Developing a Web Server Platform with SAPI Support for AJAX RPC using JSON

Iulian ILIE-NEMEDI, Bucharest, Romania, inemedi@ie.ase.ro

Writing a custom web server with SAPI support is a useful task which helps students and future system architects to understand the link between network programming, object oriented programming, enterprise application designing patterns and development best practices because it offers a vision upon inter-process communication and application extensibility in a distributed environment.
Keywords: Web, Server, Proxy, SAPI, HTTP, RPC, AJAX, JSON, XML.

In nowadays, writing a custom web server from scratches seams like a nonsense task. So, why implementing your own web server when there are a lot of options to choose from, like Apache or IIS. If we need a free application server then we can also use Tomcat, JBoss, JOnAs, Jetty, and so on. Therefore, first we have to explain the purpose of this article. Implementing a web server platform with SAPI support is a nontrivial task, which involves knowledge from different areas like network programming, object oriented programming, enterprise application designing patterns, web development and more. This task is nevertheless a challenge which will provide useful experience for understanding and teaching network concepts, like protocols, services, client-server or request-response patterns and it can be used as an example, like a guide of writing web servers from scratches. Our implementation can be used as well as a lite server platform for web development, since it will support the latest technologies in remote procedure calling combined with AJAX. In fact, it started as an alternative to a more complex ASP.NET, JSP, or even PHP developing environment, which requires technical and administrative resources, like special configured servers for project publishing. Therefore, developing a small but extensible web server platform looks like a very useful challenge which will combine networking theory with development best practices. Now that we know how it stands, let's focus on our task. A web server is an application that listens for HTTP requests and delivers requested resources to application clients through a process which involves both file streaming and custom module execution. This can be done by opening a socket, binding to a network interface and starting a new thread which accepts client connections, redirect them to dedicated external sockets, extract HTTP request from each client socket, compose a HTTP response based on client demand and send it back to the client. Before listening for incoming requests, we'll configure our server by reading an XML configuration file using C# declarative serialization support. We define a set of configuration classes, one for each type of tag found in the configuration file, which will be used by an XmlSerializer to deserialize the text file. Our server will listen on localhost if Listen is set to Internal, or on the first available network interface if External. We can also configure its HTTP port, identification name and connection pending queue. To publish a directory we have to specify an alias for its virtual path and a physical mapping directory. Any request will be mapped using these settings by a MapPath function, which will search for path matching into virtual directories collection. Our server looks for additional default documents when a directory alias is requested, which can be defined by Defaults option. As required by HTTP protocol, we provide MIME mapping for any extension delivered by our server. Modules section maps custom module to its extension. A custom module is a dynamic library loaded from a .NET assembly which has a type that implements IModule interface to handle an extension-based request. Our server implements the following methods and properties:

```
internal class Server : IServer{
        public Server(string configFile){…}
        private void LoadConfiguration(string configFile){…}
        public string MapPath(string path){…}
        private string MapPath(string path, bool useDefaultDocument){…}
        private void Send(Socket socket, SAPI.Response response){…}
        public void WriteLog(string message, params object[] arguments){…}
        public SAPI.IModuleConfiguration ModuleConfiguration{get{…}}
        private void Run(){…}
        private SAPI.IModule LoadModule(Configuration.Module module){…}
        private void Process(Socket socket){…}
        private byte[] ReadFile(string path){…}
        private string GetStatusCodeDescription(HttpStatusCode statusCode){…}}
```

While listening for incoming requests, our server splits its workflow into simple tasks, like accept client connections and process accepted socket. Any exception will be formatted with timestamp and additional information and than logged into a file. When processing a request, first we deserialize client call into a Request object, which extracts HTTP headers and additional body data. This will hold Query string and Form arguments as well. Than we instantiate a Response object and use it to store output data, which will be sent to the client. Since this is a demonstration, we won't use direct streaming to the client socket, but write intro a memory stream instead. If an exception occurs while trying to process the client request, then we set the response status code to 500 (Internal Server Error) and write to response output a HTML file explaining that an internal server error has occurred. Our server answers only to GET and POST verbs from HTTP. It looks into module list to find the appropriate module used to process request by its extension. If none was found, then if requested path could not be mapped to an existing file, or default directory document, we set status code to 404 (Not Found) and output a HTML file to the client explaining that the requested resource couldn't be found. If requested resource is mapped to an existing file, then we set the content type of the HTTP response by searching the MIME for file extension into MIME configuration, and than we read file content into response output, setting the Status Cod to 200 (OK). Since response is completed, we send it to the client using a Send method, than log action and close client socket. In this way, our server delivers static resources stored into file system. If the request targets an extension handled by a custom module, then we load its assembly using reflection and invoke the Process method implementation from module type, passing reference to this server, and to the current request and response objects. Both Request and Response classes implements their corresponding interface from SAPI. A SAPI (Server Application Programming Interface) provides extensibility support to implement and host custom modules into an application server which will be used to process custom extensions. In a straightforward approach we'll define an IServer interface providing a mapping method called to map a virtual path to a physical file or directory, a logging method by which a module can log its errors and exceptions into server log file, and also details regarding module configuration like its extension, so that any request will be able to extract its virtual path without extension:

```
public interface IServer{
        SAPI.IModuleConfiguration ModuleConfiguration{get;}
        string MapPath(string path);
        void WriteLog(string message,
        params object[] arguments);}
```

Further more, we define an IRequest interface used to access information extracted from HTTP request by the server, like URL, method, GET Query String, POST Strings, headers or client Remote EndPoint:

```
public interface IRequest{
        NameValueCollection Query{get;}
        NameValueCollection Form{get;}
        string Url{get;}
        string Method{get;}
```

string Version{get;}
NameValueCollection Headers{get;}
string RemoteEndPoint{get;}}

We need to specify also an IResponse interface which will be used by a given module to output its processing result, having methods to write data to the output stream, eventually with formatting support, and also access to HTTP response headers and status code:

public interface IResponse{
void Write(string format, params object[] arguments);
void WriteLine(string format, params object[] arguments);
void Write(byte[] data, int offset, int count);
void Write(byte[] data);
Stream Stream{get;}
HttpStatusCode StatusCode{get;set;}
string ContentType{get;}
NameValueCollection Headers{get;}}

Since in the IServer interface we get access to module configuration, this can be done by having another IModuleConfiguration interface which will be implemented by the server through a ModuleConfiguration class, which is used to deserialize data from a configuration file:

public interface IModuleConfiguration{string Extension{get;}}

The IModule interface contains only one method, Process, which is called to process requests handled by the current module:

public interface IModule{void Process(IServer server, IRequest request, IResponse response);}

Note that the Process method is the core extension point by which our server is more resilient than a simple layered web server because it can plug-in future extensions by that entry point. Since it receives references to server, request and response, the Process method is able to intercept HTTP communication and build its own output to the client, based on demand. To illustrate the SAPI interface, we can implement a simple module, which handles, for instance, .smp extension by outputting variables received from a HTML form. All we have to do is testing the request method and iterate through Query or Form collection to build a HTML list output. Than we can register our simple module into Modules section and test a HTML form that sends data to a fictive .smp file on our server. A more complex example can configure and run SQL queries against a relational data store and output a XML serialization of results, delivered by a query builder module for .xsql extension. Since we have SAPI support, we can extend our server basic functionality to execute remote procedure calls (RPC) that will handle JavaScript client requests made using AJAX. AJAX (Asynchronous JavaScript and XML) is a new approach in building dynamic web pages which enables a web page to make client request to server scripts or applications using XMLHttpRequest or ActiveXObjects supported by browsers. The typical scenario for which AJAX was intended is a web application that manages data displayed and editable into a large client grid. In order to save any change, a client must submit data to the server, which will reload the entire grid, even though only one record has been changed. To avoid transport overhead, as well as controls state management issues, AJAX makes HTTP request directly from client JavaScript, send changes to server-side scripts, receives new data and displays it using the DOM model for HTML tags. At the beginning, AJAX used XML for object serialization and deserialization, which proved to be inefficient in this context. Although XML is widely spread through object serializers of application servers, it is not suitable for client deserialization, since it requires additional DOM parsers. We can use JSON (JavaScript Object Notation) instead, because a JSON serialized object can be extracted only by calling the eval function. We have to implement a JSON parser and builder which will receive strings from requests and extract C# (Java, or PHP) objects, and in the same time will be able to serialize a given object into a JSON string. Although this task is a little bit trivial in PHP, it is difficult in C# or Java, since we have to produce a strongly time object from a naked-type string. Fourth generation languages like

C# uses runtime XSD schema to deserialize object from naked-type XML, which is not easy to handle from client JavaScript. Event though server can send schema to the client, adding information to it is a complicated issue on the client. Therefore, we must implement JSON deserialization in C# or Java without schema, which is impossible unless we know it from somewhere else, like method signatures. We'll explain in this article how to implement in C# a remote procedure caller using JSON serialization. The client will send call request using an XMLHttpRequest or an ActiveXObject. Each call will target a method of a given type from an assembly published on the server. So, the client will serialize into JSON a post argument with these information and will expect a response containing method result or an exception if this is the case. First, we create a JSON serializer, which will serialize into JSON any C# object by extracting its sub-objects from public fields and properties:

```
internal static string Serialize(object obj){…}
private static string Serialize(FieldInfo field, object obj){…}
private static string Serialize(PropertyInfo property, object obj){…}
```

We can do that by loading the assembly file, searching the given type by its name and get all public fields and properties using reflection. Than we have to implement two methods, one for object deserialization and the other for array deserialization, since they are represented in different ways:

```
internal static object DeserializeObject(string data, Type type){…}
internal static object DeserializeArray(string data, Type type){…}
```

In order to do this, we have to write other two helper methods, which split an object or an array into members:

```
internal static NameValueCollection SplitObject(string data){…}
internal static string[] SplitArray(string data){…}
```

Now we are able to extract client arguments and compose server response. All we have to do is to implement an RcpModule like this:

```
public class RpcModule : SAPI.Imodule{
    public void Process(SAPI.IServer server, SAPI.IRequest request, SAPI.IResponse response){…}
    private static void ProcessCall(SAPI.IServer server, SAPI.IRequest request, SAPI.IResponse response){…}
    private static void ProcessSearch(SAPI.IServer server, SAPI.IRequest request, SAPI.IResponse response){…}}
```

In the Process method we search into request object for the call argument given by the client, and execute ProcessCall if found it. Otherwise, we can search for callable assemblies, types and methods into given virtual path of the client request and display a self-description HTML page. The ProcessCall method loads the requested assembly, extract call type and execute call method by its name with the given arguments deserialized using our JSON serializer. If any exception occurs, then it will be serialized into result exception property, as well as reported into log file by calling WriteLog method of the server object given as argument. Method's result will be sent as result value and serialized as JSON string to client. The JavaScript client will have a helper proxy and a JSON serializer; there is no need for a JSON deserializer since deserialization is done simply by calling eval function with the result string received from the server. We implement the serializer using singleton pattern, which allows us to call it directly without having to instantiate it, remember that JavaScript is only object-based language and not object oriented so it doesn't allow static methods on classes. The proxy will create an asynchronous HTTP request containing posting data with call arguments and will send it to the server. It will also register a callback client function for the OnReadyStateChange event of the XMLHttpRequest or ActiveXObject used to execute the callback function with the received data from the server, or to handle server side exception received as well. A client can call remote methods by executing Proxy.Invoke method and register a callback in which the result may by processed. Note that by using JSON, any client object will be

sent to the server and any object created by the server will arrive to the client as if it was created there. The serialization protocol allows simple extraction of result value or exception as well as natural representation of the calling arguments. There are a lot of RPC API-s for AJAX. Most of them use JSON, but there are some which use XML also. They are targeting different server languages like C#, Java, PHP, Python, Ruby, etc. The most complex ones have even server-to-client type registration, which enables better call integration by generating client wrappers for server methods using self-documented binaries, like C# attributes or Java annotations. This article was intended to describe a wide integration between web servers, RPC and serialization modules, by designing and developing from scratches a web server platform which provides SAPI extensibility for custom modules like AJAX RPC with JSON serialization. Its result is a platform which might help students to understand network concepts and how web application servers are built, as well as providing them a useful tool for developing their own complex web applications enriched with asynchronous communication with the server and remote procedure calling.

References

Crane D., Pascarello E. – AJAX in Action, Manning, 2006
JSON – http://www.json.org
JSON-RPC – http://json-rpc.org