Beginner’s Guide: How IIS Process ASP.NET Request

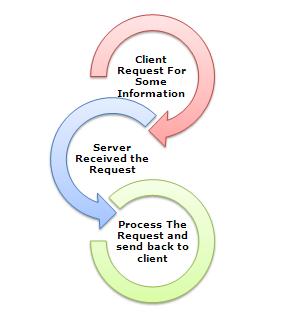
Introduction

When request come from client to the server a lot of operation is performed before sending response to the client. This is all about how IIS Process the request.  Here I am not going to describe the Page Life Cycle and there events, this article is all about the operation of IIS Level.  Before we start with the actual details, let’s start from the beginning so that each and everyone understand it’s details easily.  Please provide your valuable feedback and suggestion to improve this article.

What is Web Server ?

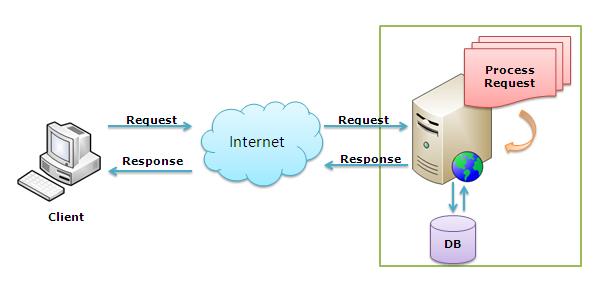
When we run our ASP.NET Web Application from visual studio IDE, VS Integrated ASP.NET Engine is responsible to execute all kind of asp.net requests and responses.  The process name is**“WebDev.WebServer.Exe”** which actually takw care of all request and response of an web application which is running from Visual Studio IDE.

Now, the name *“Web Server”*comes into picture when we want to host the application on a centralized location and wanted to access from many locations. Web server is responsible for handle all the requests that are coming from clients, process them and provide the responses.



What is IIS ?

*IIS (Internet Information Server)* is one of the most powerful web servers from Microsoft that is used to host your ASP.NET Web application. IIS has it’s own ASP.NET Process Engine  to handle the ASP.NET request. So, when a request comes from client to server, IIS takes that request and  process it and send response back to clients.

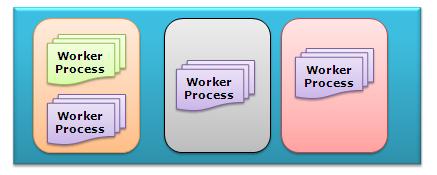


Request Processing :

Hope, till now it’s clear to you that what is Web server and IIS is and what is the use of them. Now let’s have a look how they do things internally. Before we move ahead, you have to know about two main concepts

1.    *Worker Process*  
2.   *Application Pool*

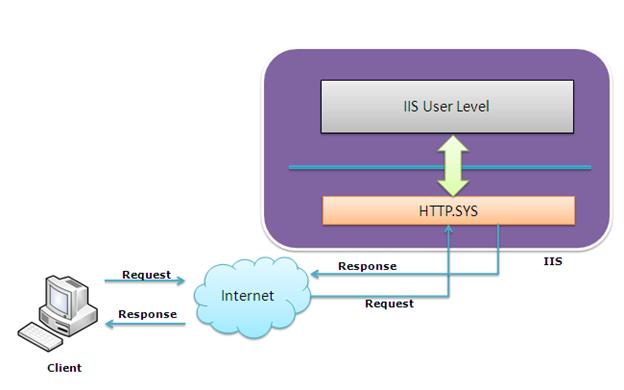
**Worker Process:**  Worker Process (*w3wp.exe*) runs the ASP.Net application in IIS. This process is responsible to manage all the request and response that are coming from client system.  All the ASP.Net functionality runs under the scope of worker process.  When a request comes to the server from a client worker process is responsible to generate the request and response. In a single word we can say worker process is the heart of ASP.NET Web Application which runs on IIS.  
**Application Pool:**Application pool is the container of worker process.  Application pools is used to separate sets of IIS worker processes that share the same configuration.  Application pools enables a better *security, reliability, and availability*for any web application.  The worker process serves as the process boundary that separates each application pool so that when one worker process or application is having an issue or recycles, other applications or worker processes are not affected. This makes sure that a particular web application doesn’t not impact other web application as they they are configured into different application pools.



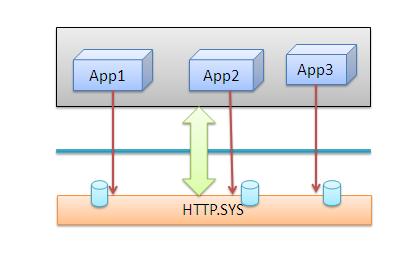
Application Pool with multiple worker process is called*“Web Garden”*.  
Now, I have covered all the basic stuff like Web server, Application Pool, Worker process. Now let’s have look how IIS process the request when a new request comes up from client.  
If we look into the IIS 6.0 Architecture, we can divided them into Two Layer

1.    *Kernel Mode*  
2.    *User Mode*

Now, *Kernel mode* is introduced with IIS 6.0, which contains the **HTTP.SYS**.  So whenever a request comes from Client to Server, it will hit *HTTP.SYS* First.

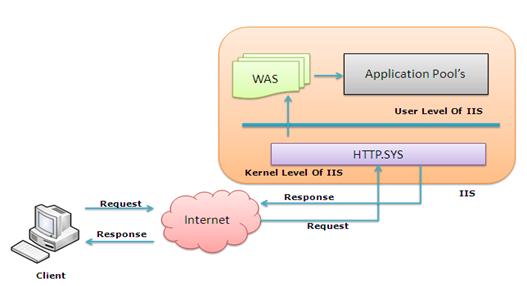


Now, HTTP.SYS is Responsible for pass the request to particular Application pool. *Now here is one question*, *How HTTP.SYS comes to know where to send the request?*  This is not a random pickup. Whenever we creates a new Application Pool, the ID of the Application Pool is being generated and it’s registered with the HTTP.SYS. So whenever HTTP.SYS Received the request from any web application, it checks for the Application Pool and based on the application pool it send the request.

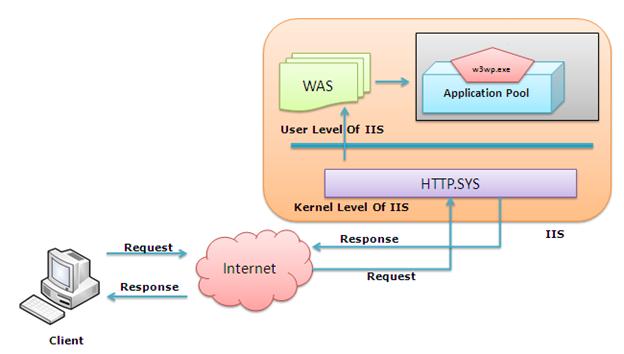


So, this was the first steps of IIS Request Processing.  
Till now, Client Requested for some information and request came to the Kernel level of IIS means at HTTP.SYS. HTTP.SYS has been identified the name of the application pool where to send. Now, let’s see how this request moves from HTTP.SYS to Application Pool.

In *User Level*of IIS, we have *Web Admin Services (WAS)* which takes the request from HTTP.SYS and pass it to the respective application pool.



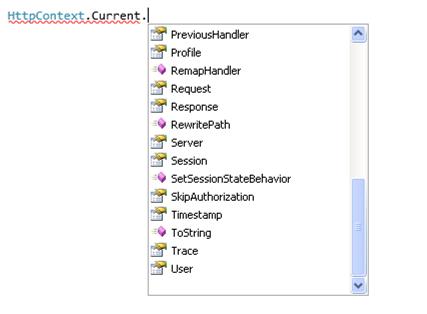
When Application pool receive the request, it simply pass the request to worker process (w3wp.exe) . The worker process*“w3wp.exe”* looks up the URL of the request in order to load the correct ISAPI extension. ISAPI extensions are the IIS way to handle requests for different resources. Once ASP.NET is installed, it installs its own ISAPI extension (aspnet\_isapi.dll)and adds the mapping into IIS.  
**Note :** Sometimes if we install IIS after installing asp.net, we need to register the extension with IIS using **aspnet\_regiis**command.



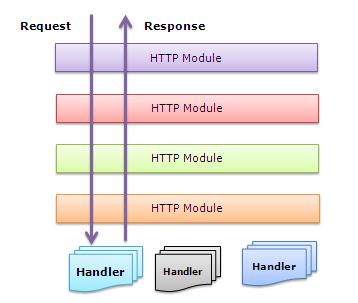
When Worker process loads the *aspnet\_isapi.dll*, it start an *HTTPRuntime*, which is the entry point of an application.*HTTPRuntime*is a class which calls the *ProcessRequest*method to start Processing.



When this methods called, a new instance of **HTTPContext**is been created.  Which is accessible using**HTTPContext.Current**  Properties. This object still remains alive during life time of object request.  Using HttpContext.Current we can access some other objects like *Request, Response, Session* etc.

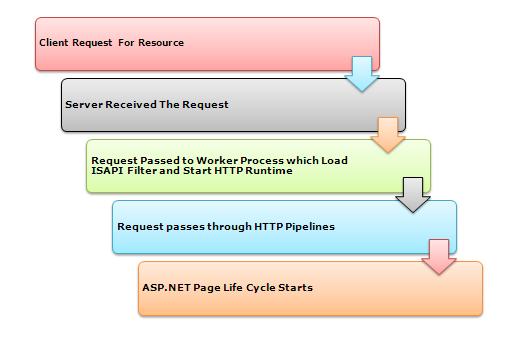


After that HttpRuntime load an *HttpApplication*object with the help of  *HttpApplicationFactory*class.. Each and every request should pass through the corresponding HTTPModule to reach to HTTPHandler, this list of module are configured by the HTTPApplication.  
Now, the concept comes called “**HTTPPipeline**”. It is called a pipeline because it contains a set of HttpModules ( For Both Web.config and Machine.config level) that intercept the request on its way to the HttpHandler. HTTPModules are classes that have access to the incoming request. We can also create our own HTTPModule if we need to handle anything during upcoming request and response.



HTTP Handlers are the *endpoints*in the HTTP pipeline. All request that are passing through the HTTPModule should reached to HTTPHandler.  Then  HTTP Handler  generates the output for the requested resource. So, when we requesting for any aspx web pages,   it returns the corresponding HTML output.

All the request now passes from  httpModule to  respective HTTPHandler then method and the ASP.NET Page life cycle starts.  This ends the IIS Request processing and start the ASP.NET Page Lifecycle.



Conclusion

When client request for some information from a web server, request first reaches to HTTP.SYS of IIS. HTTP.SYS then send the request to respective  Application Pool. Application Pool then forward the request to worker process to load the ISAPI Extension which will create an HTTPRuntime Object to Process the request via HTTPModule and HTTPHanlder. After that the ASP.NET Page LifeCycle events starts.

This was just overview of IIS Request Processing to let Beginner’s know how the request get processed in backend.  If you want to learn in details please check the link for Reference and further Study section.

For More

<http://www.west-wind.com/presentations/howaspnetworks/howaspnetworks.asp>

# ASP.Net Page Life Cycle And Events

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.UI;

using System.Web.UI.WebControls;

public partial class \_Default : System.Web.UI.Page

{

**//  Event 1**

    protected void Page\_PreInit(object sender, EventArgs e)

    {

        //  Use this event for the following:

        //  Check the IsPostBack property to determine whether this is the first time the page is being processed.

        //  Create or re-create dynamic controls.

        //  Set a master page dynamically.

        //  Set the Theme property dynamically.

        //  Note:

        //  If the request is a postback, the values of the controls have not yet been restored from view state. If you set a control property at this stage, its value might be overwritten in the next event.

    }

**//  Event 2**

    protected void Page\_Init(object sender, EventArgs e)

    {

        // Raised after all controls have been initialized and any skin settings have been applied. Use this event to read or initialize control properties.

    }

**//  Event 3**

   protected void Page\_InitComplete(object sender, EventArgs e)

    {

       // Raised by the  Page object. Use this event for processing tasks that require all initialization be complete.

    }

**//  Event 4:**Load view state for page

**//  Event 5**

    protected override void OnPreLoad(EventArgs e)

    {

        // Use this event if you need to perform processing on your page or control before the  Load event.

        // Before the Page instance raises this event, it loads view state for itself and all controls, and then processes any postback data included with the Request instance.

    }

**//  Event 6**

    protected void Page\_Load(object sender, EventArgs e)

    {

        // The  Page calls the  OnLoad event method on the  Page, then recursively does the same for each child control, which does the same for each of its child controls until the page and all controls are loaded.

        // Use the OnLoad event method to set properties in controls and establish database connections.

    }

**//  Event 7**

    // Control Events takes place after Page Load

**//  Event 8**

    protected void Page\_LoadComplete(object sender, EventArgs e)

    {

        // Use this event for tasks that require that all other controls on the page be loaded.

    }

**//  Event 9**

    //Databind events takes place here

**//  Event 10**

    protected override void OnPreRender(EventArgs e)

    {

       // Before this event occurs:

        // The Page object calls EnsureChildControls for each control and for the page.

        // Each data bound control whose DataSourceID property is set calls its DataBind method. For more information, see Data Binding Events for Data-Bound Controls later in this topic.

        // The PreRender event occurs for each control on the page. Use the event to make final changes to the contents of the page or its controls.

    }

**//  Event 11**

    protected override void OnSaveStateComplete(EventArgs e)

    {

        // Before this event occurs,  ViewState has been saved for the page and for all controls. Any changes to the page or controls at this point will be ignored.

        // Use this event perform tasks that require view state to be saved, but that do not make any changes to controls.

    }

    // Render stage goes here. This is not an event

**//  Event 12**

    protected void Page\_UnLoad(object sender, EventArgs e)

    {

        // This event occurs for each control and then for the page. In controls, use this event to do final cleanup for specific controls, such as closing control-specific database connections.

        // For the page itself, use this event to do final cleanup work, such as closing open files and database connections, or finishing up logging or other request-specific tasks.

        // Note:

        // During the unload stage, the page and its controls have been rendered, so you cannot make further changes to the response stream. If you attempt to call a method such as the Response.Write method, the page will throw an exception.

    }

}

<http://www.altafkhatri.com/Technical/ASP_NET_Page_Life_Cycle_And_Events/Common_ASP_NET/Page_Events>

# **Difference Between DataReader, DataSet, DataAdapter and DataTable in C#**

**DataReader**

DataReader is used to read the data from database and it is a read and forward only connection oriented architecture during fetch the data from database. DataReader will fetch the data very fast when compared with dataset. Generally we will use ExecuteReader object to bind data to datareader.

To bind DataReader data to GridView we need to write the code like as shown below:

Protected void BindGridview()

{

using (SqlConnection conn = new SqlConnection("Data Source=abc;Integrated Security=true;Initial Catalog=Test"))

{

con.Open();

SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location FROM Users", conn);

SqlDataReader sdr = cmd.ExecuteReader();

gvUserInfo.DataSource = sdr;

gvUserInfo.DataBind();

conn.Close();

}

}

* Holds the connection open until you are finished (don't forget to close it!).
* Can typically only be iterated over once
* Is not as useful for updating back to the database

**DataSet**

DataSet is a disconnected orient architecture that means there is no need of active connections during work with datasets and it is a collection of DataTables and relations between tables. It is used to hold multiple tables with data. You can select data form tables, create views based on table and ask child rows over relations. Also DataSet provides you with rich features like saving data as XML and loading XML data.

protected void BindGridview()

{

    SqlConnection conn = new SqlConnection("Data Source=abc;Integrated Security=true;Initial Catalog=Test");

    conn.Open();

    SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location FROM Users", conn);

    SqlDataAdapter sda = new SqlDataAdapter(cmd);

    DataSet ds = new DataSet();

    da.Fill(ds);

    gvUserInfo.DataSource = ds;

    gvUserInfo.DataBind();

}

**DataAdapter**

DataAdapter will acts as a Bridge between DataSet and database. This dataadapter object is used to read the data from database and bind that data to dataset. Dataadapter is a disconnected oriented architecture. Check below sample code to see how to use DataAdapter in code:

protected void BindGridview()

{

    SqlConnection con = new SqlConnection("Data Source=abc;Integrated Security=true;Initial Catalog=Test");

    conn.Open();

    SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location FROM Users", conn);

    SqlDataAdapter sda = new SqlDataAdapter(cmd);

    DataSet ds = new DataSet();

    da.Fill(ds);

    gvUserInfo.DataSource = ds;

    gvUserInfo.DataBind();

}

* Lets you close the connection as soon it's done loading data, and may even close it for you automatically
* All of the results are available in memory
* You can iterate over it as many times as you need, or even look up a specific record by index
* Has some built-in faculties for updating back to the database.

**DataTable**  
DataTable represents a single table in the database. It has rows and columns. There is no much difference between dataset and datatable, dataset is simply the collection of datatables.

protected void BindGridview()

{

     SqlConnection con = new SqlConnection("Data Source=abc;Integrated Security=true;Initial Catalog=Test");

     conn.Open();

     SqlCommand cmd = new SqlCommand("Select UserName, First Name,LastName,Location FROM Users", conn);

     SqlDataAdapter sda = new SqlDataAdapter(cmd);

     DataTable dt = new DataTable();

     da.Fill(dt);

     gridview1.DataSource = dt;

     gvidview1.DataBind();

}

**Microsoft SQL Server Connection String**

**connetionString="Data Source=ServerName;Initial Catalog=Databasename;**

**User ID=UserName;Password=Password"**

**Web.config and ConnectionString**

<configuration>

<connectionStrings>

<add name="SQLDbConnection"

connectionString="Server=SQlServerName; Database=YouDatabaseName; User Id=userid; password= password"

providerName="System.Data.SqlClient" />

</connectionStrings>

</configuration>

C#:

**string conn = ConfigurationManager.ConnectionStrings["SQLDbConnection"].ToString();**

|  |  |
| --- | --- |
| **PRIMARY KEY** | **UNIQUE KEY** |
| **NULL** | It doesn’t allow Null values. Because of this we refer PRIMARY KEY = UNIQUE KEY + Not Null CONSTRAINT | Allows Null value. But only one Null value. |
| **INDEX** | By default it adds a clustered index | By default it adds a UNIQUE non-clustered index |
| **LIMIT** | A table can have only one PRIMARY KEY Column[s] | A table can have more than one UNIQUE Key Column[s] |
| **CREATE SYNTAX** | Below is the sample example for defining a single column as a PRIMARY KEY column while creating a table:  CREATE TABLE dbo.Customer ( Id INT NOT NULL PRIMARY KEY, FirstName VARCHAR(100), LastName VARCHAR(100), City VARCHAR(50) )  Below is the Sample example for defining multiple columns as PRIMARY KEY. It also shows how we can give name for the PRIMARY KEY:  CREATE TABLE dbo.Customer ( Id INT NOT NULL, FirstName VARCHAR(100) NOT NULL, LastName VARCHAR(100), City VARCHAR(50), CONSTRAINT PK\_CUSTOMER PRIMARY KEY(Id,FirstName) ) | Below is the sample example for defining a single column as a UNIQUE KEY column while creating a table:  CREATE TABLE dbo.Customer ( Id INT NOT NULL UNIQUE, FirstName VARCHAR(100), LastName VARCHAR(100), City VARCHAR(50) )  Below is the Sample example for defining multiple columns as UNIQUE KEY. It also shows how we can give name for the UNIQUE KEY:  CREATE TABLE dbo.Customer ( Id INT NOT NULL, FirstName VARCHAR(100) NOT NULL, LastName VARCHAR(100), City VARCHAR(50), CONSTRAINT UK\_CUSTOMER UNIQUE(Id,FirstName) ) |
| **ALTER SYNTAX** | Below is the Syntax for adding PRIMARY KEY CONSTRAINT on a column when the table is already created:  ALTER TABLE dbo.Customer ADD CONSTRAINT PK\_CUSTOMER PRIMARY KEY (Id) | Below is the Syntax for adding UNIQUE KEY CONSTRAINTon a column when the table is already created:  ALTER TABLE dbo.Customer ADD CONSTRAINT UK\_CUSTOMER UNIQUE(Id) |
| **DROP SYNTAX** | Below is the Syntax for dropping a PRIMARY KEY:  ALTER TABLE dbo.Customer DROP CONSTRAINT PK\_CUSTOMER | Below is the Syntax for dropping a UNIQUE KEY:  ALTER TABLE dbo.Customer DROP CONSTRAINT UK\_CUSTOMER |

### Share this: