper Objective-C++ class in that case). For example, this handler uses the `queue` STL class to create a FIFO storage of `NSTimeInterval`s until /clear is accessed:

```
#import "MyHandler.h"
#import <queue>

@implementation MyHandler

static std::queue<NSTimeInterval> _timeQueue;

- (id)renderWithTransport:(BxTransport *)transport {
    [transport setHeader:@"Content-Type"
                   value:@"text/plain"];
    if ([transport.requestPath isEqualToString:@"/"]) {
        _timeQueue.push([NSDate timeIntervalSinceReferenceDate]);
        [transport write:@"Stored date"];
    } else if ([transport.requestPath isEqualToString:@"/clear"]) {
        while (! _timeQueue.empty()) {
            NSDate *date = [NSDate
dateWithTimeIntervalSinceReferenceDate:_timeQueue.front()];
            [transport writeFormat:@"First event at %@\n", date];
            _timeQueue.pop();
        }
    }
    return self;
}

@end
```

After several visits to http://127.0.0.1:10228/, accessing http://127.0.0.1:10228/clear might return output like this:



For more information about using Objective-C++, please see the relevant chapter in Apple's *The Objective-C Programming Language* book.

5.3 - Interfacing with Other Languages

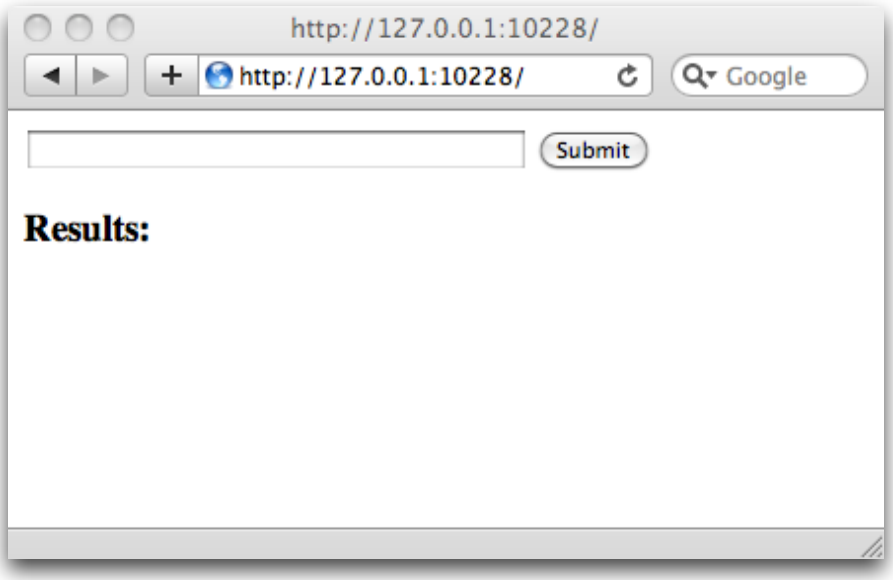
Bombax has access to a large range of Inter-Process Communication (IPC) mechanisms to communicate with other programs, such as Erlang through sockets and scripts through `NSPipe`. In addition to these common techniques, Bombax has a special characteristic for interacting with other languages by virtue of being 100% C-compatible.

Most high level languages are built upon lower-level languages, especially C, to minimize the performance and memory inefficiencies their abstractions create. As a result, many languages are available through instantiating them internally in the BxApp. Although this is a fairly advanced technique, it can be useful when using a speciality language or when porting legacy code.

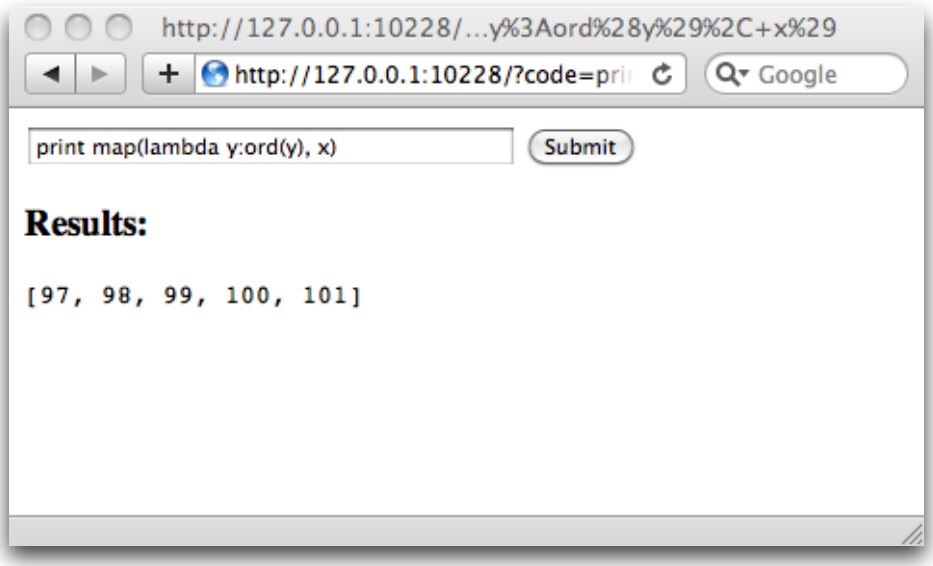
For example, if you link to the `libpython` dynamic library, the following BXML handler will create a persistent Python session:

```
<?import Python/Python.h ?>
<?setup
Py_Initialize();
NSString *outputPath = [NSTemporaryDirectory()
stringByAppendingPathComponent:@"py_output"];
[self.state setObject:outputPath
                forKey:@"output"];
freopen([outputPath UTF8String], "w", stdout);
?>
<html>
 <body>
  <form>
   <input size="35" type="text" name="code" />
   <input type="submit" />
  </form>
  <h3>Results:</h3>
  <pre>
<?
NSString *code = [_.queryVars objectForKey:@"code"];
if (code) {
    PyRun_SimpleString([code UTF8String]);
    fflush(stdout);
}
NSString *outputPath = [self.state objectForKey:@"output"];
[_ write:[NSString stringWithContentsOfFile:outputPath
                                   encoding:NSUTF8StringEncoding
                                      error:nil]];
?>
  </pre>
 </body>
</html>
```

This handler creates an internal Python environment and maps the `stdout` stream used for Python output to a temporary file that is written out to the client. The resulting web page initially looks like this:



Try entering a line into the text input such as `x='abcde'` and press the *Submit* button. As this command doesn't output anything to `stdout`, no result is shown but the variable `x` is now bound in the persistent Python session. Now try entering `print map (lambda y:ord(y), x)` and pressing the *Submit* button. This command prints out the ordinal character number of each character in variable `x`. The result should look like this:



Between the C, C++, and Objective-C libraries and frameworks available to you in your BxApp, you have access to more pre-existing functionality than in **any** other web development platform and it's very easy to leverage 3rd party code often with spectacular results. With Bombax the development power traditionally associated with advanced desktop and system programs is now yours when developing web applications.

6 - Persisting Data

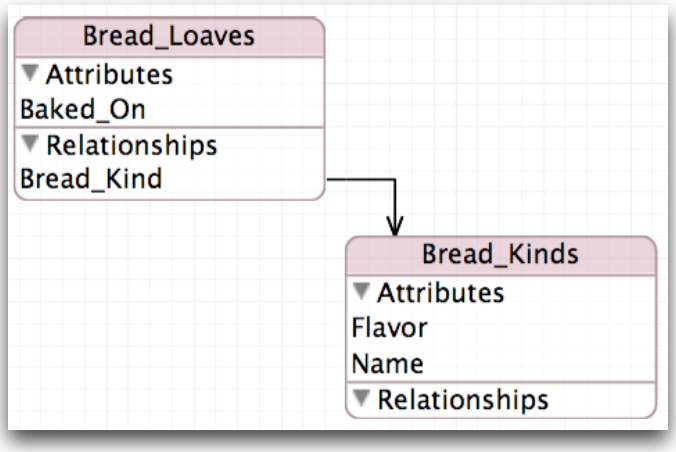
A variety of options are available for persisting data between client requests and BxApp processes. At the simplest level, you can use instance and static variables within your `BxHandler` subclass. As a single handler instance per class will be shared throughout your BxApp, any instance state is also accessible. Note for any persistent mechanism that multiple threads may be rendering in the handler at the same time. Additionally, while only one process is typically running for a BxApp, the Bombax server GUI does allow for load balancing between multiple processes for a single BxApp. Not all applications need to handle this eventuality, but if yours does it may affect which persistence strategy you use.

6.1 - The *state* Property

In the `BxTransport`, `BxHandler`, and `BxApp` instances available during rendering you have access to `NSMutableDictionary` designed to hold state through a request, handler, or application scope respectively. This dictionary is held in the `state` property for each of these classes' instances. The API documentation gives example of their use, and they are designed to provide simple persistence for generic use throughout the application. Note that because a dictionary may hold a dictionary as a value, `state` can be useful for session handling when mapped by a cookie, IP address, or (most secure) a tuple formed from the two.

6.2 - Core Data

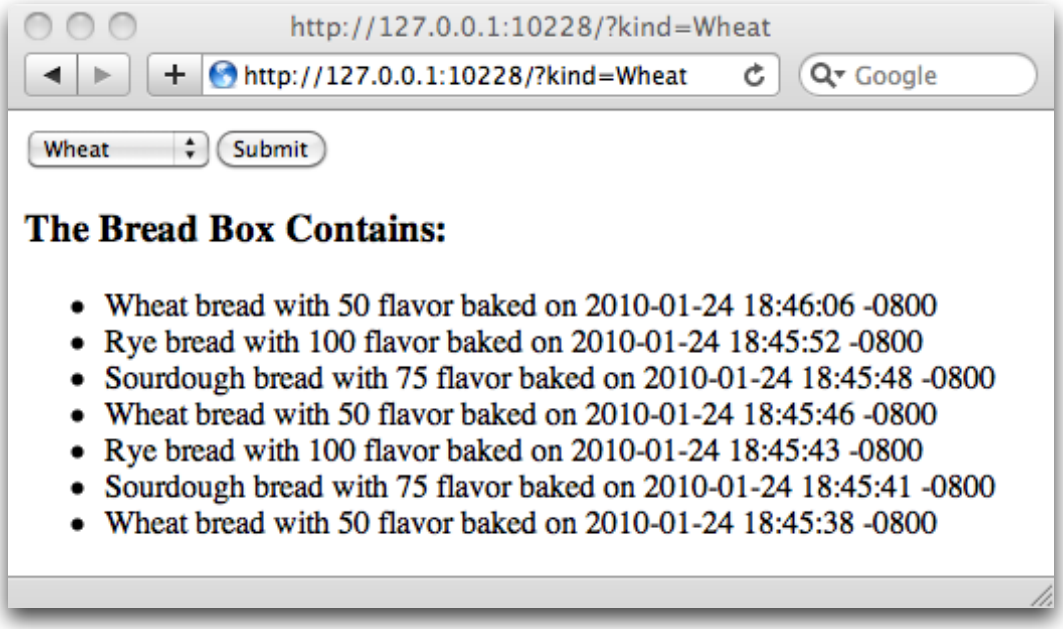
You can use Core Data in your BxApp as long as you are only supporting a single process running at a time. To use Core Data, you need to add the Core Data framework to your project and then create any data models, etc for your application. For example, using the following data model:



For our example, our handler will create a new store on setup, add three kinds of bread, and present a simple interface for adding and fetching breads:

```
#import "MyHandler.h"
#import <CoreData/CoreData.h>

@implementation MyHandler

static NSManagedObjectContext *_moc;
static NSEntityDescription *_loafEntity;
static NSFetchRequest *_allLoavesFetch;
static NSManagedObject *_wheatKind;
static NSManagedObject *_sourdoughKind;
static NSManagedObject *_ryeKind;

- (id)setup {
    _moc = [[NSManagedObjectContext alloc] init];
    NSString *path = [[NSBundle mainBundle] pathForResource:@"Breads" ofType:@"mom"];
    NSURL *url = [NSURL fileURLWithPath:path];
    NSManagedObjectModel *model = [[NSManagedObjectModel alloc] initWithContentsOfURL:url];
    NSPersistentStoreCoordinator *coordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:model];
    [_moc setPersistentStoreCoordinator:coordinator];
    path = [NSHomeDirectory() stringByAppendingPathComponent:@"breads.xml"];
    [[NSFileManager defaultManager] removeItemAtPath:path error:nil];
    url = [NSURL fileURLWithPath:path];
    [coordinator addPersistentStoreWithType:NSXMLStoreType
                              configuration:nil
                                        URL:url
                                    options:nil
                                      error:nil];
    _loafEntity = [[NSEntityDescription entityForName:@"Bread_Loaves"
                            inManagedObjectContext:_moc] retain];
    _allLoavesFetch = [[NSFetchRequest alloc] init];
    [_allLoavesFetch setEntity:_loafEntity];
    NSSortDescriptor *descriptor = [[NSSortDescriptor alloc] initWithKey:@"Baked_On"
                                                               ascending:NO];
    [_allLoavesFetch setSortDescriptors:[NSArray arrayWithObject:descriptor]];
    _wheatKind = [[NSEntityDescription insertNewObjectForEntityForName:@"Bread_Kinds"
                                                inManagedObjectContext:_moc] retain];
    [_wheatKind setValue:@"Wheat" forKey:@"name"];
    [_wheatKind setValue:[NSNumber numberWithInt:50] forKey:@"flavor"];
    _sourdoughKind = [[NSEntityDescription insertNewObjectForEntityForName:@"Bread_Kinds"
                                                    inManagedObjectContext:_moc] retain];
    [_sourdoughKind setValue:@"Sourdough" forKey:@"name"];
    [_sourdoughKind setValue:[NSNumber numberWithInt:75] forKey:@"flavor"];
    _ryeKind = [[NSEntityDescription insertNewObjectForEntityForName:@"Bread_Kinds"
                                            inManagedObjectContext:_moc] retain];
    [_ryeKind setValue:@"Rye" forKey:@"name"];
    [_ryeKind setValue:[NSNumber numberWithInt:100] forKey:@"flavor"];
    [_moc save:nil];
    return self;
}

- (id)renderWithTransport:(BxTransport *)transport {
    NSString *kind = [transport.queryVars objectForKey:@"kind"];
    if (kind != nil) {
        NSManagedObject *loaf = [NSEntityDescription insertNewObjectForEntityForName:@"Bread_Loaves"
                                                               inManagedObjectContext:_moc];
        [loaf setValue:[NSDate date] forKey:@"Baked_On"];
        if ([kind isEqualToString:@"Wheat"]) {
            [loaf setValue:_wheatKind forKey:@"Bread_Kind"];
        } else if ([kind isEqualToString:@"Sourdough"]) {
            [loaf setValue:_sourdoughKind forKey:@"Bread_Kind"];
        } else {
            [loaf setValue:_ryeKind forKey:@"Bread_Kind"];
        }
        [_moc save:nil];
    }
    [transport write:@"<html><body><form><select name='kind'>"];
    [transport write:@"<option>Wheat</option><option>Sourdough</option><option>Rye</option>"];
    [transport write:@"</select><input type='submit' /></form><h3>The Bread Box Contains:</h3><ul>"];
    NSArray *loaves = [_moc executeFetchRequest:_allLoavesFetch error:nil];
    for (NSManagedObject *loaf in loaves) {
        [transport writeFormat:@"<li>%@ bread with %@ flavor baked on %@</li>",
         [loaf valueForKeyPath:@"Bread_Kind.Name"],
         [loaf valueForKeyPath:@"Bread_Kind.Flavor"],
         [loaf valueForKey:@"Baked_On"]];
    }
    [transport write:@"</ul></body></html>"];
    return self;
}
@end
```

Note that your BxApp will need to create the managed object context and persistent store manually, and call save: on the context (e.g. in your `BxApp` subclass's `exit:` method). The resulting web application renders like this:



Please see Apple's *Core Data Programming Guide* for more information.

6.3 - Using Databases

The Bombaxtic framework includes extensive functionality for accessing relational databases through the `BxDatabaseConnection` and `BxDatabaseStatement` classes. Support is provided for Oracle, PostgreSQL, MySQL, and SQLite. Notably, all Bombax servers have built-in support for these databases as well so you do not need to worry about whether drivers are available on the host. Please see the API documentation for details and examples using Bombaxtic's database classes.

6.4 - Configurators and *NSUserDefaults*

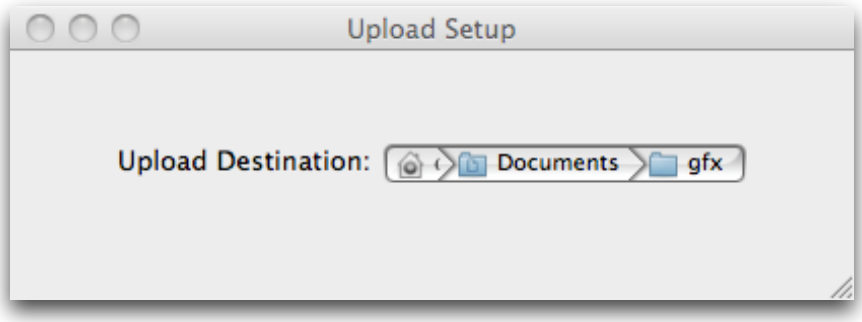
`BxApp` includes a special method named `launchConfigurator` that may be overridden to customize what happens when the *Configure...* button is pressed in the Bombax server application. When the *Configure...* button is pressed, a new process for your BxApp is created which simply calls `launchConfigurator` and then exits. You can customize `launchConfigurator` to load a UI `xib` or `nib` file and operate like a standard desktop application.

For example, we can create a simple window file named `ConfigWindow.xib`, set our `BxApp` subclass as its owner, and define the subclass like this:

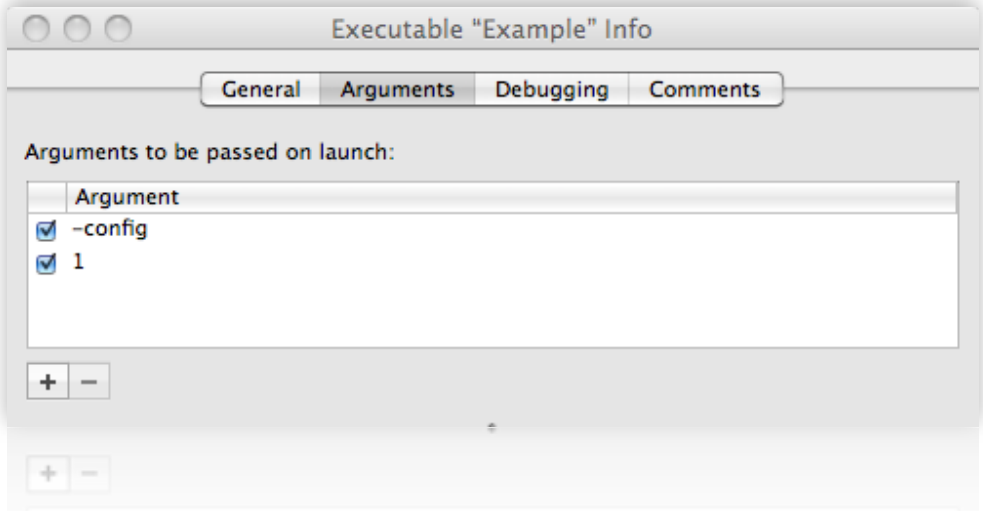
```
#import <Cocoa/Cocoa.h>
#import <Bombaxtic/Bombaxtic.h>

@interface MyApp : BxApp {
    IBOutlet NSWindow *configWindow;
    IBOutlet NSPathControl *uploadPathControl;
}

- (IBAction)setUploadPath:(id)sender;

@end

@implementation MyApp

- (id)setup {
    [self setDefaultHandler:@"MyHandler"];
    return self;
}

- (id)launchConfigurator {
    NSApplication *app = [NSApplication sharedApplication];
    [NSBundle loadNibNamed:@"ConfigWindow" owner:self];
    [[NSNotificationCenter defaultCenter] addObserver:app
                                             selector:@selector(terminate:)

name:NSWindowWillCloseNotification
                                               object:configWindow];
    NSString *path = [[NSUserDefaults standardUserDefaults]
objectForKey:@"uploadPath"];
    if (path != nil) {
        [uploadPathControl setURL:[NSURL fileURLWithPath:path]];
    }
    [app run];
    return self;
}

- (IBAction)setUploadPath:(id)sender {
    NSOpenPanel *panel = [NSOpenPanel openPanel];
    [panel setCanChooseFiles:NO];
    [panel setCanChooseDirectories:YES];
    NSString *oldPath = [[uploadPathControl URL] path];
    if ([panel runModalForDirectory:oldPath
                               file:nil] == NSFileHandlingPanelOKButton) {
        NSURL *url = [panel URL];
        [uploadPathControl setURL:url];
        [[NSUserDefaults standardUserDefaults] setObject:[url path]
                                                  forKey:@"uploadPath"];
    }
}

@end
```

With the `xib` file properly connected to the `MyApp` instance, this application will allow for setting and maintaining the configuration through multiple instantiations of your BxApp. By using `NSUserDefaults` you have a simple configuration persistence mechanism available to your BxApp in a way that can support advanced setup GUIs. This is much more secure and versatile than traditional web or file based configuration.

Depending on how you designed your `xib` file, the end result when you configure your application may look like this:



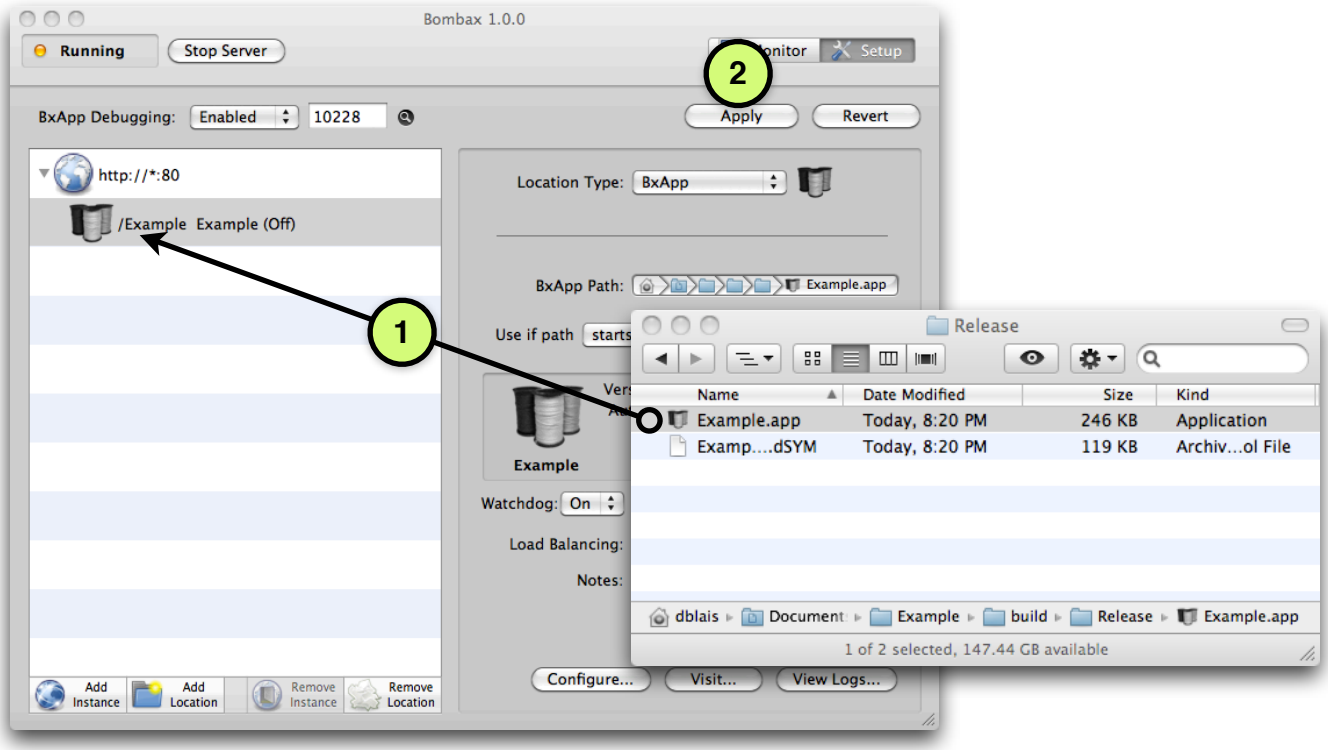
In order to test your configurator during debug, it is necessary to add two arguments (specifically, `-config 1`) to the executable. This is done by editing the executable in Xcode as shown here:



Note that, unlike in its normal mode of operation, your BxApp can only have your configurator open as a single application; multiple presses of *Configure...* in the Bombax server application are ignored. This behavior is identical to trying to open e.g. `TextEdit.app` when it is already open.

7 - Packaging and Running your BxApp

When your BxApp is finished and ready for deployment in a Bombax server, simply perform a release build and locate the resulting `.app` package. This package may be added to the Bombax server by selecting *Add Location*, setting the *Location Type* to `BxApp` and the *BxApp Path* to the location of the `.app` package. Alternatively, you may simply drag and drop the package onto the outline view on the righthand side. Press *Apply* or *Start Server* and your BxApp is immediately available in the location.



7.1 - Info.plist Properties

A nice finishing touch on your BxApp is customization of the properties that show up when the BxApp is installed. These are also shown through the default configurator. You can use whatever values you'd like in your BxApp. Here is an example:

Key	Value
Information Property List	(19 items)
BxApp Name	${PRODUCT_NAME}
BxApp Description	This is an example of what can be done in a BxApp
BxApp Author	Bombaxtic LLC
BxApp Author URL	http://www.bombaxtic.com

7.2 - The Next Steps

This guide should have given you a good overview of the core capabilities of BxApps. While you are developing your BxApps, we recommend that you check out the API documentation available in Xcode. Simply search for a class or choose the Bombaxtic docset to browse the full class list. If you have any questions that the documentation doesn't address or want to share your tips and tricks for using Bombax, please visit the Bombaxtic LLC support page at http://www.bombaxtic.com/support.

Thank you for using Bombax for your web application development!