UNIT II SERVER SIDE PROGRAMING WITH NODE JS

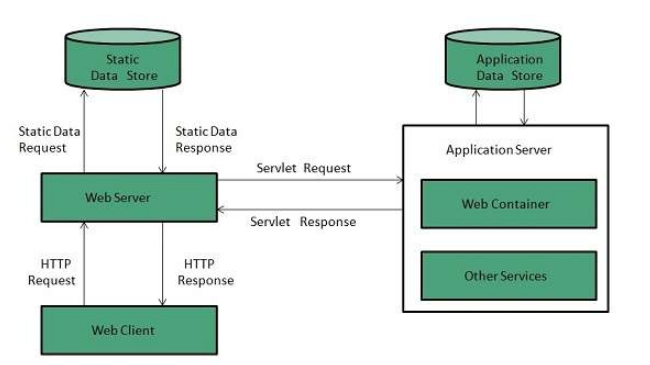
Introduction to Web Servers

**Web server** is a computer where the web content is stored. Basically web server is used to host the web sites but there exists other web servers also such as gaming, storage, FTP, email etc.

Web Server Working

Web server respond to the client request in either of the following two ways:

* Sending the file to the client associated with the requested URL.
* Generating response by invoking a script and communicating with database



**Key Points**

* When client sends request for a web page, the web server search for the requested page if requested page is found then it will send it to client with an HTTP response.
* If the requested web page is not found, web server will the send an **HTTP response:Error 404 Not found.**
* If client has requested for some other resources then the web server will contact to the application server and data store to construct the HTTP response.

### Concurrent Approach

Concurrent approach allows the web server to handle multiple client requests at the same time. It can be achieved by following methods:

* Multi-process
* Multi-threaded
* Hybrid method.

### Multi-processing

In this a single process (parent process) initiates several single-threaded child processes and distribute incoming requests to these child processes. Each of the child processes are responsible for handling single request.

It is the responsibility of parent process to monitor the load and decide if processes should be killed or forked.

### Multi-threaded

Unlike Multi-process, it creates multiple single-threaded process.

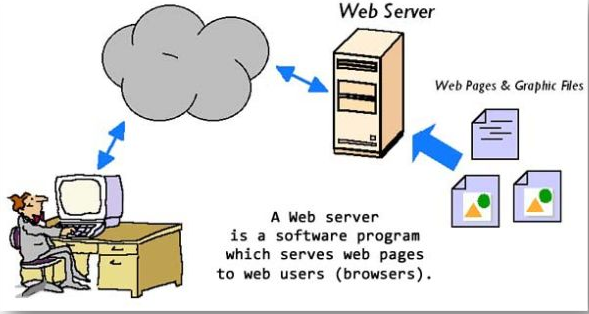
### Hybrid

It is combination of above two approaches. In this approach multiple process are created and each process initiates multiple threads. Each of the threads handles one connection. Using multiple threads in single process results in less load on system resources.

Server

A web server is a software program that serves web pages to web users (browsers).

A web server delivers requested web pages to users who enter the URL in a web browser. Every computer on the internet that contains a web site must have a web server program.  
  
The computer in which a web server program runs is also usually called a "web server". So, the term "web server" is used to represent both the server program and the computer in which the server program runs.

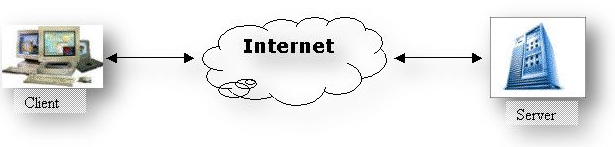


**Characteristics of web servers**  
A web server computer is just like any other computer. The basic characteristics of web servers are:

* It is always connected to the internet so that clients can access the web pages hosted by the web server.
* It always has an application called "web server" running.

In short, a "web server" is a computer that is connected to the internet/intranet and has software called "web server". The web server program will always be running in the computer. When a user tries to access a website hosted by the web server, it is actually the web server program that delivers the web page that the client asks for.  
  
All web sites in the internet are hosted in web servers sitting in various parts of the world.

**Is a Web Server hardware or software?**  
Mostly, Web server refers to the software program, that serves the clients request. But sometimes, the computer in which the web server program is installed is also called a "web server".



**Web Server, Behind the Scenes**  
When I type in an URL such as [http://www.ASP.NET](http://www.asp.net/) and click on some link, I dropped into this page.  
  
But what happens behind the scenes to bring you to this page and make you read this line of text.  
  
So now, let's see what is actually happening behind the scenes.  
  
The first you might do is, you type the http://www.asp.net/ in the address bar of your browser and press your return key.  
  
We could break this URL into the following two parts:

1. The protocol we will use to connect to the server (http)
2. The server name ( [ASP.NET](http://www.asp.net/) )

And the following process happens:

* The browser breaks up the URL into these parts and then it tries to communicate with the server looking up for the server name.
* The server is identified through a unique IP address but the alias for the IP address is maintained in the DNS Server or the Naming server.
* The browser looks up these naming servers, identifies the IP address of the server requested and gets the site and gets the HTML tags for the web page.
* Finally it displays the HTML Content in the browser.

**Where is my web server?**  
  
When you try to access a web site, you don't really need to know where the web server is located. The web server may be located in another city or country, but all you need to do is, type the URL of the web site you want to access in a web browser. The web browser will send this information to the internet and find the web server. Once the web server is located, it will request the specific web page from the web server program running in the server. The Web server program will process your request and send the resulting web page to your browser. It is the responsibility of your browser to format and display the web page to you.  
  
**How many web servers are needed for a web site?**  
Typically, there is only one web server required for a web site. But large web sites like Yahoo, Google, MSN and so on will have millions of visitors every minute. One computer cannot process such huge numbers of requests. So, they will have hundreds of servers deployed in various parts of the world so that can provide a faster response.  
  
**How many web sites can be hosted in one server?**  
A web server can host hundreds of web sites. Most of the small web sites in the internet are hosted on shared web servers. There are several web hosting companies who offer shared web hosting. If you buy a shared web hosting from a web hosting company, they will host your web site in their web server along with several other web sites for a fee.  
  
Examples of web server applications:

1. IIS
2. Apache

JavaScript in the Desktop with Nodejs

NodeJS

Node.js is a server-side platform built on Google Chrome's JavaScript Engine (V8 Engine). Node.js was developed by Ryan Dahl in 2009 and its latest version is v0.10.36.

Node.js is an open source, cross-platform runtime environment for developing server-side and networking applications. Node.js applications are written in JavaScript, and can be run within the Node.js runtime on OS X, Microsoft Windows, and Linux.

Node.js also provides a rich library of various JavaScript modules which simplifies the development of web applications using Node.js to a great extent.

Node.js = Runtime Environment + JavaScript Library

## Features of Node.js

Following are some of the important features that make Node.js the first choice of software architects.

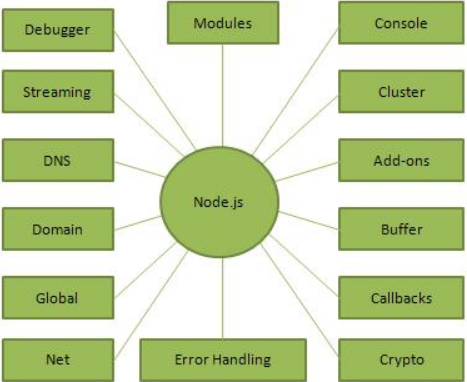
* **Asynchronous and Event Driven** − All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.
* **Very Fast** − Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.
* **Single Threaded but Highly Scalable** − Node.js uses a single threaded model with event looping. Event mechanism helps the server to respond in a non-blocking way and makes the server highly scalable as opposed to traditional servers which create limited threads to handle requests. Node.js uses a single threaded program and the same program can provide service to a much larger number of requests than traditional servers like Apache HTTP Server.
* **No Buffering** − Node.js applications never buffer any data. These applications simply output the data in chunks.
* **License** − Node.js is released under the MIT license

## Who Uses Node.js?

Following is the link on github wiki containing an exhaustive list of projects, application and companies which are using Node.js. This list includes eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, and Yammer to name a few.

Concepts

The following diagram depicts some important parts of Node.js which we will discuss in detail in the subsequent chapters.



Where to Use Node.js?

Following are the areas where Node.js is proving itself as a perfect technology partner.

* I/O bound Applications
* Data Streaming Applications
* Data Intensive Real-time Applications (DIRT)
* JSON APIs based Applications
* Single Page Applications

## Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows, and vim or vi can be used on windows as well as Linux or UNIX.

The files you create with your editor are called source files and contain program source code. The source files for Node.js programs are typically named with the extension "**.js**".

Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, and finally execute it.

## The Node.js Runtime

The source code written in source file is simply javascript. The Node.js interpreter will be used to interpret and execute your javascript code.

Node.js distribution comes as a binary installable for SunOS , Linux, Mac OS X, and Windows operating systems with the 32-bit (386) and 64-bit (amd64) x86 processor architectures.

Before creating an actual "Hello, World!" application using Node.js, let us see the components of a Node.js application. A Node.js application consists of the following three important components −

* **Import required modules** − We use the **require** directive to load Node.js modules.
* **Create server** − A server which will listen to client's requests similar to Apache HTTP Server.
* **Read request and return response** − The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response.

NPM

**npm** (originally short for **Node Package Manager**) is a package manager for the JavaScript programming language maintained by npm, Inc. npm is the default package manager for the JavaScript runtime environment Node.js. It consists of a command line client, also called npm, and an online database of public and paid-for private packages, called the npm registry. The registry is accessed via the client, and the available packages can be browsed and searched via the npm website. The package manager and the registry are managed by npm, Inc.

**npm** is the world's largest **Software Registry**.

The registry contains over 800,000 **code packages**.

**Open-source** developers use **npm** to **share** software.

Many organizations also use npm to manage private development.

**npm** is free to use.

Install

**npm** includes a **CLI** (Command Line Client) that can be used to download and install software:

### Windows Example

C:\>npm install <package>

### Mac OS Example

>npm install <package>

## Installing npm

**npm** is installed with **Node.js**

This means that you have to install Node.js to get npm installed on your computer.

Download Node.js from the official Node.js web site: [https://nodejs.org](https://nodejs.org/)

## Software Package Manager

The name **npm** (Node Package Manager) stems from when npm first was created as a package manager for Node.js.

All **npm** packages are defined in files called **package.json**.

The content of package.json must be written in **JSON**.

At least two fields must be present in the definition file: **name** and **version**.

### Example

{  
"name" : "foo",  
"version" : "1.2.3",  
"description" : "A package for fooing things",  
"main" : "foo.js",  
"keywords" : ["foo", "fool", "foolish"],  
"author" : "John Doe",  
"licence" : "ISC"  
}

Managing Dependencies

**npm** can manage **dependencies**.

**npm** can (in one command line) install all the dependencies of a project.

Dependencies are also defined in **package.json**.

Serving file with HTTP Module

When you view a webpage in your browser, you are making a request to another computer on the internet, which then provides you the webpage as a response. That computer you are talking to via the internet is a web server. A web server receives HTTP requests from a client, like your browser, and provides an HTTP response, like an HTML page or JSON from an API.

A lot of software is involved for a server to return a webpage. This software generally falls into two categories: frontend and backend. Front-end code is concerned with how the content is presented, such as the color of a navigation bar and the text styling. Back-end code is concerned with how data is exchanged, processed, and stored. Code that handles network requests from your browser or communicates with the database is primarily managed by back-end code.

Node.js allows developers to use JavaScript to write back-end code, even though traditionally it was used in the browser to write front-end code. Having both the frontend and backend together like this reduces the effort it takes to make a web server, which is a major reason why Node.js is a popular choice for writing back-end code.

One of the most fundamental uses of an HTTP server is to serve static files to a user's browser, like CSS, JavaScript, or image files. Beyond normal browser usage, there are thousands of other reasons you'd need to serve a static files, like for downloading music or scientific data. Either way, you'll need to come up with a simple way to let the user download these files from your server.

One simple way to do this is to create a Node HTTP server. As you probably know, Node.js excels at handling I/O-intensive tasks, which makes it a natural choice here. You can choose to create your own simple HTTP server from the base http module that's shipped with Node, or you can use the popular serve-static package, which provides many common features of a static file server.

The end goal of our static server is to let the user specify a file path in the URL and have that file returned as the contents of the page. However, the user shouldn't be able to specify just any path on our server, otherwise a malicious user could try to take advantage of a misconfigured system and steal sensitve information. A simple attack might look like this: localhost:8080/etc/shadow. Here the attacker would be requesting the /etc/shadow file. To prevent these kinds of attacks, we should be able to tell the server to only allow the user to download certain files, or only files from certain directories (like /var/www/my-website/public).

Creating your own

This section is meant for those of you needing a more custom option, or for those wanting to learn how static servers (or just servers in general) work. If you have a fairly common use-case, then you'd be better off moving on to the next section and start working directly with the serve-static module.

While creating your own server from the http module takes a bit of work, it can be very rewarding in showing you how servers work underneath, which includes trade-offs for performance and security concerns that need to be taken in to account. Although, it is fairly easily create your own custom Node static server using only the built-in http module, so we don't have to dive too deep in to the internals of an HTTP server.

Obviously the http module isn't going to be as easy to use as something like Express, but it's a great starting point as an HTTP server. Here I'll show you how to create a simple static HTTP server, which you can then add on to and customize to your liking.

# FileServer function

The http.FileServer() function provides the functionality to serve the entire file-system directory with **indexes**. Let’s have a look at its syntax.

func FileServer(root **FileSystem**) **Handler**

As you can see, it returns a Handler object which makes it a perfect candidate to be used as handler in [ListenAndServe](https://golang.org/pkg/net/http/#ListenAndServe) function or the [Handle](https://golang.org/pkg/net/http/#Handle) function. It takes an argument of [FileSystem](https://golang.org/pkg/net/http/" \l "FileSystem" \t "_blank) interface type.

type FileSystem interface {  
 Open(name string) (http.[File](https://golang.org/pkg/net/http/#File), error)  
}

The root argument represents the file-system directory from which the content will be served. Luckily, we don’t have to create our own implementation of FileSystem interface.

## http.Dir type

The [Dir](https://golang.org/pkg/net/http/#Dir) type may look like a **function** but it is an alias of string data type and it implements Open method defined by the FileSystem interface. We can call the Dir type like a function which is nothing but a **type-casting** syntax.

The string value passed to the type-casting syntax is a **relative** or an **absolute** directory on the native file-system.

var **fs** FileSystem = http.Dir("/tmp")

💡 *The http.Dir type does not****exclude****. prefixed files like .git or .htpasswd, so be careful what you are serving. Else, you can create your own implemention FileSystem interface. If we pass empty directory string “”, the Go will use the os.*[*Executable*](https://golang.org/pkg/os/#Executable)*() as the default directory to serve files from.*

## Serving a directory

I have created a simple **temporary folder** under my user directory to demonstrate this example. It looks as follows.

/Users/Uday.Hiwarale/tmp  
├── .htpasswd  
├── files  
| └── test.pdf  
├── main.js  
├── page.html  
└── style.css

Let’s write a simple Go HTTP server to serve /Users/Uday.Hiwarale/tmp directory using http.FileServer handler and http.Dir type.

Let's start by just initializing and running our HTTP server:

"use strict";

var http = require('http');

var staticServe = function(req, res) {

res.statusCode = 200;

res.write('ok');

return res.end();

};

var httpServer = http.createServer(staticServe);

httpServer.listen(8080);

Introduction to Express Framework

Express.js is a web framework for Node.js. It is a fast, robust and asynchronous in nature.

Our Express.js tutorial includes all topics of Express.js such as Express.js installation on windows and linux, request object, response object, get method, post method, cookie management, scaffolding, file upload, template etc.

What is Express.js?

Express is a fast, assertive, essential and moderate web framework of Node.js. You can assume express as a layer built on the top of the Node.js that helps manage a server and routes. It provides a robust set of features to develop web and mobile applications.

Let's see some of the core features of Express framework:

* It can be used to design single-page, multi-page and hybrid web applications.
* It allows to setup middlewares to respond to HTTP Requests.
* It defines a routing table which is used to perform different actions based on HTTP method and URL.
* It allows to dynamically render HTML Pages based on passing arguments to templates.

Why use Express

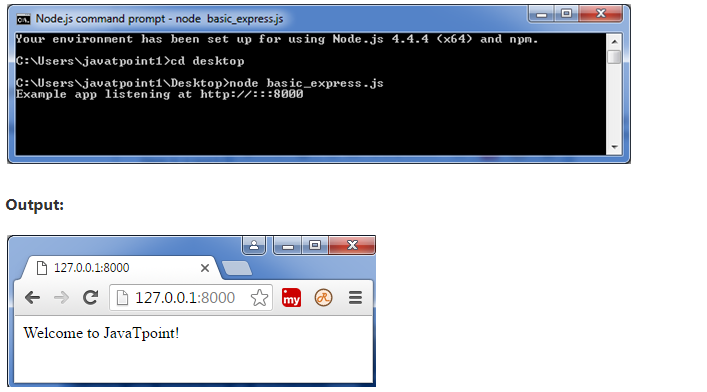
* Ultra fast I/O
* Asynchronous and single threaded
* MVC like structure
* Robust API makes routing easy

How does Express look like

Let's see a basic Express.js app.

**File: basic\_express.js**

1. var express = require('express');
2. var app = express();
3. app.get('/', function (req, res) {
4. res.send('Welcome to JavaTpoint!');
5. });
6. var server = app.listen(8000, function () {
7. var host = server.address().address;
8. var port = server.address().port;
9. console.log('Example app listening at http://%s:%s', host, port);
10. });



Express is a minimal and flexible Node.js web application framework that provides a robust set of features to develop web and mobile applications. It facilitates the rapid development of Node based Web applications. Following are some of the core features of Express framework −

* Allows to set up middlewares to respond to HTTP Requests.
* Defines a routing table which is used to perform different actions based on HTTP Method and URL.
* Allows to dynamically render HTML Pages based on passing arguments to templates.

[Node](https://nodejs.org/) (or more formally Node.js) is an open-source, cross-platform runtime environment that allows developers to create all kinds of server-side tools and applications in [JavaScript](https://developer.mozilla.org/en-US/docs/Glossary/JavaScript). The runtime is intended for use outside of a browser context (i.e. running directly on a computer or server OS). As such, the environment omits browser-specific JavaScript APIs and adds support for more traditional OS APIs including HTTP and file system libraries.

From a web server development perspective Node has a number of benefits:

* Great performance! Node was designed to optimize throughput and scalability in web applications and is a good solution for many common web-development problems (e.g. real-time web applications).
* Code is written in "plain old JavaScript", which means that less time is spent dealing with "context shift" between languages when you're writing both client-side and server-side code.
* JavaScript is a relatively new programming language and benefits from improvements in language design when compared to other traditional web-server languages (e.g. Python, PHP, etc.) Many other new and popular languages compile/convert into JavaScript so you can also use TypeScript, CoffeeScript, ClojureScript, Scala, LiveScript, etc.
* The node package manager (NPM) provides access to hundreds of thousands of reusable packages. It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.
* Node.js is portable. It is available on Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS, and NonStop OS. Furthermore, it is well-supported by many web hosting providers, that often provide specific infrastructure and documentation for hosting Node sites.
* It has a very active third party ecosystem and developer community, with lots of people who are willing to help.

You can use Node.js to create a simple web server using the Node HTTP package.

### Hello Node.js

The following example creates a web server that listens for any kind of HTTP request on the URL http://127.0.0.1:8000/ — when a request is received, the script will respond with the string: "Hello World". If you have already installed node, you can follow these steps to try out the example:

1. Open Terminal (on Windows, open the command line utility)
2. Create the folder where you want to save the program, for example, test-node and then enter it by entering the following command into your terminal:
3. cd test-node

Copy to Clipboard

1. Using your favorite text editor, create a file called hello.js and paste the following code into it:
2. // Load HTTP module
3. const http = require("http");
4. const hostname = "127.0.0.1";
5. const port = 8000;
6. // Create HTTP server
7. const server = http.createServer(function(req, res) {
8. // Set the response HTTP header with HTTP status and Content type
9. res.writeHead(200, {'Content-Type': 'text/plain'});
10. // Send the response body "Hello World"
11. res.end('Hello World\n');
12. });
13. // Prints a log once the server starts listening
14. server.listen(port, hostname, function() {
15. console.log(`Server running at http://${hostname}:${port}/`);
16. })

Copy to Clipboard

1. Save the file in the folder you created above.
2. Go back to the terminal and type the following command:
3. node hello.js

Copy to Clipboard

Finally, navigate to http://localhost:8000 in your web browser; you should see the text "**Hello World**" in the upper left of an otherwise empty web page.

## Web Frameworks

Other common web-development tasks are not directly supported by Node itself. If you want to add specific handling for different HTTP verbs (e.g. GET, POST, DELETE, etc.), separately handle requests at different URL paths ("routes"), serve static files, or use templates to dynamically create the response, Node won't be of much use on its own. You will either need to write the code yourself, or you can avoid reinventing the wheel and use a web framework!

## Introducing Express

Express is the most popular Node web framework, and is the underlying library for a number of other popular Node web frameworks. It provides mechanisms to:

* Write handlers for requests with different HTTP verbs at different URL paths (routes).
* Integrate with "view" rendering engines in order to generate responses by inserting data into templates.
* Set common web application settings like the port to use for connecting, and the location of templates that are used for rendering the response.
* Add additional request processing "middleware" at any point within the request handling pipeline.

## Where did Node and Express come from?

Node was initially released, for Linux only, in 2009. The NPM package manager was released in 2010, and native Windows support was added in 2012.

Express was initially released in November 2010 and is currently on version 4.17.3 of the API (with 5.0 in "beta"). You can check out the [changelog](https://expressjs.com/en/changelog/4x.html) for information about changes in the current release, and [GitHub](https://github.com/expressjs/express/blob/master/History.md) for more detailed historical release notes.

## How popular are Node and Express?

The popularity of a web framework is important because it is an indicator of whether it will continue to be maintained, and what resources are likely to be available in terms of documentation, add-on libraries, and technical support.

There isn't any readily-available and definitive measure of the popularity of server-side frameworks (although you can estimate popularity using mechanisms like counting the number of GitHub projects and StackOverflow questions for each platform). A better question is whether Node and Express are "popular enough" to avoid the problems of unpopular platforms. Are they continuing to evolve? Can you get help if you need it? Is there an opportunity for you to get paid work if you learn Express?

Based on the number of high profile companies that use Express, the number of people contributing to the codebase, and the number of people providing both free and paid for support, then yes, Express is a popular framework!

## Is Express opinionated?

Web frameworks often refer to themselves as "opinionated" or "unopinionated".

Opinionated frameworks are those with opinions about the "right way" to handle any particular task. They often support rapid development in a particular domain (solving problems of a particular type) because the right way to do anything is usually well-understood and well-documented. However they can be less flexible at solving problems outside their main domain, and tend to offer fewer choices for what components and approaches they can use.

Unopinionated frameworks, by contrast, have far fewer restrictions on the best way to glue components together to achieve a goal, or even what components should be used. They make it easier for developers to use the most suitable tools to complete a particular task, albeit at the cost that you need to find those components yourself.

Express is unopinionated. You can insert almost any compatible middleware you like into the request handling chain, in almost any order you like. You can structure the app in one file or multiple files, and using any directory structure. You may sometimes feel that you have too many choices!

## What does Express code look like?

In a traditional data-driven website, a web application waits for HTTP requests from the web browser (or other client). When a request is received the application works out what action is needed based on the URL pattern and possibly associated information contained in POST data or GET data. Depending on what is required it may then read or write information from a database or perform other tasks required to satisfy the request. The application will then return a response to the web browser, often dynamically creating an HTML page for the browser to display by inserting the retrieved data into placeholders in an HTML template.

Express provides methods to specify what function is called for a particular HTTP verb (GET, POST, SET, etc.) and URL pattern ("Route"), and methods to specify what template ("view") engine is used, where template files are located, and what template to use to render a response. You can use Express middleware to add support for cookies, sessions, and users, getting POST/GET parameters, etc. You can use any database mechanism supported by Node (Express does not define any database-related behavior).

The following sections explain some of the common things you'll see when working with Express and Node code.

### [Helloworld Express](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction#helloworld_express)

First lets consider the standard Express [Hello World](https://expressjs.com/en/starter/hello-world.html) example (we discuss each part of this below, and in the following sections).

**Note:** If you have Node and Express already installed (or if you install them as shown in the [next article](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/development_environment)), you can save this code in a text file called **app.js** and run it in a bash command prompt by calling:

**node ./app.js**

const express = require('express');

const app = express();

const port = 3000;

app.get('/', function(req, res) {

res.send('Hello World!')

});

app.listen(port, function() {

console.log(`Example app listening on port ${port}!`)

});

Copy to Clipboard

The first two lines require() (import) the express module and create an Express application. This object, which is traditionally named app, has methods for routing HTTP requests, configuring middleware, rendering HTML views, registering a template engine, and modifying application settings that control how the application behaves (e.g. the environment mode, whether route definitions are case sensitive, etc.)

The middle part of the code (the three lines starting with app.get) shows a route definition. The app.get() method specifies a callback function that will be invoked whenever there is an HTTP GET request with a path ('/') relative to the site root. The callback function takes a request and a response object as arguments, and calls send() on the response to return the string "Hello World!"

The final block starts up the server on a specified port ('3000') and prints a log comment to the console. With the server running, you could go to localhost:3000 in your browser to see the example response returned.

### Importing and creating modules

A module is a JavaScript library/file that you can import into other code using Node's require() function. Express itself is a module, as are the middleware and database libraries that we use in our Express applications.

The code below shows how we import a module by name, using the Express framework as an example. First we invoke the require() function, specifying the name of the module as a string ('express'), and calling the returned object to create an Express application. We can then access the properties and functions of the application object.

const express = require('express');

const app = express();

Copy to Clipboard

You can also create your own modules that can be imported in the same way.

To make objects available outside of a module you just need to expose them as additional properties on the exports object. For example, the **square.js** module below is a file that exports area() and perimeter() methods:

exports.area = function(width) { return width \* width; };

exports.perimeter = function(width) { return 4 \* width; };

Copy to Clipboard

We can import this module using require(), and then call the exported method(s) as shown:

const square = require('./square'); // Here we require() the name of the file without the (optional) .js file extension

console.log('The area of a square with a width of 4 is ' + square.area(4));

Copy to Clipboard

If you want to export a complete object in one assignment instead of building it one property at a time, assign it to module.exports as shown below (you can also do this to make the root of the exports object a constructor or other function):

module.exports = {

area: function(width) {

return width \* width;

},

perimeter: function(width) {

return 4 \* width;

}

};

### Using asynchronous APIs

JavaScript code frequently uses asynchronous rather than synchronous APIs for operations that may take some time to complete. A synchronous API is one in which each operation must complete before the next operation can start. For example, the following log functions are synchronous, and will print the text to the console in order (First, Second).

console.log('First');

console.log('Second');

Copy to Clipboard

By contrast, an asynchronous API is one in which the API will start an operation and immediately return (before the operation is complete). Once the operation finishes, the API will use some mechanism to perform additional operations. For example, the code below will print out "Second, First" because even though setTimeout() method is called first, and returns immediately, the operation doesn't complete for several seconds.

setTimeout(function() {

console.log('First');

}, 3000);

console.log('Second');

Copy to Clipboard

Using non-blocking asynchronous APIs is even more important on Node than in the browser because Node is a single-threaded event-driven execution environment. "Single threaded" means that all requests to the server are run on the same thread (rather than being spawned off into separate processes). This model is extremely efficient in terms of speed and server resources, but it does mean that if any of your functions call synchronous methods that take a long time to complete, they will block not just the current request, but every other request being handled by your web application.

There are a number of ways for an asynchronous API to notify your application that it has completed. The most common way is to register a callback function when you invoke the asynchronous API, that will be called back when the operation completes. This is the approach used above.

The Express application object also provides methods to define route handlers for all the other HTTP verbs, which are mostly used in exactly the same way:

checkout(), copy(), **delete()**, **get()**, head(), lock(), merge(), mkactivity(), mkcol(), move(), m-search(), notify(), options(), patch(), **post()**, purge(), **put()**, report(), search(), subscribe(), trace(), unlock(), unsubscribe().

Server Side Rendering Template Engine

Most front-end developers have needed to work with server-side templates at some point. They may not have been called that—there was a time when these templates were simply called “PHP pages”, “JSPs”, or similar, before the push to apply separation of concerns on the web. These days it’s more common to see pages and views rendered by any back-end framework trimmed of as much business logic as possible.

Node is no different. Just as those other application frameworks need a way to separate the HTML produced from the data that populates it, so does ours. We want to be able to create a set of views loosely coupled to our application logic and have the application decide when to render them, and with what data.

Server-side templates provide an easy method of managing the dynamic generation of HTML code. But they can also fall victim to server-side template injection (SSTI). Take a look at the basics of server-side web templates, and how to detect, identify and mitigate SSTI in web applications.

Server-side templates allow developers to [pre-populate a web page](https://securityintelligence.com/news/fbi-announcement-watch-out-for-spoofed-websites/) with custom user data directly on the server. After all, it is often faster to make all the requests within a server than to make extra browser-to-server roundtrips for them. This is different from client-side templating, where browsers tend to load a template that affects the overall end-user experience.

Feature-rich web applications often embed user input in web templates to offer flexible functionality and shortcuts. This creates a vulnerability easily mistaken for cross-site scripting (XXS).

## How Do Server-Side Templates Work?

The most common approach for web frameworks to generate HTML dynamically is using templates. A template contains the static parts of the desired HTML output, as well as some special syntax describing how it will insert dynamic content.

The template system isn’t safe from untrusted template authors. For example, a site shouldn’t allow its users to provide their own templates, since template authors can perform XSS attacks and access properties of template variables that may contain sensitive information.

The template engines then process template files created by the developers, which helps to populate dynamic data into web pages. The template engine creates the HTML output response when an HTTP request comes in.

Some of the server-side template engines that are most often used are Smarty, Mako, Twig and Jinja2.

* For PHP: Smarty, Twig
* Java: Velocity, Freemaker
* Python: Jinja, Mako, Tornado

Jinja, also known as Jinja2, is a popular Python template engine written as a self-contained open-source project.

For example:

Hello {{user.name}}

These provide Django templates that adopt the model, view and templates framework, instead of the more popular model, view and controller framework, as part of a larger web framework. This can make them difficult to reuse in projects outside their coupled library.

## Server-Side Template Injection

So, how does an SSTI vulnerability in a web application work? The attacker injects malicious input or invalid syntax into a template to execute commands on the server-side.

This vulnerability occurs when the template engine contains embedded invalid user input, which can lead to a remote code execution (RCE) attack.

Example:

Template = ‘Username:’  + USER\_INPUT

render(template)

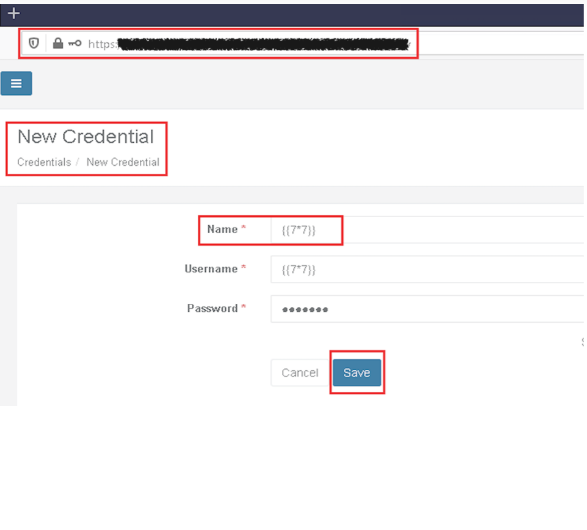
Here, the USER\_INPUT is part of the template. This allows the user to input a username or any input parameter of a web application. Because the templates can accept arbitrary code, a user might be able to input some differential calculus like:

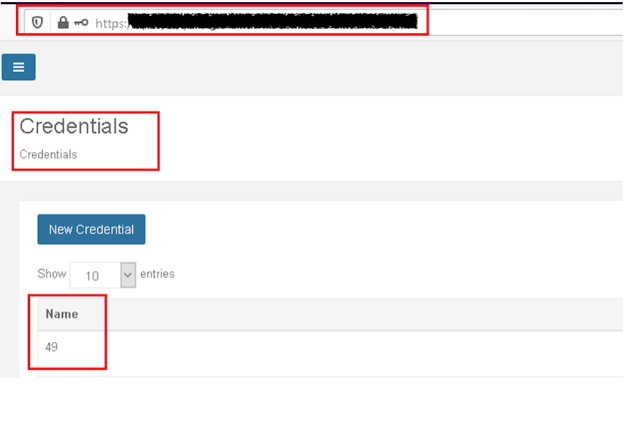
Username: {{ 7 \* 7 }}

Username 49

In the above example, the server-side template engine processes {{7\*7}} to give an output of 49. This shows that the web application is vulnerable to SSTI.

The below screenshots shows an example of how to add an SSTI payload in an input parameter ‘name,’ which results in an output of 49 that is evaluated server-side.

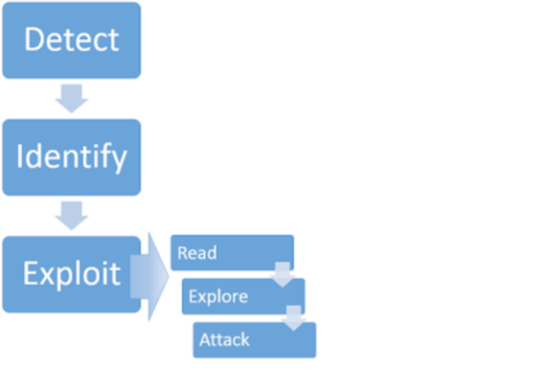




A malicious user can even enter a malicious code to compromise the system to gain complete remote access:

Ex -  bio: {{ malicious code() }}

The following chart shows the path for those finding and exploiting SSTI.



How to Detect Server-Side Template Injection

Defenders can find or detect SSTI via automated scripts or through manual methods. Two of the manual methods are:

* Plain text detection method
* Code context detection method

**Detect Plaintext:**

To detect SSTI in a plain text context, the tester can use some of the common template expressions in the form of a payload that is used by various template engines. They can then observe the server’s HTTP responses in the error message.

Here are some common template expression examples:

=${7\*3}  
={{7\*3}}  
=<%= 7\*3 %>

**Detect Code Context**

To test the code context, the tester needs to construct a payload to fetch a blank or error response from the server.

In the below request, the tester inserts an HTTP GET parameter into the variable ‘personal\_greeting’ in a template statement.

personal\_greeting=username  
Hello user01

The payload server responds with a blank ‘Hello:’

personal\_greeting=username

Hello

The next step is breaking out of the template statement and injecting the HTML tag after using the below payload.

personal\_greeting=username}}<tag>  
Hello user01 <tag>

How to Identify Server-Side Template Engines

After the tester detects the point of injection, they can identify the template engine on the basis of various mathematical or template expressions.

Identifying the SSTI depends upon the malformed/malicious payload crafted to be used in the user input. As a result, the server may display the error message or flag an exception.

Example 1:

POST /some-endpoint HTTP/1.1  
Host: victim-website.com  
parameter=${{<%[%'"}}%\.

To detect the vulnerability, the tested can inject the above shown polyglot payload (the sequence of special characters) at the user input parameter. If the vulnerability exists, the server’s response with an error message can reflect the underlying template engine.

The below screenshot shows an error output from an application that indicates the kind of template engine used.

The probe {{7\*’7′}} would result in 49 in Twig, 7777777 in Jinja2, and neither if no template language is in use.

Twig:

Custom\_email={{7\*’7’}}

49

Jinja:

Custom\_email={{7\*’7’}}

7777777

How to Remediate SSTI

With SSTI, once the threat actors identify the template engine, they can also gain further control of the server by using the remote code execution exploit. The untrusted users can identify the objects, methods and properties. Such exposure can further lead to information disclosure about application passwords, application programming interface keys and more.

Therefore, an SSTI vulnerability targeting the template engines and allowing untrusted users to edit templates introduces an array of serious risks. It is important for developers to take remediation steps that depend upon the different template engines in place.

The following remediation steps are abstract and can be applied to any template engine.

Sanitization

Templates should not be created from user-controlled input. User input should be passed to the template using template parameters. Sanitize the input before passing it into the templates by removing unwanted and risky characters before parsing the data. This minimizes the vulnerabilities for any malicious probing of your templates.

Static Files

Static files are files that clients download as they are from the server. Create a new directory, **public**. Express, by default does not allow you to serve static files. You need to enable it using the following built-in middleware.

app.use(express.static('public'));

**Note** − Express looks up the files relative to the static directory, so the name of the static directory is not part of the URL.

Note that the root route is now set to your public dir, so all static files you load will be considering public as root. To test that this is working fine, add any image file in your new **public** dir and change its name to "**testimage.jpg**". In your views, create a new view and include this file like −

html

head

body

h3 Testing static file serving:

img(src = "/testimage.jpg", alt = "Testing Image

Multiple Static Directories

We can also set multiple static assets directories using the following program −

var express = require('express');

var app = express();

app.use(express.static('public'));

app.use(express.static('images'));

app.listen(3000);

Virtual Path Prefix

We can also provide a path prefix for serving static files. For example, if you want to provide a path prefix like **'/static'**, you need to include the following code in your **index.js** file −

var express = require('express');

var app = express();

app.use('/static', express.static('public'));

app.listen(3000);

Now whenever you need to include a file, for example, a script file called main.js residing in your public directory, use the following script tag −

<script src = "/static/main.js" />

Async/await In Javascript

We all know that JavaScript is Synchronous in nature which means that it has an event loop that allows you to queue up an action that won’t take place until the loop is available sometime after the code that queued the action has finished executing.

But there’s a lot of functionalities in our program which makes our code Asynchronous and one of them is the Async/Await functionality. Async/Await is the extension of promises which we get as a support in the language.

* Async and Await both are considered as special keywords which are provided by [ES6](https://www.geeksforgeeks.org/introduction-to-es6/)in order to perform some asynchronous data operations. Even synchronous operations could also be performed using these keywords.
* [Async](https://www.geeksforgeeks.org/understanding-the-async-in-javascript/) keyword is used along with the function declaration which specifies that this function is now able to accept all types of asynchronous events on itself.
* In other words, the async keyword is used along with functions (or methods) which enables them to receive all types of asynchronous data easily.
* Async keyword usage along with the functions always returns a [promise](https://www.geeksforgeeks.org/javascript-promises/) at the end along with its state (pending or resolved or rejected).
* Await is used inside the async function which is though useful for the waiting purpose of the result.
* Await basically waits for the results which are particularly to be fetched from the source from which that async function is about to fetch the data.
* Await takes a little time to fetch the results from the source (like API) and thereafter along with the async function returns the result in the form of a promise.
* Await can also be used if Async is used along with the function declaration.

**Working of Async and Await:**

* Whenever a user declares a function with the async keyword in its declaration, it automatically implies a fact that this function or a method is ready to receive all of the asynchronous events on itself.
* After this, we generally use await keyword which then waits for the results and then fetches it successfully.
* After that, we store the fetched result in some random variable then display the result or use that result for several other purposes as per the requirement.
* A function that uses Async/Await keywords will eventually hold their results for some little amount of time as compared with other normal functions.
* Upon successful completion, the result may be used for further data operations within the code itself or may be displayed on the browser’s console successfully.

**Async:** It simply allows us to write promises based code as if it was synchronous and it checks that we are not breaking the execution thread. It operates asynchronously via the event-loop. Async functions will always return a value. It makes sure that a promise is returned and if it is not returned then JavaScript automatically wraps it in a promise which is resolved with its value.

**Example-1:**

* javascript

|  |
| --- |
| const getData = async() => {      var data = "Hello World";      return data;  }  getData().then(data => console.log(data)); |

**Output:**

Hello World

Ex: 2

async function f() {

return 1;

}

f().then(alert); // 1

**Await:** Await function is used to wait for the promise. It could be used within the async block only. It makes the code wait until the promise returns a result. It only makes the async block wait.

**Example-2:**

* javascript

|  |
| --- |
| const getData = async() => {      var y = await "Hello World";      console.log(y);  }  console.log(1);  getData();  console.log(2); |

**Output:**

1

2

Hello World

Ex:2

async function f() {

let promise = new Promise((resolve, reject) => {

setTimeout(() => resolve("done!"), 1000)

});

let result = await promise; // wait until the promise resolves (\*)

alert(result); // "done!"

}

f();

Fetching JSON from Express

**JSON** (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

* A collection of name/value pairs. In various languages, this is realized as an *object*, record, struct, dictionary, hash table, keyed list, or associative array.
* An ordered list of values. In most languages, this is realized as an *array*, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

### [JSON structure](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON#json_structure)

As described above, JSON is a string whose format very much resembles JavaScript object literal format. You can include the same basic data types inside JSON as you can in a standard JavaScript object — strings, numbers, arrays, booleans, and other object literals. This allows you to construct a data hierarchy, like so:

{

"squadName": "Super hero squad",

"homeTown": "Metro City",

"formed": 2016,

"secretBase": "Super tower",

"active": true,

"members": [

{

"name": "Molecule Man",

"age": 29,

"secretIdentity": "Dan Jukes",

"powers": [

"Radiation resistance",

"Turning tiny",

"Radiation blast"

]

},

{

"name": "Madame Uppercut",

"age": 39,

"secretIdentity": "Jane Wilson",

"powers": [

"Million tonne punch",

"Damage resistance",

"Superhuman reflexes"

]

},

{

"name": "Eternal Flame",

"age": 1000000,

"secretIdentity": "Unknown",

"powers": [

"Immortality",

"Heat Immunity",

"Inferno",

"Teleportation",

"Interdimensional travel"

]

}

]

}

In JavaScript, we can use the require() method to load files and modules. This takes the path of the local file where it has been saved. With the help of the console.log() function, it loads the data in the server and displays it.

For example,

const jsonData= require('./students.json');

console.log(jsonData);

Output:

"Students" : [

{

"firstName":"Ram","lastName":"Sharma"

}

]

}

## [Use the fetch() Function to Load JSON Files in JavaScript](https://www.delftstack.com/howto/javascript/load-json-file-in-javascript/#use-the-fetch-function-to-load-json-files-in-javascript)

This function fetches the file from the path where it is saved and then returns the file as the response in the console.data. This function is only suitable for working in the web-based environment as the fetch API works only in that environment.

After reading the file, we parse the data using json() function and display it.

The below code demonstrates the above function.

fetch("./students.json")

.then(response => {

return response.json();

})

.then(jsondata => console.log(jsondata));

Output:

"Students" : [

{

"firstName":"Ram","lastName":"Sharma"

}

]

}

The **express.json()** function is a built-in middleware function in Express. It parses incoming requests with JSON payloads and is based on **body-parser**.

**Syntax:**

express.json( [options] )

**Parameters:** The options parameter have various property like inflate, limit, type, etc.

**Return Value:** It returns an Object.

**Installation of express module:**

1. You can visit the link to [Install express module](https://www.npmjs.com/package/express). You can install this package by using this command.

npm install express

1. After installing the express module, you can check your express version in command prompt using the command.

npm version express

1. After that, you can just create a folder and add a file for example, index.js. To run this file you need to run the following command.

node index.js

**Example 1:** **Filename: index.js**

|  |
| --- |
| var express = require('express');  var app = express();  var PORT = 3000;    app.use(express.json());    app.post('/', function (req, res) {      console.log(req.body.name)      res.end();  })    app.listen(PORT, function(err){      if (err) console.log(err);      console.log("Server listening on PORT", PORT);  }); |

**Steps to run the program:**

1. The project structure will look like this:
2. Make sure you have installed **express** module using the following command:

npm install express

3.Run index.js file using below command:

node index.js

**Output:**

Server listening on PORT 3000

1. Now make a POST request to *http://localhost:3000/* with header set to **‘content-type: application/json’** and body **{“name”:”GeeksforGeeks”}**, then you will see the following output on your console:
2. Server listening on PORT 3000
3. GeeksforGeeks

**Example 2:** **Filename: index.js**

|  |
| --- |
| var express = require('express');  var app = express();  var PORT = 3000;    // Without this middleware  // app.use(express.json());  app.post('/', function (req, res) {      console.log(req.body.name)      res.end();  })    app.listen(PORT, function(err){      if (err) console.log(err);      console.log("Server listening on PORT", PORT);  }); |

Run index.js file using below command:

node index.js