

# Asp.Net Core

Eng.Ahmed Kh. Shalayel

# Pre requirements

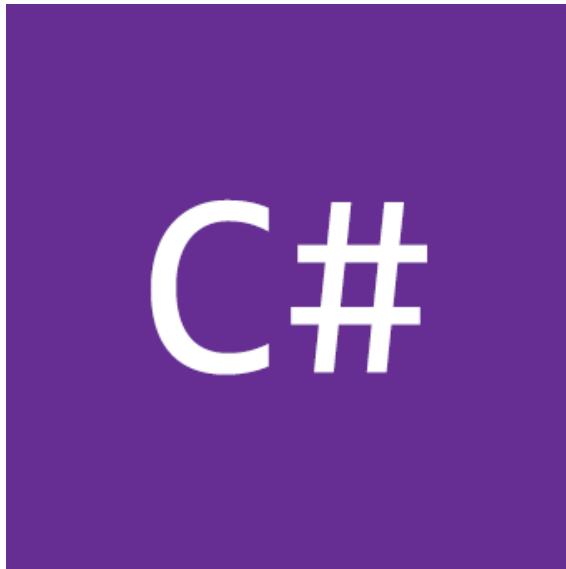
Basic knowledge Of :

1. Html
2. CSS
3. Java Scripts
4. Basic bootstrap
5. Any Programming Language

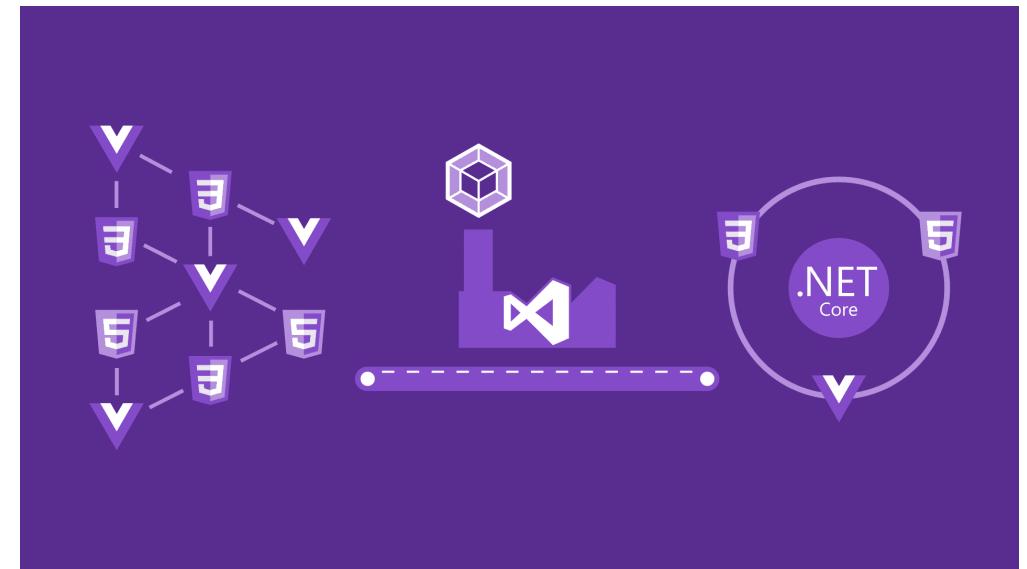


# Our Course

Language



Framework



# What I Need To Start !!

## Needs Of:

1. Visual Studio 2017
2. Sql Server 2017

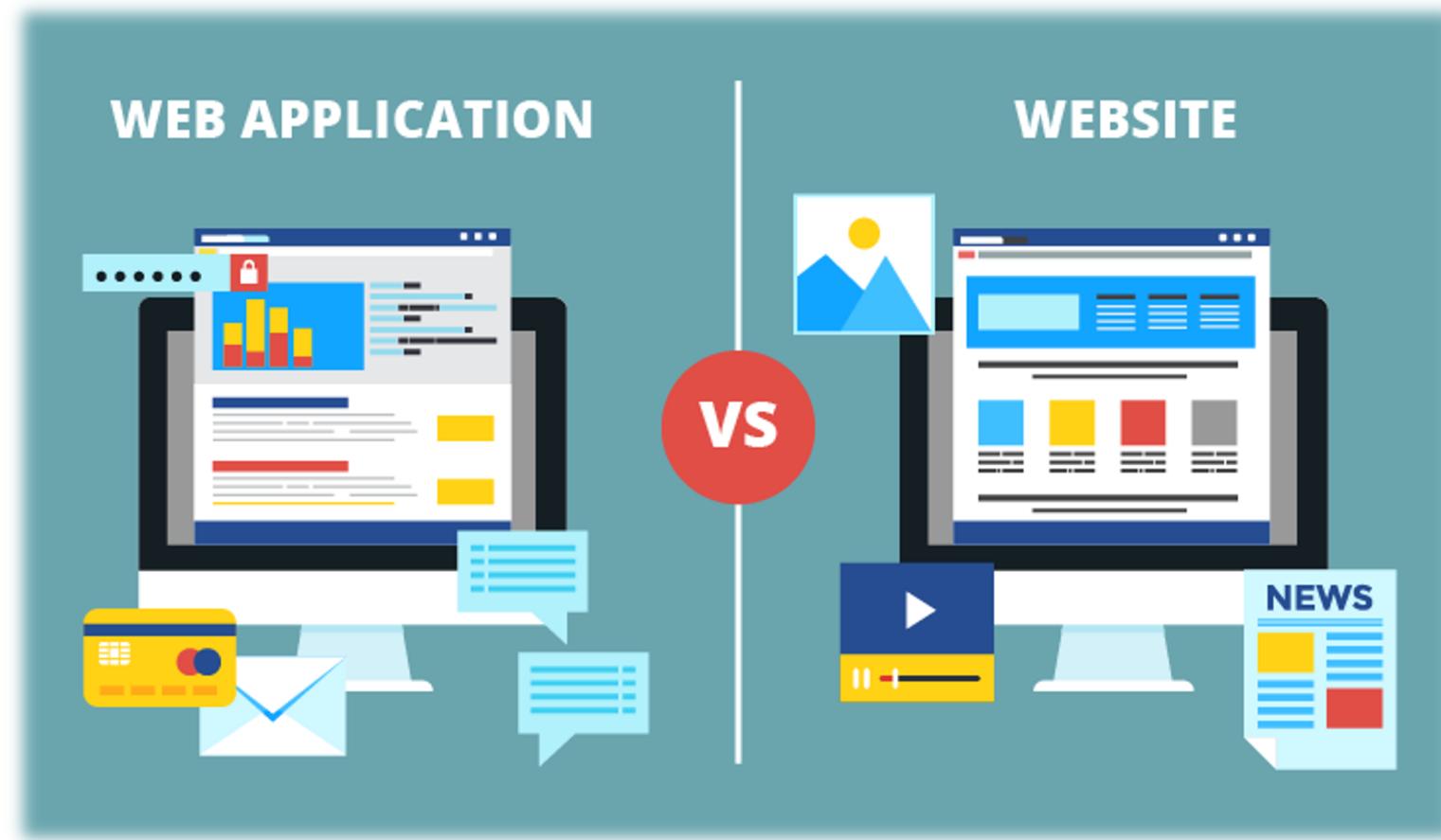


# Programme vs. application

- Computer Program
  - is a set of instructions that can be executed on a computer.
- Application
  - is software that directly helps a user perform tasks.

```
extract_number_and_incr(destination, source) int
    *destination; unsigned char **source; { extract_number_and_incr(destination, *source); *source += 2; } #ifndef EXTRACT_MACROS #undef EXTRACT_NUMBER_AND_INCR #define EXTRACT_NUMBER_AND_INCR(dest, src)\ extract_number_and_incr(&dest, &src) #endif /* not EXTRACT_MACROS */ #endif /* DEBUG */ /* If DEBUG is defined, Regex prints many voluminous messages about what it is doing (if the variable 'debug' is nonzero). If linked with the main program in 'iregex.c', you can enter patterns and strings interactively. And if linked with the main program in 'main.c' and the other test files, you can run the already-written tests. */ #ifdef DEBUG /* We use standard I/O for debugging. */ #include <stdio.h> /* It is useful to test things that "must" be true when debugging. */ #include <assert.h> static int debug = 0; #define DEBUG_STATEMENT(e) e#define DEBUG_PRINT1(x) if (debug) printf(x) #define DEBUG_PRINT2(x1, x2) if (debug) printf(x1, x2) #define DEBUG_PRINT3(x1, x2, x3) if (debug) printf(x1, x2, x3) #define DEBUG_PRINT4(x1, x2, x3, x4) if (debug) printf(x1, x2, x3, x4) #define DEBUG_PRINT_COMPILED_PATTERN(p, s, e) if (debug) print_partial_compiled_pattern(s, e) #define BUG_PRINT_DOUBLE_STRING(w, s1, sz1, s2, sz2) \ if (debug) print_double_string(w, s1, sz1, s2, sz2) extern void printchar() /* Print the fastmap in human-readable form. */ void print_fastmap(fastmap)
char *fastmap; { unsigned was_a_range = 0; unsigned i = 0; while (i < (1 << BYTESWIDTH)) { if (fastmap[i] & was_a_range = 0; printchar(i - 1); while (i < (1 << BYTESWIDTH) && fastmap[i]) { was_a_range = 1; i++; } (was_a_range) { printf("); printchar(i - 1); } } putchar ('\n'); /* Print a compiled pattern string in human-readable form, starting at the START pointer into it and ending just before the pointer END. */ void print_partial_compiled_pattern (start, end) unsigned char *start; unsigned char *end; { int mcnt, mcnt2; unsigned char *p = start; unsigned char *pend = end; if (start == NULL) { printf ("(\null)\n"); return; } /* Loop over pattern commands. */ while (p < pend) { switch ((re_opcode_t) *p++) { case no_op: printf ("no_op"); break; case exactn: mcnt = *p++; printf ("exactn%d", mcnt); do { putchar ('>'); printchar (*p++); } while (--mcnt); break; case start_memory: mcnt = *p++; printf ("start_memory%d/%d", mcnt, *p++); break; case stop_memory: mcnt = *p++; printf ("stop_memory%d/%d", mcnt, *p++); break; case duplicate: printf ("duplicate%d", *p++); break; case anychar: printf ("anychar"); break; case charset: case charset_not: { register int c; printf ("charset%os", (re_opcode_t) *(p - 1) == charset_not ? ".not": ""); assert (p + *p < pend); for (c = 0; c < *p; c++) { unsigned bit; unsigned char map_byte = p[1 + c]; putchar ('>'); for (bit = 0; bit < BYTESWIDTH; bit++) if (map_byte & (1 << bit)) printchar (c * BYTESWIDTH + bit); } p += 1 + *p; break; } case begin_line: printf ("beginline"); break; case endline: printf ("endline"); break; case on_failure_jump: extract_number_and_incr (&mcnt, &p); printf ("on_failure_jump/0/%d", mcnt); break; case on_failure_keep_string_jump: extract_number_and_incr (&mcnt, &p); printf ("on_failure_keep_string_jump/0/%d", mcnt); break; case dummy_failure_jump: extract_number_and_incr (&mcnt, &p); printf ("dummy_failure_jump/0/%d", mcnt); break; case push_dummy_failure: printf ("push_dummy_failure"); break; case maybe_pop_jump: extract_number_and_incr (&mcnt, &p); printf ("maybe_pop_jump/0/%d", mcnt); break; case pop_failure_jump: extract_number_and_incr (&mcnt, &p); printf ("pop_failure_jump/0/%d", mcnt); break; case jump_past_alt: extract_number_and_incr (&mcnt, &p); printf ("jump_past_alt/0/%d", mcnt); break; } }
```

# Web application vs. website

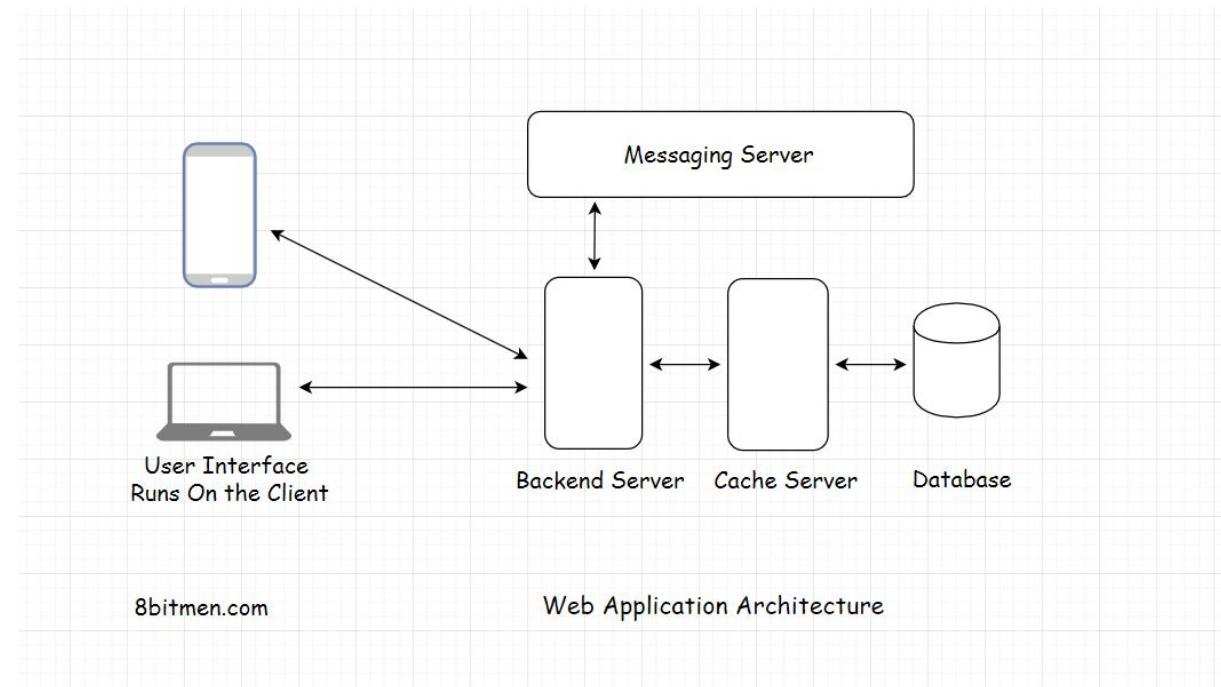


# Web Application & Software Architecture

Eng.Ahmed Kh. Shalayel

# What Is Web Architecture?

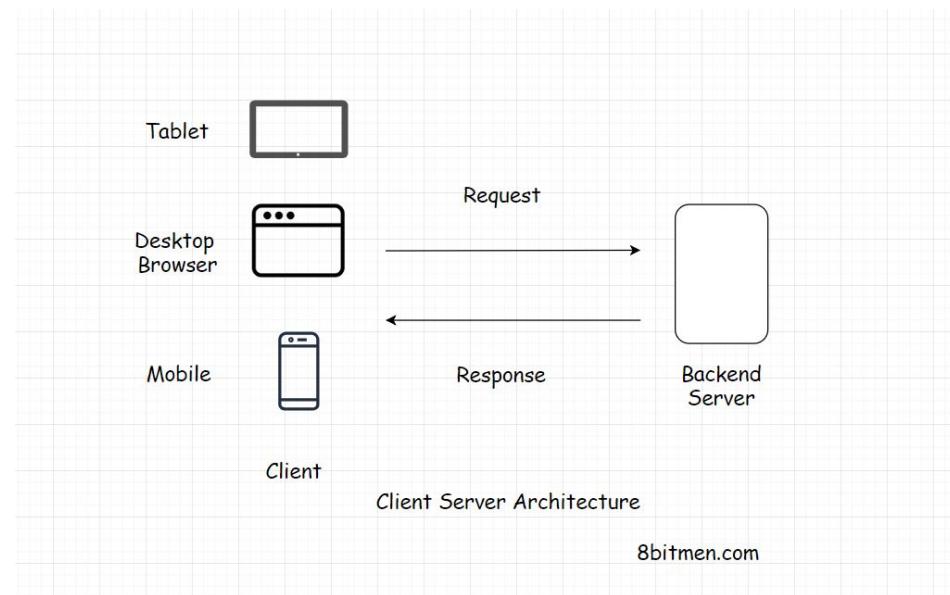
Web architecture involves multiple components like *database*, *message queue*, *cache*, *user interface* & all running in conjunction with each other to form an online service.



# Client Server Architecture

*Client-Server* architecture is the fundamental building block of the web.

- The architecture works on a *request-response* model.
- The *client* sends the request to the *server* for information & the *server* responds with it.
- Every website you browse, be it a Wordpress blog or a web application like *Facebook*, *Twitter* or your banking app is built on the client-server architecture.



# Client

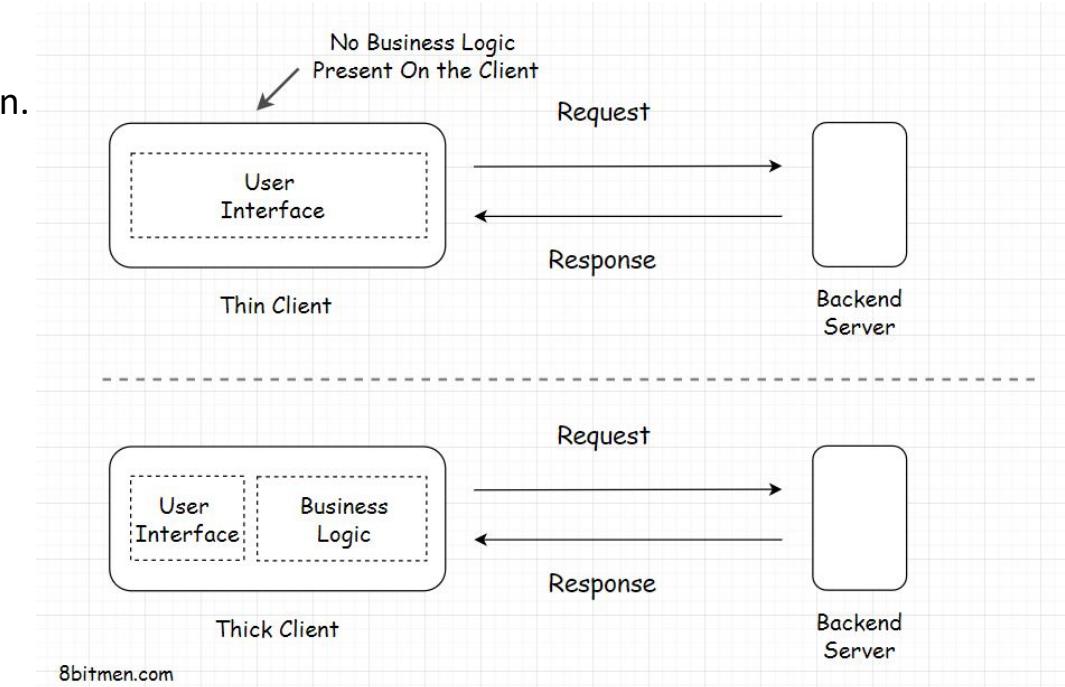
- The *client* holds our *user interface*.
- The user interface is the presentation part of the application.
- It's written in *HTML, JavaScript, CSS* and is responsible for the look & feel of the application.
- The user interface runs on the client.
- The client can be a mobile app, a desktop or a tablet like an *iPad*. It can also be a web-based console, running commands to interact with the backend server.



# Types of Client

## Thin Client :

- Thin Client is the client which holds just the user interface of the application.
- It has no business logic of any sort.
- For every action, the client sends a request to the backend server.



## Thick Client (sometimes also called the Fat client):

- The thick client holds all or some part of the business logic.

# Server & Server-Side Rendering

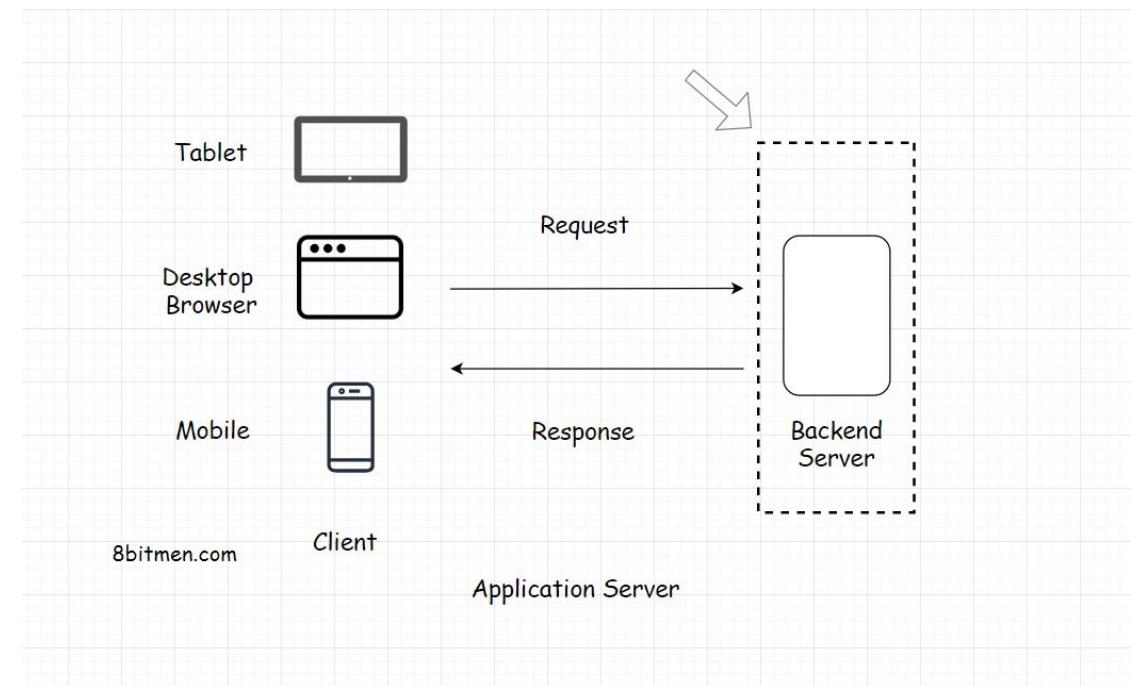
## Server

The primary task of a web server is to receive the requests from the client & provide the response after executing the business logic based on the request parameters received from the client.

Every service, running online, needs a server to run. Servers running web applications are commonly known as the *application servers*.

## Server-Side Rendering

Often the developers use a server to render the user interface on the backend & then send the rendered data to the client.



# Communication Between the Client & the Server

## **Request-Response Model**

- The client & the server have a request-response model.
- The client sends the request & the server responds with the data.
- If there is no request, there is no response

# HTTP Protocol

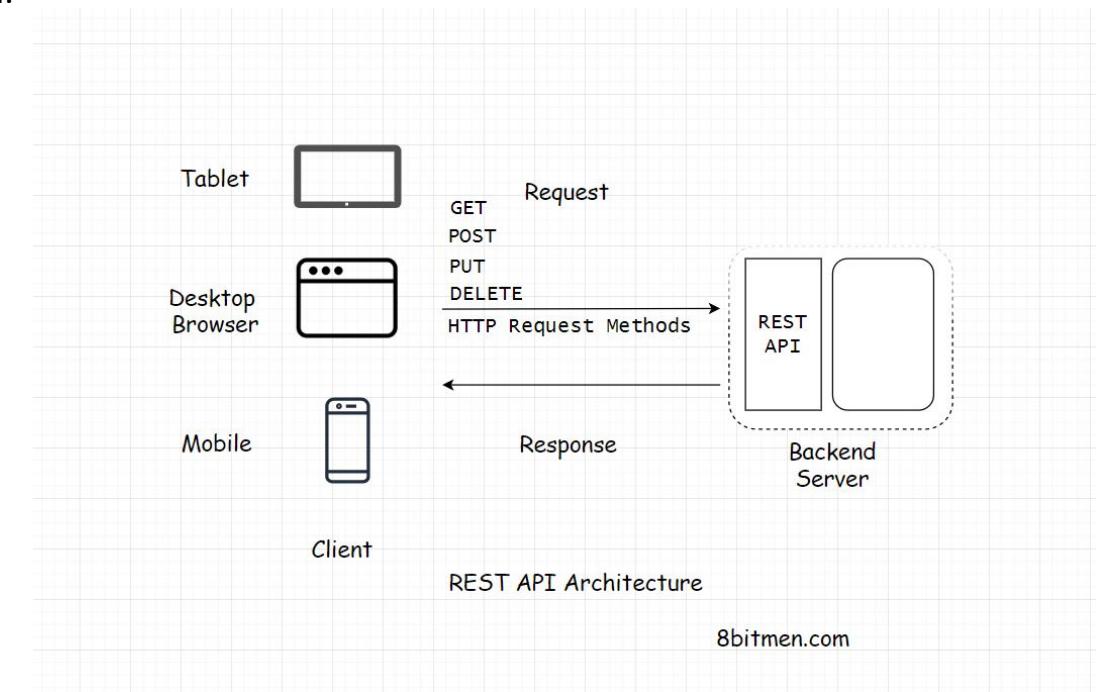
- The entire communication happens over the HTTP protocol.
- It is the protocol for data exchange over the World Wide Web.
- HTTP protocol is a request-response protocol that defines how information is transmitted across the web.

# REST API & API Endpoints

Every client has to hit a *REST end-point* to fetch the data from the backend.

The backend application code has a *REST-API* implemented which acts as an interface to the outside world requests.

Every request be it from the client written by the business or the third-party developers which consume our data have to hit the REST-endpoints to fetch the data.



# WHAT IS REST?

REST stands for Representational State Transfer.

It's a software architectural style for implementing web services.

Web services implemented using the REST architectural style are known as the RESTful Web services.

# REST API

A REST API is an API implementation that adheres to the REST architectural constraints.  
It acts as an interface.

The communication between the client & the server happens over HTTP.

A REST API takes advantage of the HTTP methodologies to establish communication between the client and the server.

REST also enables servers to cache the response that improves the performance of the application.

# REST End Point

An API/REST/Backend endpoint means the URL of a service.

For example, **<https://myservice.com/users/{username}>** is a backend endpoint for fetching the user details of a particular user from the service.

# Real World Example Of Using A REST API

For instance, let's say we want to write an application which would keep track of the birthdays of all our Facebook friends & send us a reminder a couple of days before the event date.

To implement this, the first step would be to get the data on the birthdays of all our Facebook friends.

We would write a client which would hit the Facebook Social Graph API which is a REST-API to get the data & then run our business logic on the data.

# HTTP Pull 01

As I stated earlier, for every response, there has to be a request first.

The client sends the request & the server responds with the data.

This is the default mode of HTTP communication, called the HTTP PULL mechanism.

The client pulls the data from the server whenever it requires.

Client keeps doing it over and over to fetch the updated data.

# HTTP Pull 02

An important thing to note here is that every request to the server and the response to it consumes bandwidth.

Every hit on the server costs the business money & adds more load on the server.

What if there is no updated data available on the server, every time the client sends a request?

The client doesn't know that, so naturally, it would keep sending the requests to the server over and over. This is not ideal & a waste of resources.

Excessive pulls by the clients have the potential to bring down the server.

# HTTP PUSH 01

The HTTP PUSH based mechanism.

In this mechanism, the client sends the request for particular information to the server, just for the first time, & after that the server keeps pushing the new updates to the client whenever they are available.

The client doesn't have to worry about sending requests to the server, for data, every now & then.

This saves a lot of network bandwidth & cuts down the load on the server by notches.

This is also known as a *Callback*. Client phones the server for information The server responds, Hey!! I don't have the information right now but I'll call you back whenever it is available.

# HTTP PUSH 02

There are multiple technologies involved in the HTTP Push based mechanism such as:

- Ajax Long polling
- Web Sockets
- HTML5 Event Source
- Message Queues
- Streaming over HTTP

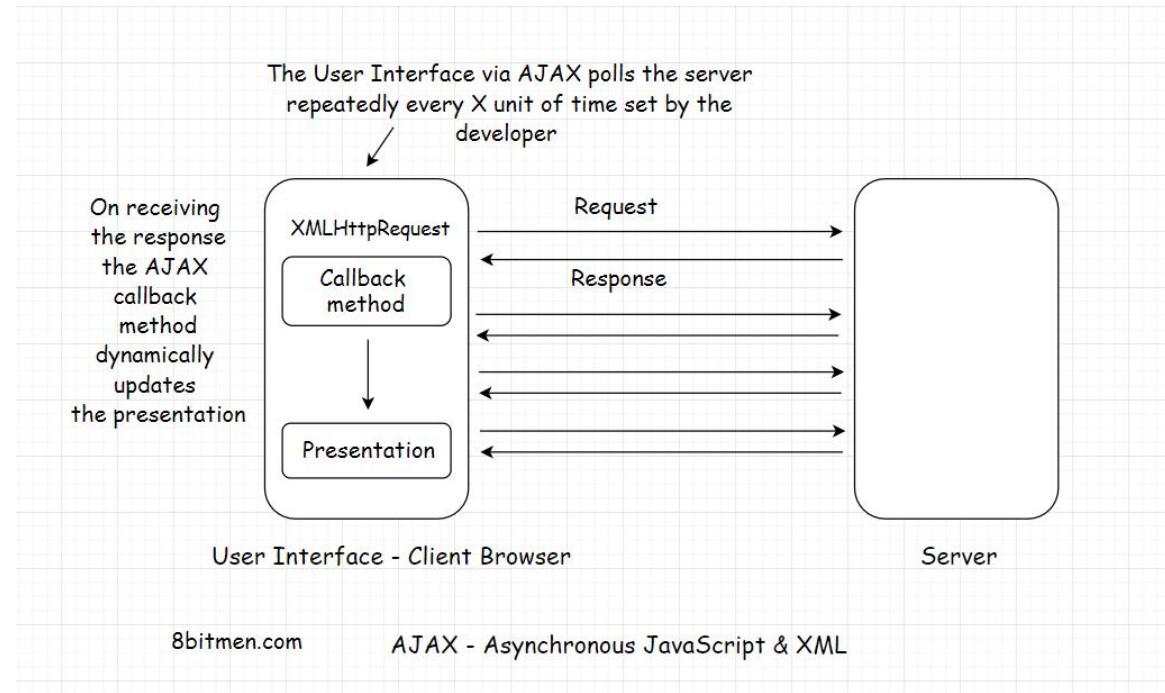
# HTTP Pull - Polling with Ajax

There are two ways of pulling/fetching data from the server.

- The first is sending an HTTP GET request to the server manually by triggering an event, like by clicking a button or any other element on the web page.
- The other is fetching data dynamically at regular intervals by using AJAX without any human intervention

# AJAX – Asynchronous JavaScript & XML

AJAX stands for asynchronous JavaScript & XML. The name says it all, it is used for adding asynchronous behaviour to the web page.

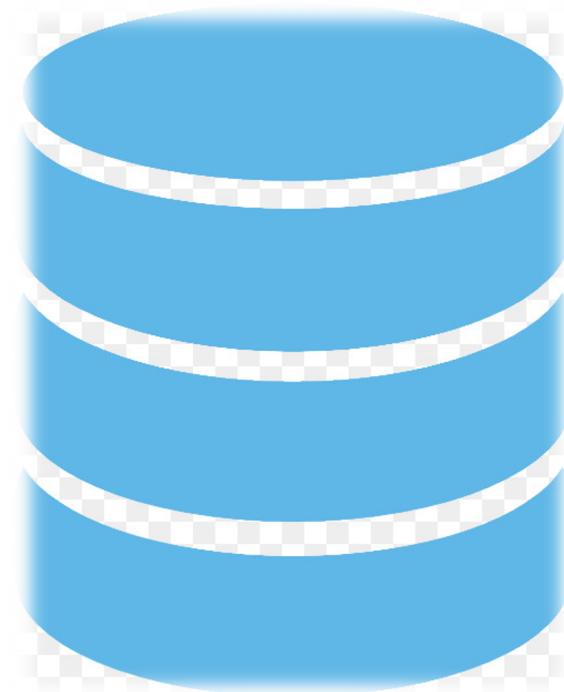


# Ajax Example



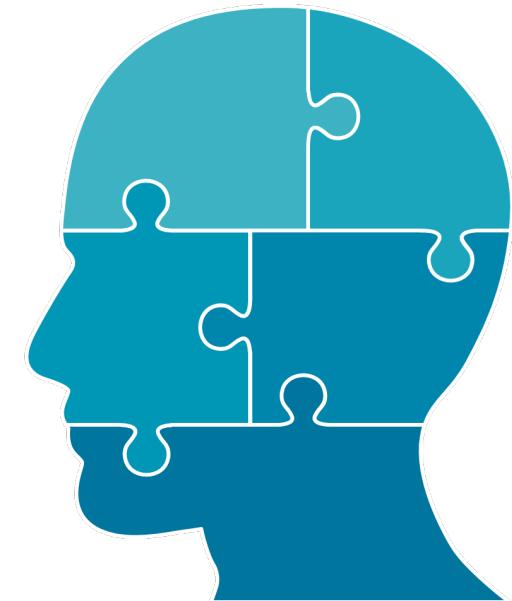
# mODEL

- **Models** : Classes that represent the data of the app .



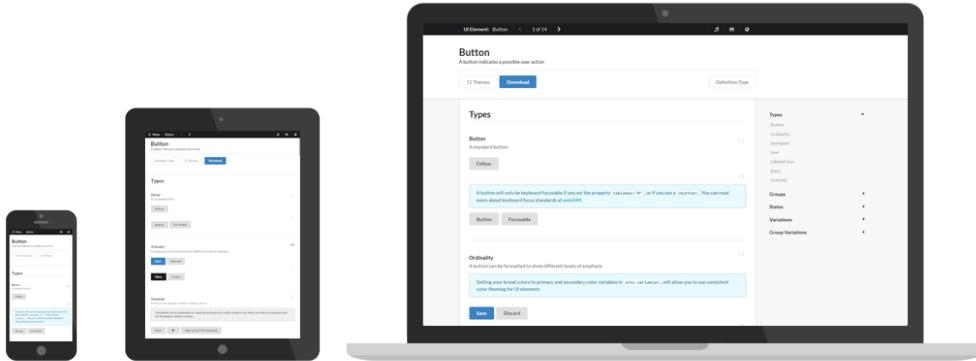
# Controller

- **Controllers:** Classes that handle browser requests.

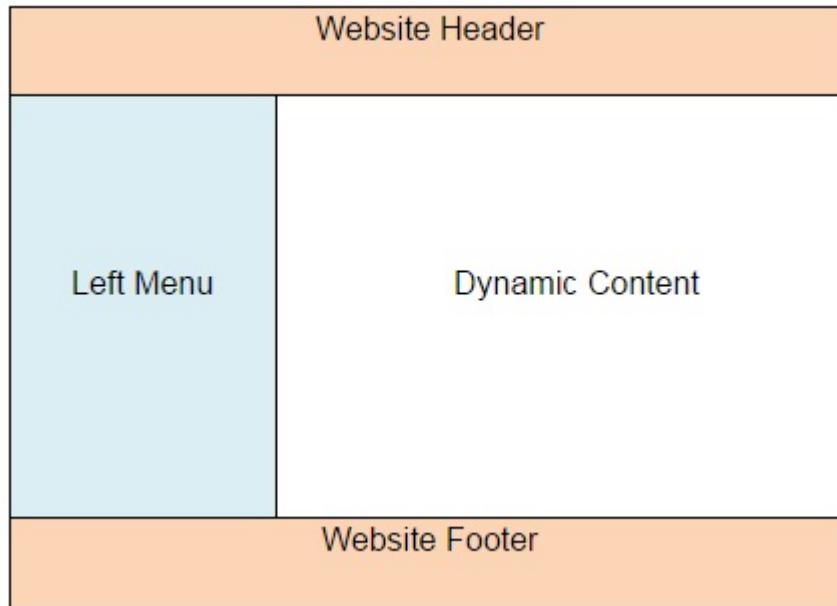


# View

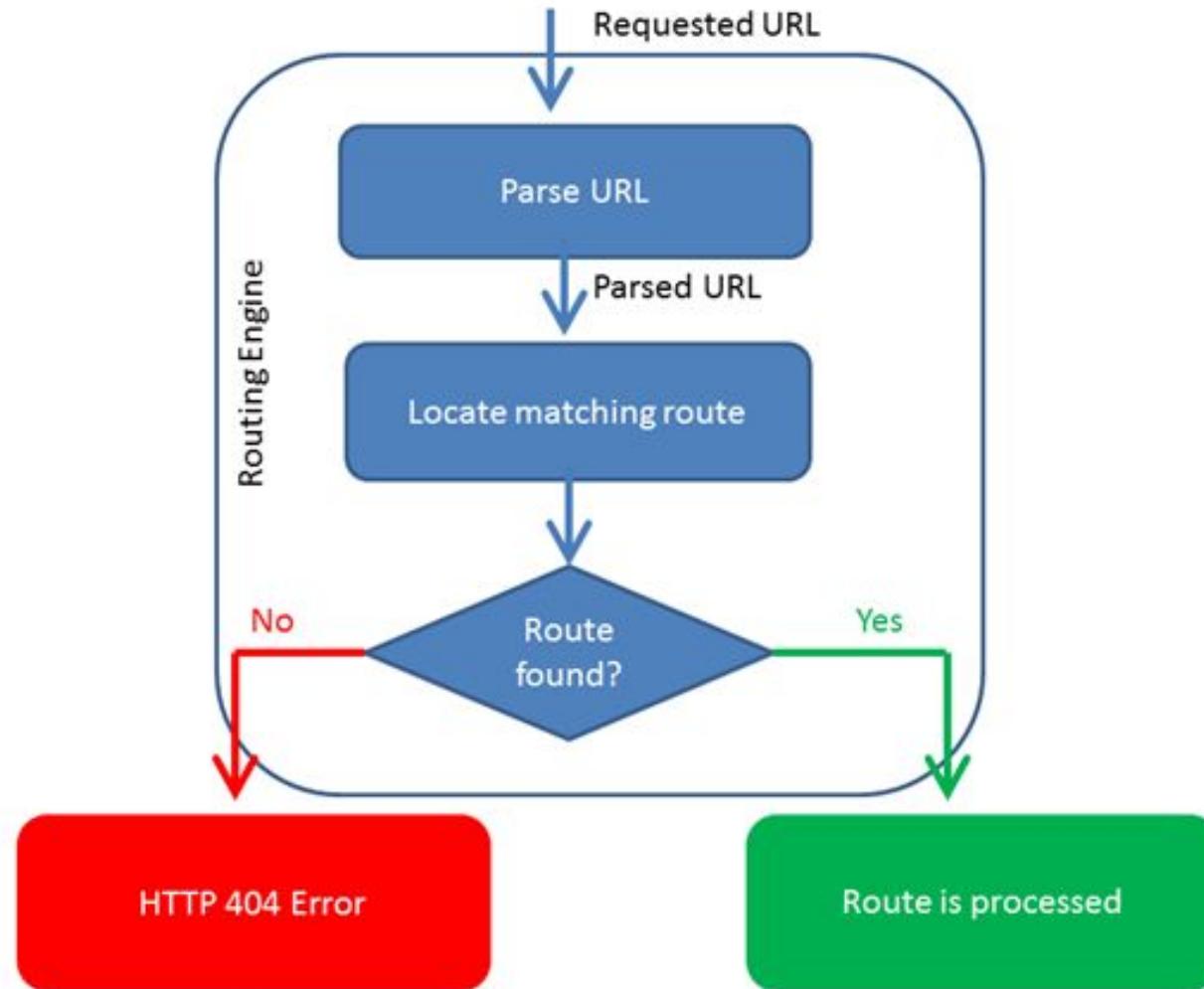
- **Views:** Views are the components that display the app's user interface (UI).



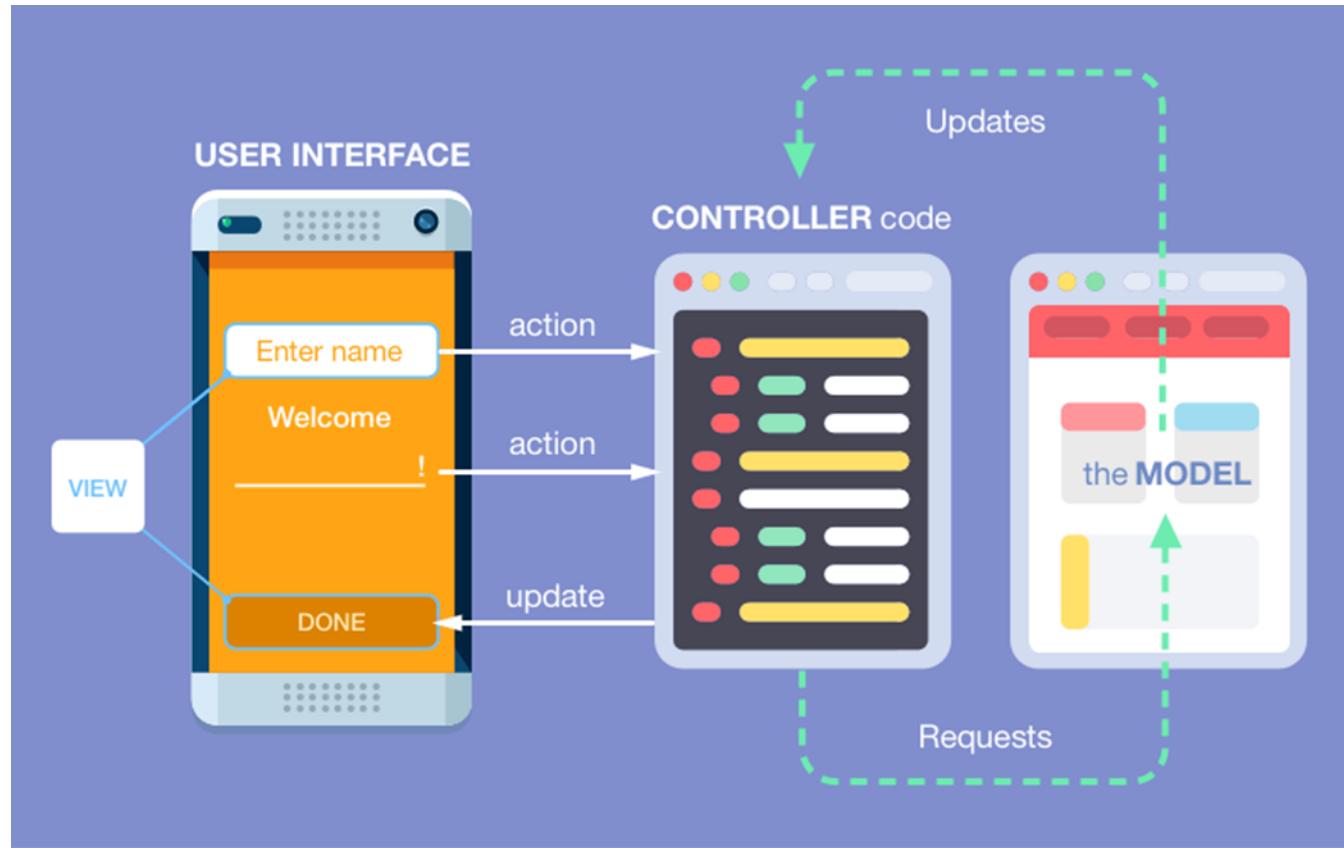
# Layout



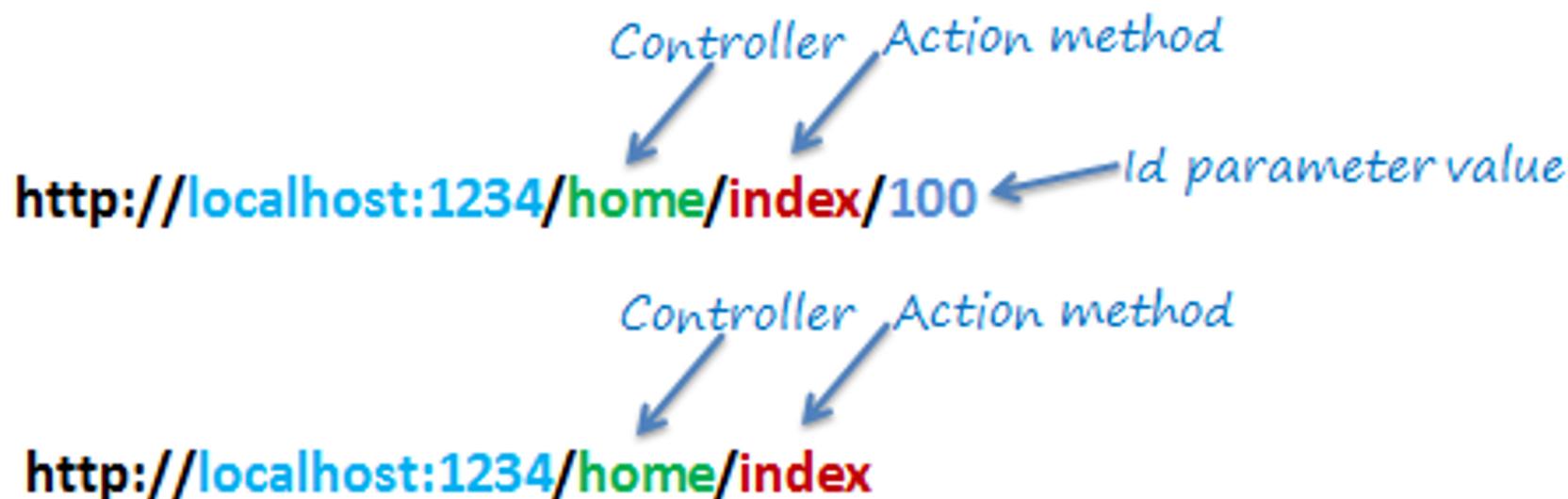
# Route



# Route



# Url in Mvc





LET'S START A JOURNEY

