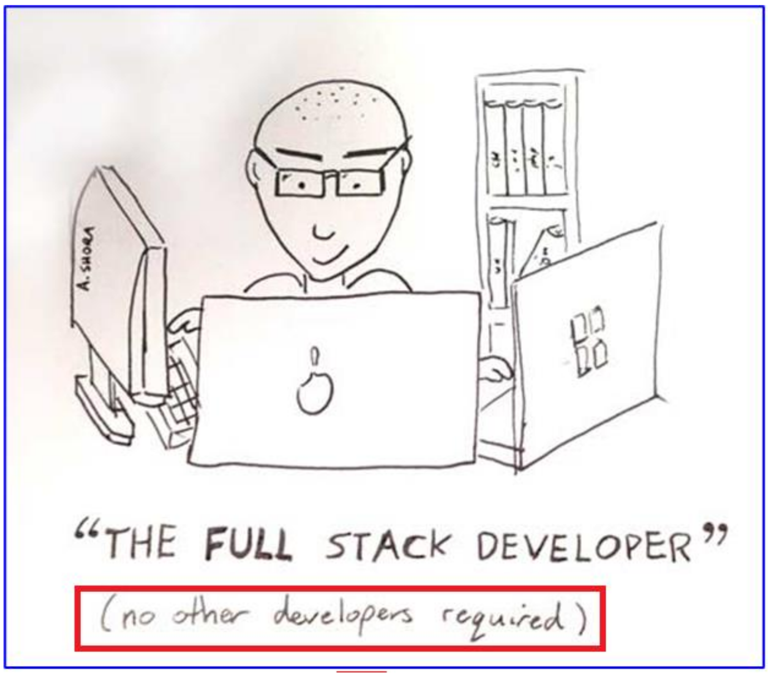
UNIT-1

Full Stack Development Basics

What is Full Stack Development?



Full stack development is the process of designing, creating, testing, and deploying a complete web application from start to finish.

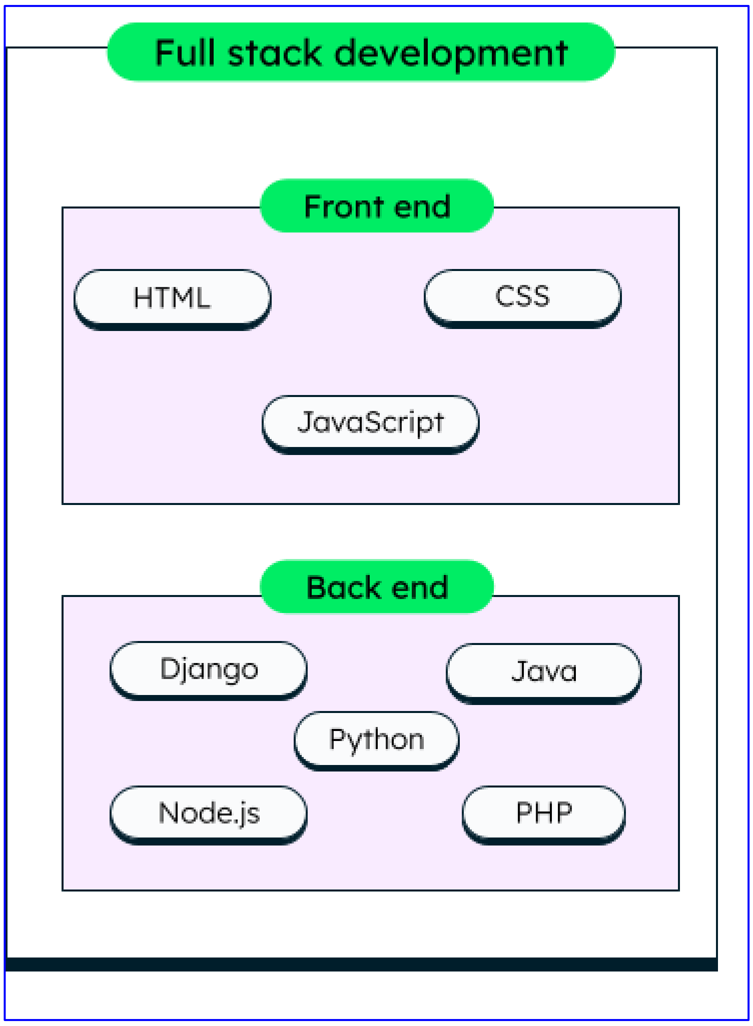
It involves working with various technologies and tools, including front-end web development, back-end web development, and database development.

Full stack development is a term used to describe a software engineer or developer who works with both the front and back end of a website or application. A full-stack developer is comfortable working with front-end and back-end technologies that power a website or application.

Full-stack developers are often responsible for the entire web application development process, from start to finish, which means they must have a strong understanding of all the technologies and tools involved in web development.

They also need to work effectively with others on a team, as web development is typically a collaborative process. Most full-stack developers have a firm foundation in web development technologies, such as HTML, CSS, and JavaScript. They also have experience with server-side technologies such as PHP, Ruby on Rails, and Node.js. In addition to their technical skills, full-stack developers also deeply understand how the various parts of a website or application work together.

Full stack development refers to the end-to-end application software development, including the front end and back end. The front end consists of the user interface, and the back end takes care of the business logic and application workflows.

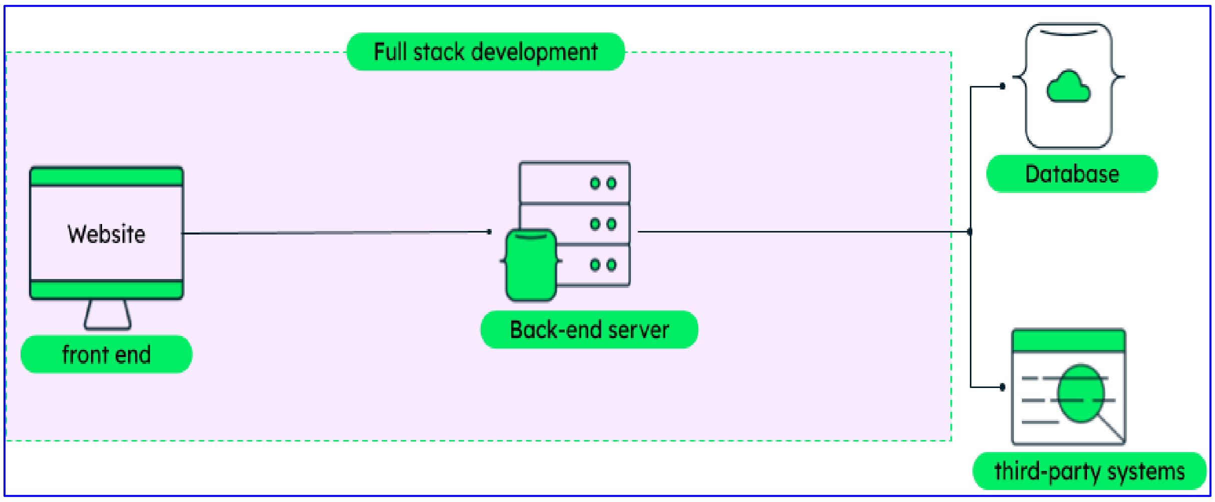


*[The main components of a full stack development are the **front**-end, **back**-end and **database**]*

Consider a retail website. Users can browse or purchase specific items, delete or add items in cart, change their profile, and do many other things. All these actions require a front-end user interface (UI), as well as some business logic, written in the back-end.

- The website UI can be built using various, *front-end technologies* like HTML, CSS, Javascript.
- The back end is written in *programming languages* like Java or Python. Further, a good web application would need scalability, event handling, and routing, which are usually handled by *libraries and frameworks* like SpringBoot or Django.
- The back end also consists of logic that can connect the application to other services and databases. For example, all the user and transaction data is stored in a database through specific *drivers* handled on the back end.

A **full stack developer** is one who can single-handedly implement both the front-end and back-end workflows, like placing the order or changing the user profile.



[Demonstration of full stack development in an end-to-end workflow]

What is a full stack developer and what do they do?

Full stack developers must have knowledge of an entire technology stack, i.e., the set of technologies that are used to build an end-to-end application quickly and efficiently. For example, if they want to build an application using the **MEAN** stack, they should know how to work with **M**ongoDB, **E**xpress, **A**ngular and **N**ode.

(MongoDB is a NoSQL database, Express.js is a web application framework for Node.js, Angular is a front-end web application framework developed by Google, Node.js is a server-side JavaScript)

Full stack developers should be able to judge whether the selected technologies are the right choice for their project during the early phases. Some **responsibilities of a full stack developer are to:**

- Help in choosing the right technologies for the project development and testing both on the front end and the back end.
- Write clean code across the stack by following the best practices of the tools used.
- Be up to date with the latest technologies and tools to make the best technology usage decisions.

What languages do full stack developers use?

Full stack developers are free to use any set of languages that are compatible with each other and the overall application framework. JavaScript is a popular language often used by full-stack developers as it's one of the very few languages that can be used both on the front end and back end. Companies will most likely hire a full stack developer for smaller or medium-size projects. Some popular languages are:

- **Front end:** HTML, CSS, JavaScript.
- **Back end:** Python, Java, R, Ruby, Node.js, PHP.

It's also a popular and convenient practice to use full technology stacks like MEAN stack, MERN stack, Ruby on Rails, and LAMP for faster and more efficient development, and an easier learning curve.

Front end vs back end vs full stack

Applications that require higher scalability and more complex workflows require broader skill sets and collaboration across teams. For example, the front end may be handled by the UI team, and the back end by another team. In some organizations, individuals will be required to work on both the front-end and back-end implementation of a feature. This is where full stack developers would come into play.

Front-end developers

These developers **handle** the UI of a web application (or website)—for example, visual effects, frames, navigation, and forms. **They** focus mainly on user experience and use HTML, CSS, and JavaScript as programming languages.

Back-end developers

They **deal** with the business logic, security, performance, scalability, and handling request-response of the application. **They** create or use frameworks to design the core application workflows and use technologies like JavaScript, Python, Java, and .NET.

Full stack developers

They are responsible for coding end-to-end workflows by using both front-end and back-end technologies. MERN stack and MEAN stack are examples of JavaScript-based technology stacks that full stack developers can use to build end-to-end applications.

Is a software engineer the same as a full-stack developer?

Software engineering is a general term, **whereas** full stack development is a part of software engineering that requires knowledge of front-end and back-end technologies to create end-to-end web applications.

A full stack developer works on a complete technology stack which generally includes a back end and front end. **A software engineer** is mostly an individual contributor who can work on a specific module or technology at a time.

Advantages and Disadvantages of Full Stack Development

Advantages:

There are many advantages of hiring full stack developers for web application development:

- Complete ownership and understanding of the project
- Saves both project time and cost, and enhances productivity
- Faster bug fixing due to knowledge of complete system
- Easy knowledge transfer to other team members
- Better division of work amongst team members
- You can build a professional website from scratches for your employer.

- You can be a competent member in cross-functional Agile teams where you can perform both front-end and back-end development. As Agile project management is fast becoming the norm for web application development and UI/UX is gaining in importance for the success or failure of any web application, you will be seen as a valuable team member contributing to the success of projects.
- You can start your own website design and development business.
- You can build your own websites and monetize them through AdSense, affiliate marketing, selling your own products and more. The best of this is the flexibility of earning extra money alongside your regular job. There are many great examples of web developers turning this business into a career and quitting their regular jobs.
- as an example, you can utilize your knowledge in web development to help others to select the most suitable website hosting plan with an earning

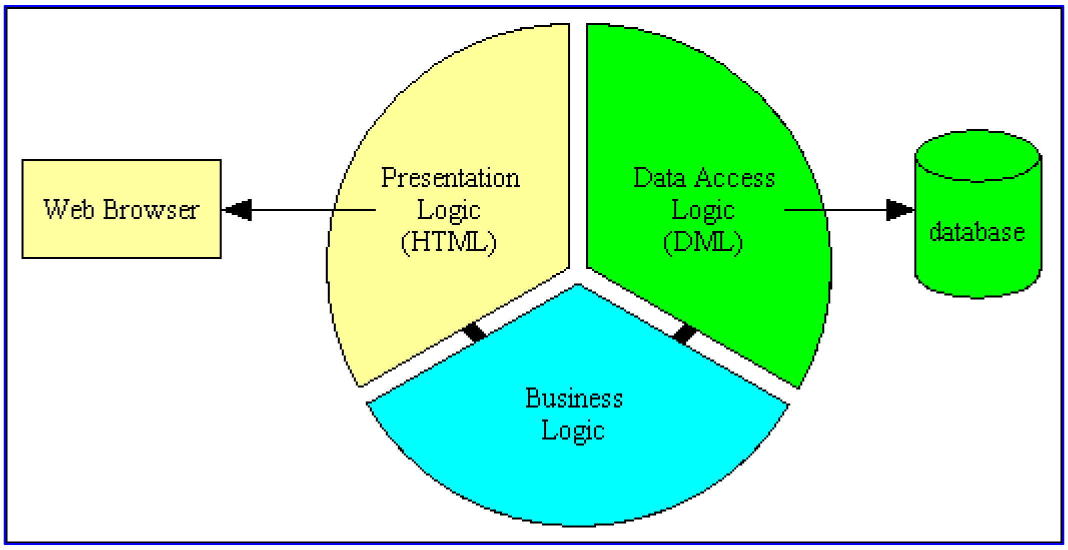
Disadvantages:

There are a few potential disadvantages to full-stack development,

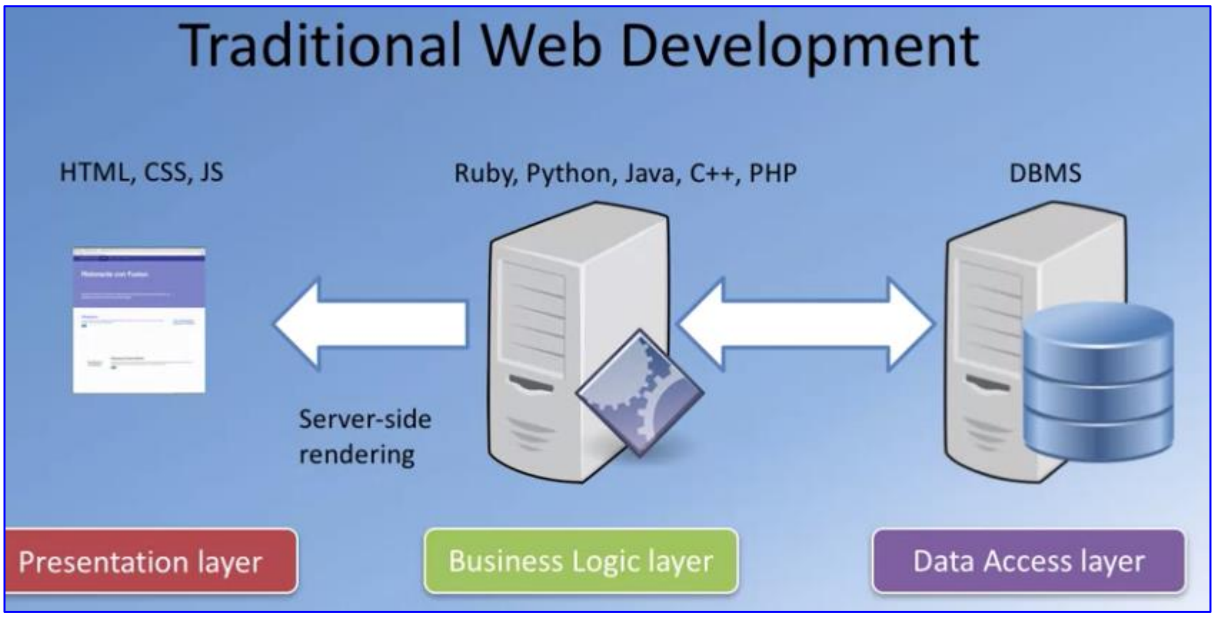
- The solution chosen can be wrong for the project
- The solution chosen can be dependent on developer skills
- The solution can generate a key person risk
- Being a full stack developer is increasingly complex

3-Tier Applications:

This is where the code for each area of responsibility can be cleanly split away from the others, as shown in below figure :



[3 Tier Architecture]



Note here that the presentation layer has no direct communication with the data access layer - it can only talk to the business layer.

Note also that you should not infer from this diagram that the entire application can be built with a single component in each of these three layers.

There should several choices as follows:

- There should be a separate component in the Presentation layer for each user transaction.
- There should be a separate component in the Business layer for each business entity (database table).

- There should be a separate component in the Data Access layer for each supported DBMS.

With this structure it is easy to replace the component in one layer with another component without having to make any changes to any component in the other layers.

- You can change the UI component so that you can switch between a variety of different output formats, such as HTML, PDF or CSV.
- You can change the data access component so that you can switch between a variety of database engines, such as MySQL, Oracle or SQL Server.
- You can change the business/domain component to update the business rules.

This structure also provides more reusability as a single component in the Business layer can be shared by several components in the Presentation layer. This means that business logic can be defined in one place yet shared by multiple components.

The Rules of the 3 Tier Architecture

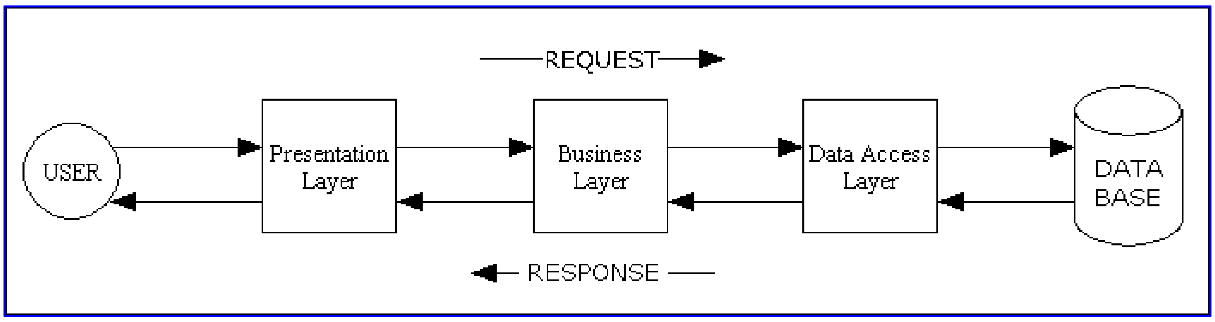
It is simply not good enough to split the code for an application into 3 parts and call it "3 Tier" if the code within each tier does not behave in a certain way.

There are rules to be followed, but these rules are pretty straightforward.

- The code for each layer must be contained within separate files which can be maintained separately, possibly by separate teams.
- Each layer may only contain code which belongs in that layer. Thus business logic can only reside in the Business layer, presentation logic in the Presentation layer, and data access logic in the Data Access layer.

- The **Presentation layer** can only receive requests from, and return responses to, an outside agent. This is usually a person, but may be another piece of software.
- The **Presentation layer** can only send requests to, and receive responses from, the Business layer. It cannot have direct access to either the database or the Data Access layer.
- The **Business layer** can only receive requests from, and return responses to, the Presentation layer.
- The **Business layer** can only send requests to, and receive responses from, the Data Access layer. It cannot access the database directly.
- The **Data Access** layer can only receive requests from, and return responses to, the Business layer. It cannot issue requests to anything other than the DBMS which it supports.
- Each layer should be totally unaware of the inner workings of the other layers. The Business layer, for example, must be database-agnostic and not know or care about the inner workings of the Data Access object. It must also be presentation-agnostic and not know or care how its data will be handled. It should not process its data differently based on what the receiving component will do with that data. The presentation layer may take the data and construct an HTML document, a PDF document, a CSV file, or process it in some other way, but that should be totally irrelevant to the Business layer.

This cycle of requests and their associated responses can be shown in the form of a simple diagram, as shown in below figure:



[Requests and Responses in the 3 Tier Architecture]

If you look carefully at those layers you should see that each one requires different sets of skills:

- The Presentation layer requires skills such as HTML, CSS and possibly JavaScript, plus UI design.
- The Business layer requires skills in a programming language so that business rules can be processed by a computer.
- The Data Access layer requires SQL skills in the form of Data Definition Language (DDL) and Data Manipulation Language (DML), plus database design.

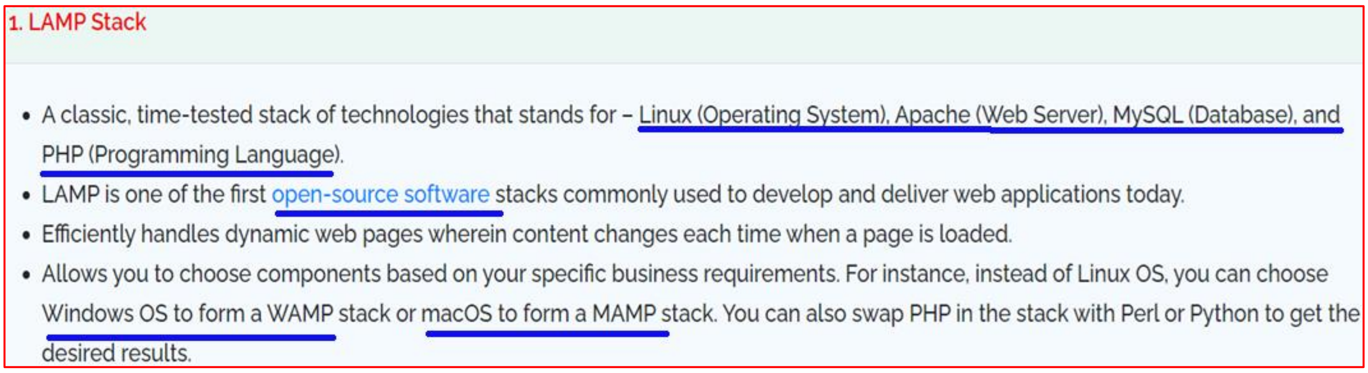
Although it is possible for a single person to have all of the above skills, such people are quite rare. In large organisations with large software applications this splitting of an application into separate layers makes it possible for each layer to be developed and maintained by different teams with the relevant specialist skills.

Various Stacks usage for web application development:

Popular Web Development Stacks

Different stacks exist to handle different tasks related to web application development and operations.

Following are some of the top stacks for front-end, back-end, and full-stack web development:

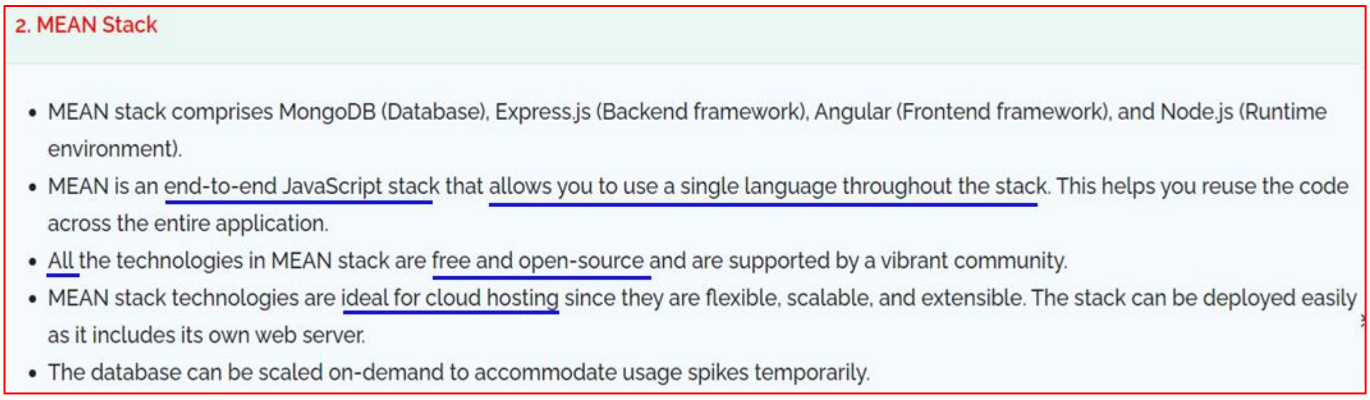


1. LAMP Stack

- A classic, time-tested stack of technologies that stands for – Linux (Operating System), Apache (Web Server), MySQL (Database), and PHP (Programming Language).
- LAMP is one of the first open-source software stacks commonly used to develop and deliver web applications today.
- Efficiently handles dynamic web pages wherein content changes each time when a page is loaded.
- Allows you to choose components based on your specific business requirements. For instance, instead of Linux OS, you can choose Windows OS to form a WAMP stack or macOS to form a MAMP stack. You can also swap PHP in the stack with Perl or Python to get the desired results.

Companies that use LAMP:

- Wikipedia, Yahoo, Etsy, Shopify, WordPress, Magento

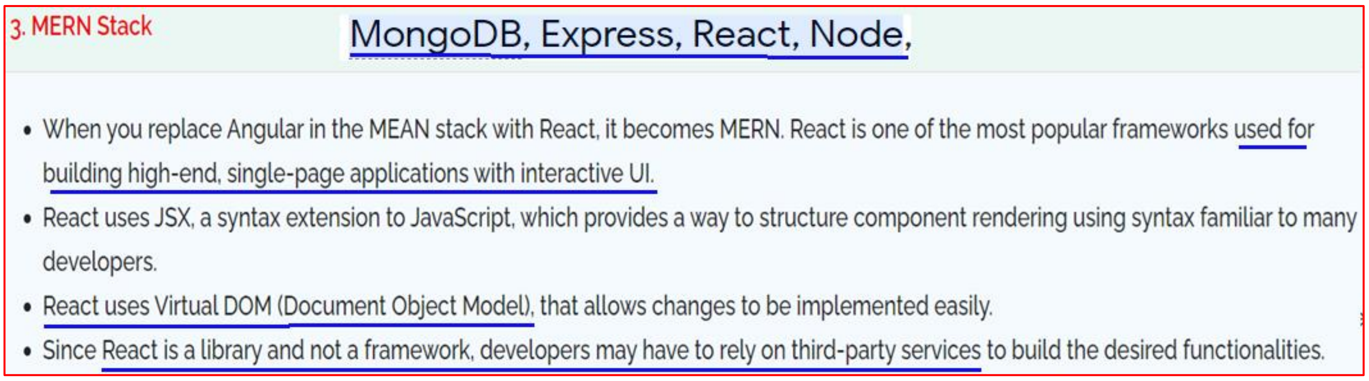


2. MEAN Stack

- MEAN stack comprises MongoDB (Database), Express.js (Backend framework), Angular (Frontend framework), and Node.js (Runtime environment).
- MEAN is an end-to-end JavaScript stack that allows you to use a single language throughout the stack. This helps you reuse the code across the entire application.
- All the technologies in MEAN stack are free and open-source and are supported by a vibrant community.
- MEAN stack technologies are ideal for cloud hosting since they are flexible, scalable, and extensible. The stack can be deployed easily as it includes its own web server.
- The database can be scaled on-demand to accommodate usage spikes temporarily.

Companies that use MEAN:

- Google, Microsoft, IBM, Amazon, Uber, PayPal, LinkedIn

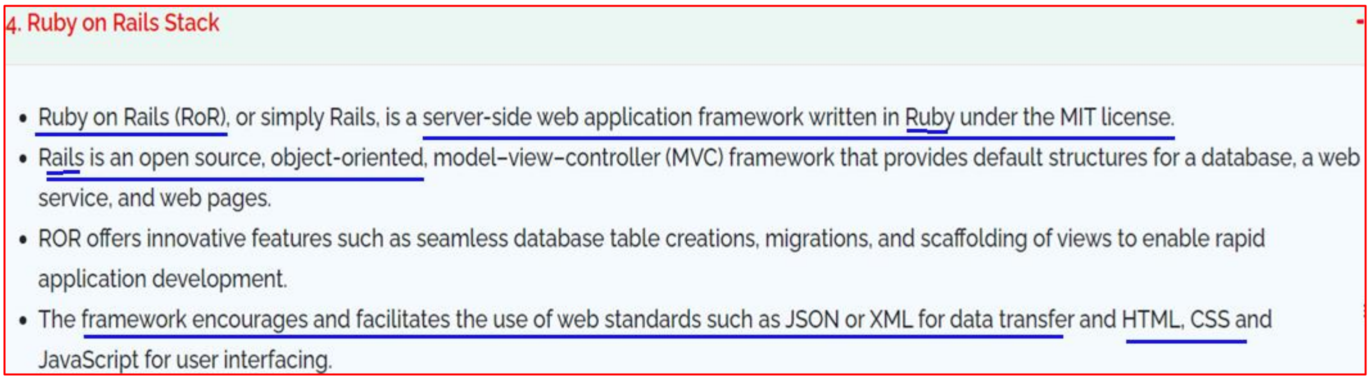


3. MERN Stack MongoDB, Express, React, Node,

- When you replace Angular in the MEAN stack with React, it becomes MERN. React is one of the most popular frameworks used for building high-end, single-page applications with interactive UI.
- React uses JSX, a syntax extension to JavaScript, which provides a way to structure component rendering using syntax familiar to many developers.
- React uses Virtual DOM (Document Object Model), that allows changes to be implemented easily.
- Since React is a library and not a framework, developers may have to rely on third-party services to build the desired functionalities.

Companies that use MERN:

- Facebook (Meta), Tesla, GoDaddy, Walmart, Shutterfly, Under Armour, Accenture, Philips, Airbnb, Netflix, Mozilla

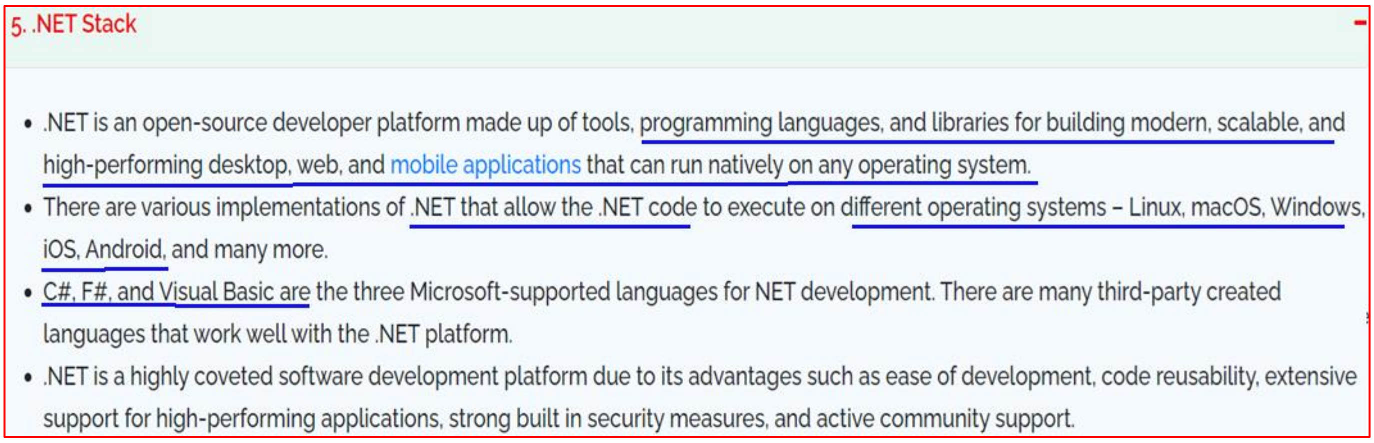


4. Ruby on Rails Stack

- Ruby on Rails (RoR), or simply Rails, is a server-side web application framework written in Ruby under the MIT license.
- Rails is an open source, object-oriented, model–view–controller (MVC) framework that provides default structures for a database, a web service, and web pages.
- ROR offers innovative features such as seamless database table creations, migrations, and scaffolding of views to enable rapid application development.
- The framework encourages and facilitates the use of web standards such as JSON or XML for data transfer and HTML, CSS and JavaScript for user interfacing.

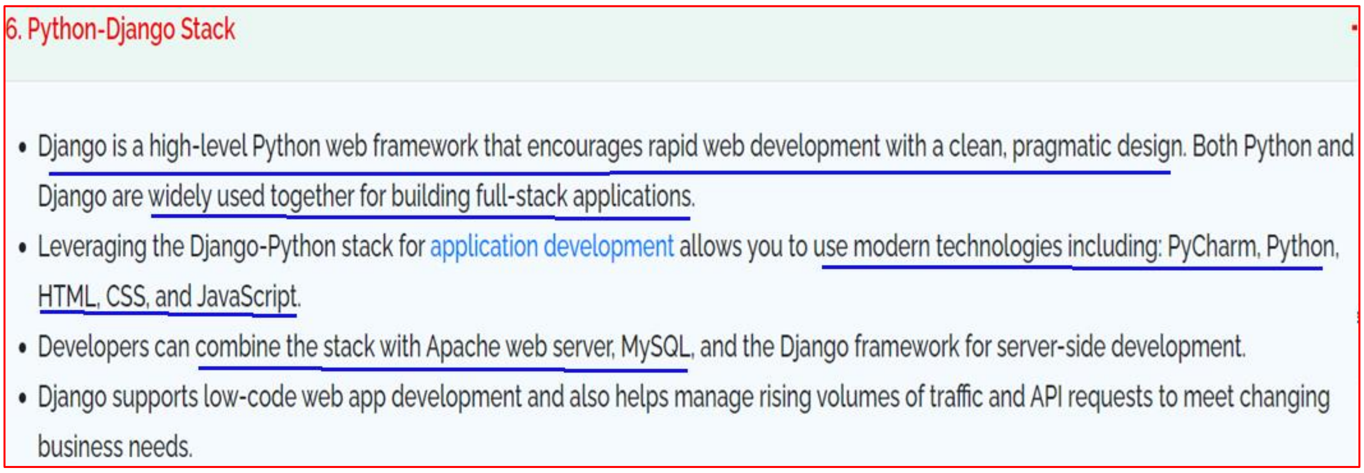
Companies that use RoR:

Companies like SlideShare, Airbnb, CrunchBase, Bloomberg, Dribble, Shopify, and GitHub (to name a few) have trusted Ruby on Rails and used the framework in their applications



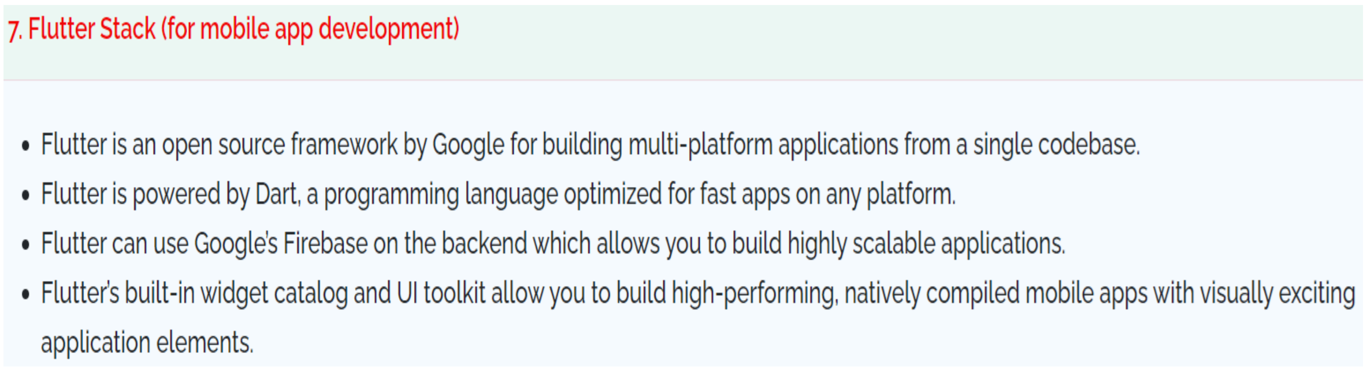
5. .NET Stack

- .NET is an open-source developer platform made up of tools, programming languages, and libraries for building modern, scalable, and high-performing desktop, web, and mobile applications that can run natively on any operating system.
- There are various implementations of .NET that allow the .NET code to execute on different operating systems – Linux, macOS, Windows, iOS, Android, and many more.
- C#, F#, and Visual Basic are the three Microsoft-supported languages for NET development. There are many third-party created languages that work well with the .NET platform.
- .NET is a highly coveted software development platform due to its advantages such as ease of development, code reusability, extensive support for high-performing applications, strong built in security measures, and active community support.



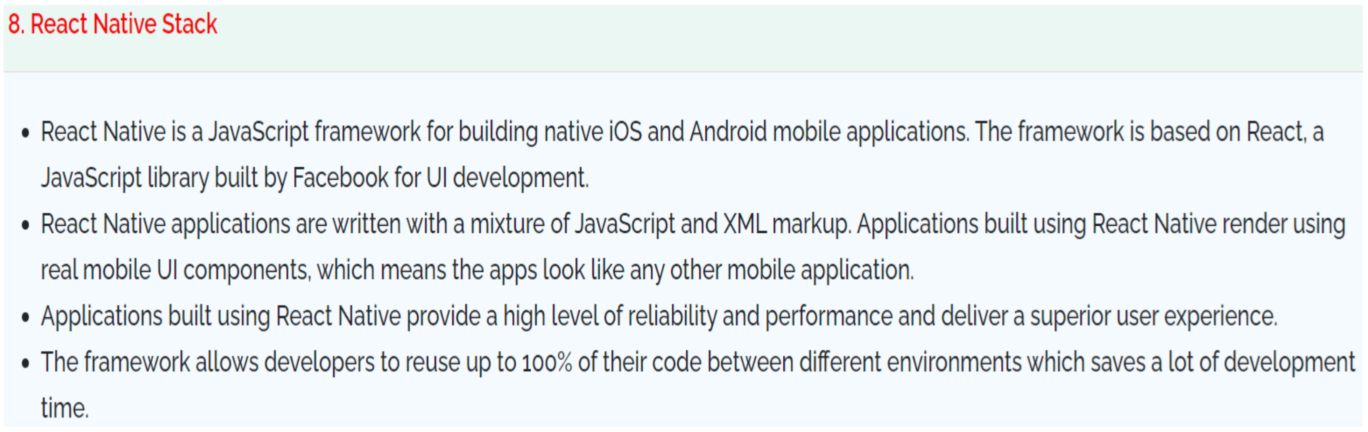
6. Python-Django Stack

- Django is a high-level Python web framework that encourages rapid web development with a clean, pragmatic design. Both Python and Django are widely used together for building full-stack applications.
- Leveraging the Django-Python stack for application development allows you to use modern technologies including: PyCharm, Python, HTML, CSS, and JavaScript.
- Developers can combine the stack with Apache web server, MySQL, and the Django framework for server-side development.
- Django supports low-code web app development and also helps manage rising volumes of traffic and API requests to meet changing business needs.



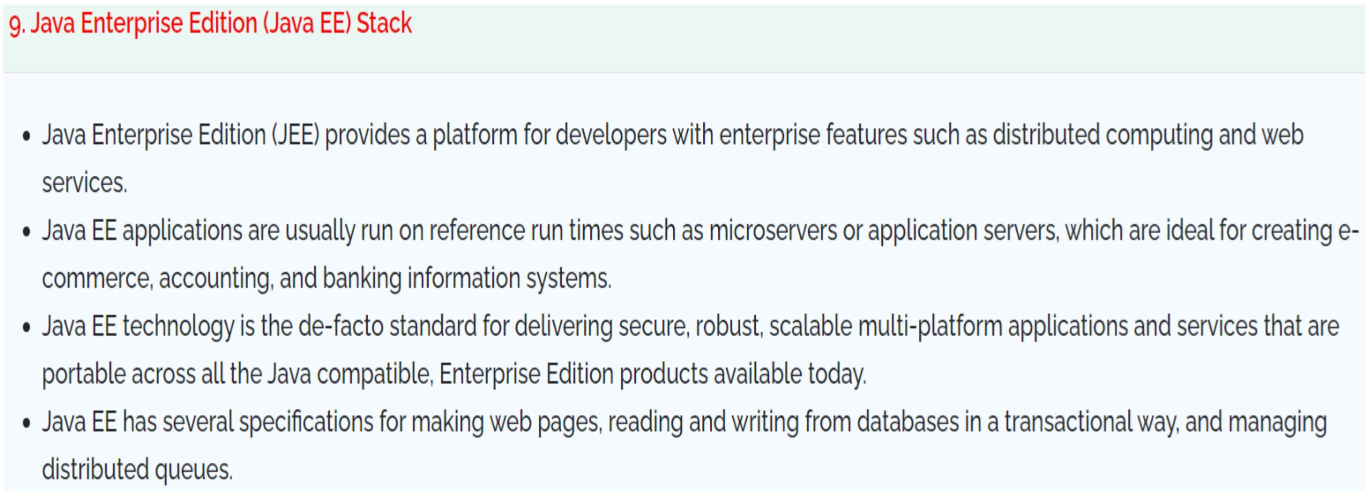
7. Flutter Stack (for mobile app development)

- Flutter is an open source framework by Google for building multi-platform applications from a single codebase.
- Flutter is powered by Dart, a programming language optimized for fast apps on any platform.
- Flutter can use Google's Firebase on the backend which allows you to build highly scalable applications.
- Flutter's built-in widget catalog and UI toolkit allow you to build high-performing, natively compiled mobile apps with visually exciting application elements.



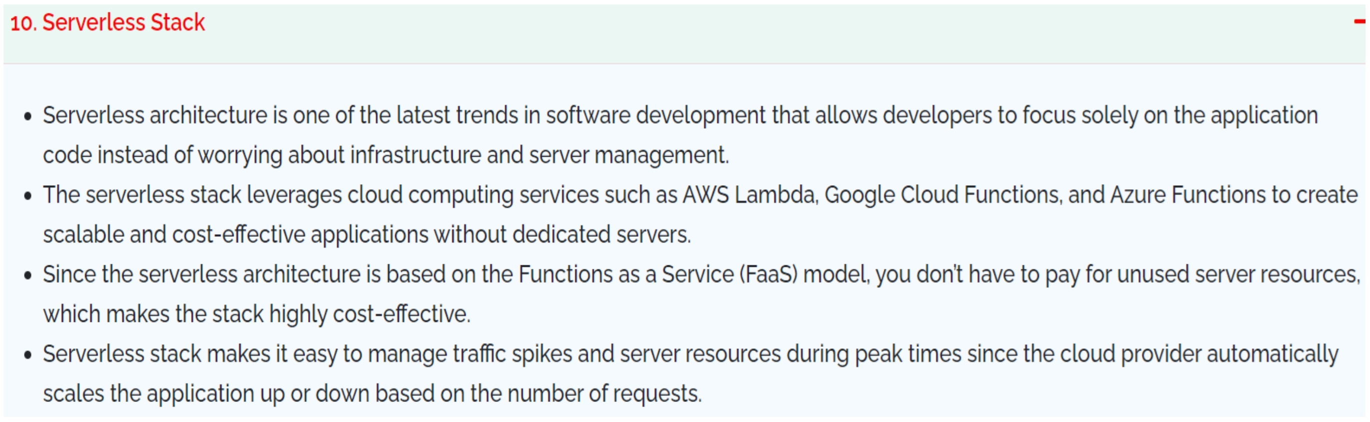
8. React Native Stack

- React Native is a JavaScript framework for building native iOS and Android mobile applications. The framework is based on React, a JavaScript library built by Facebook for UI development.
- React Native applications are written with a mixture of JavaScript and XML markup. Applications built using React Native render using real mobile UI components, which means the apps look like any other mobile application.
- Applications built using React Native provide a high level of reliability and performance and deliver a superior user experience.
- The framework allows developers to reuse up to 100% of their code between different environments which saves a lot of development time.



9. Java Enterprise Edition (Java EE) Stack

- Java Enterprise Edition (JEE) provides a platform for developers with enterprise features such as distributed computing and web services.
- Java EE applications are usually run on reference run times such as microservers or application servers, which are ideal for creating e-commerce, accounting, and banking information systems.
- Java EE technology is the de-facto standard for delivering secure, robust, scalable multi-platform applications and services that are portable across all the Java compatible, Enterprise Edition products available today.
- Java EE has several specifications for making web pages, reading and writing from databases in a transactional way, and managing distributed queues.



10. Serverless Stack

- Serverless architecture is one of the latest trends in software development that allows developers to focus solely on the application code instead of worrying about infrastructure and server management.
- The serverless stack leverages cloud computing services such as AWS Lambda, Google Cloud Functions, and Azure Functions to create scalable and cost-effective applications without dedicated servers.
- Since the serverless architecture is based on the Functions as a Service (FaaS) model, you don't have to pay for unused server resources, which makes the stack highly cost-effective.
- Serverless stack makes it easy to manage traffic spikes and server resources during peak times since the cloud provider automatically scales the application up or down based on the number of requests.

JSON and its usage in web applications:

What is JSON

JSON is an acronym for JavaScript Object Notation, is an open standard format, which is lightweight and text-based, designed explicitly for human-readable data interchange. It is a language-independent data format. It supports almost every kind of language, framework, and library.

JSON is an open standard for exchanging data on the web. It supports data structures like objects and arrays. So, it is easy to write and read data from JSON.

In JSON, data is **represented** in key-value pairs, and **curly braces** hold objects, where a **colon** is followed after each name. The **comma** is used to separate key-value pairs. **Square brackets** are used to hold arrays, where each value is comma-separated.

- JSON stands for JavaScript Object Notation.
- JSON is an open standard data-interchange format.
- JSON is lightweight and self-describing.
- JSON originated from JavaScript.
- JSON is easy to read and write.
- JSON is language independent.
- JSON supports data structures such as arrays and objects.

Features of JSON

- Simplicity
- Openness
- Self-Describing
- Internationalization
- Extensibility
- Interoperability

Why do we use JSON?

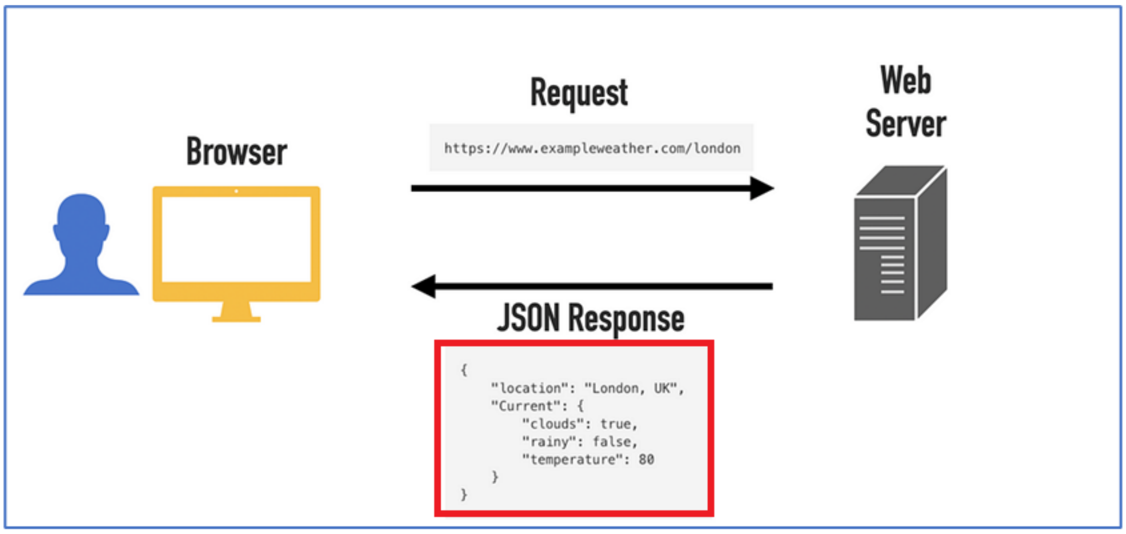
Since JSON is an **e**asy-to-use, **l**ightweight **l**anguage data interchange format in comparison to other available options, it can be used for API integration. Following are the **advantages of JSON**:

- **Less Verbose**: In contrast to XML, JSON follows a compact style to improve its users' readability. While working with a complex system, JSON tends to make substantial enhancements. (Avoid writing long and wordy sentences)
- **Faster**: The JSON parsing process is faster than that of the XML because the DOM manipulation library in XML requires extra memory

for handling large XML files. However, JSON requires less data that ultimately results in reducing the cost and increasing the parsing speed.

- **Readable**: The JSON structure is easily readable and straightforward. Regardless of the programming language that you are using, you can easily map the domain objects.
- **Structured Data**: In JSON, a map data structure is used, whereas XML follows a tree structure. The key-value pairs limit the task but facilitate the predictive and easily understandable model.
- **the Main Use of JSON:** JSON is commonly used for serializing and transmitting data over a network connection. It is used primarily to transmit data between a server and a web app.

  For **example**, when a browser opens up a weather app, the **browser** makes an API request to the webserver of the weather app. The **web server** sends back a response that is JSON. The browser then turns this JSON data into a human-readable web page.



```
{
    "location": "London, UK",
    "Current": {
        "clouds": true,
        "rainy": false,
        "temperature": 80
    }
}
```

-

JSON Data Types

- Following are the most commonly used JSON data types.

Data Type	Description	Example
String	A string is always written in double-quotes. It may consist of numbers, alphanumeric and special characters.	"student", "name", "1234", "Ver_1"
Number	Number represents the numeric characters.	121, 899
Boolean	It can be either True or False.	true
Null	It is an empty value.	

JSON Objects

In JSON, objects refer to dictionaries, which are enclosed in curly brackets, i.e., { }. These objects are written in key/value pairs, where the **key** has to be a string and **values** have to be a valid JSON data type such as string, number, object, Boolean or null. Here the key and values are separated by a colon, and a comma separates each key/value pair.

For example:

```
{"name" : "Jack", "employeeid" : 001, "present" : false}
```

JSON Arrays

In JSON, arrays can be understood as a list of objects, which are mainly enclosed in square brackets [ ]. An array value can be a string, number, object, array, boolean or null.

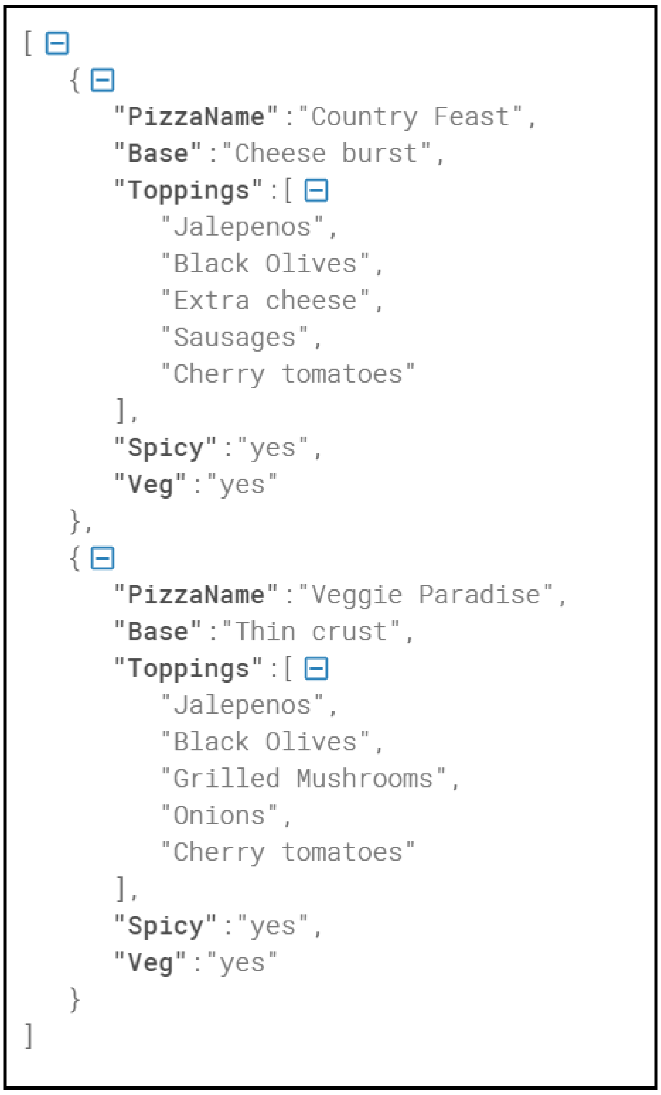
For example:

```
[
{
"PizzaName" : "Country Feast",
"Base" : "Cheese burst",
"Toppings" : ["Jalepenos", "Black Olives", "Extra cheese", "Sausages", "Ch
erry tomatoes"],
"Spicy" : "yes",
"Veg" : "yes"
},

{
"PizzaName" : "Veggie Paradise",
"Base" : "Thin crust",
"Toppings" : ["Jalepenos", "Black Olives", "Grilled Mushrooms", "Onions",
"Cherry tomatoes"],
"Spicy" : "yes",
"Veg" : "yes"
}
]
```

In the above example, the object "**Pizza**" is an array. It contains **five** objects, i.e., PizzaName, Base, Toppings, Spicy, and Veg.

Output:



```
[
   {
      "PizzaName":"Country Feast",
      "Base":"Cheese burst",
      "Toppings":[
         "Jalepenos",
         "Black Olives",
         "Extra cheese",
         "Sausages",
         "Cherry tomatoes"
      ],
      "Spicy":"yes",
      "Veg":"yes"
   },
   {
      "PizzaName":"Veggie Paradise",
      "Base":"Thin crust",
      "Toppings":[
         "Jalepenos",
         "Black Olives",
         "Grilled Mushrooms",
         "Onions",
         "Cherry tomatoes"
      ],
      "Spicy":"yes",
      "Veg":"yes"
   }
]
```

JSON Array of Objects

Let's see a simple JSON array example having **4 objects.**

```
{"employees":[
    {"name":"Ram", "email":"ram@gmail.com", "age":23},
    {"name":"Shyam", "email":"shyam23@gmail.com", "age":28},
    {"name":"John", "email":"john@gmail.com", "age":33},
    {"name":"Bob", "email":"bob32@gmail.com", "age":41}
]}
```

JSON Multidimensional Array

We can store array inside JSON array, it is known as array of arrays or multidimensional array.

```
[
 [ "a", "b", "c" ],
  [ "m", "n", "o" ],
  [ "x", "y", "z" ]
]
```

JSON Comments

JSON doesn't support comments. It is not a standard.

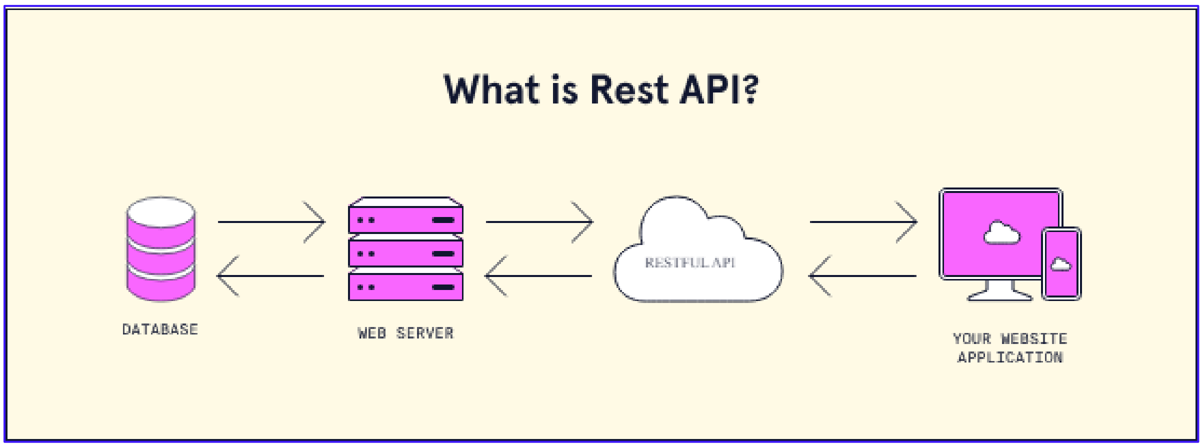
But you can do some tricks such as adding extra attribute for comment in JSON object, for example:

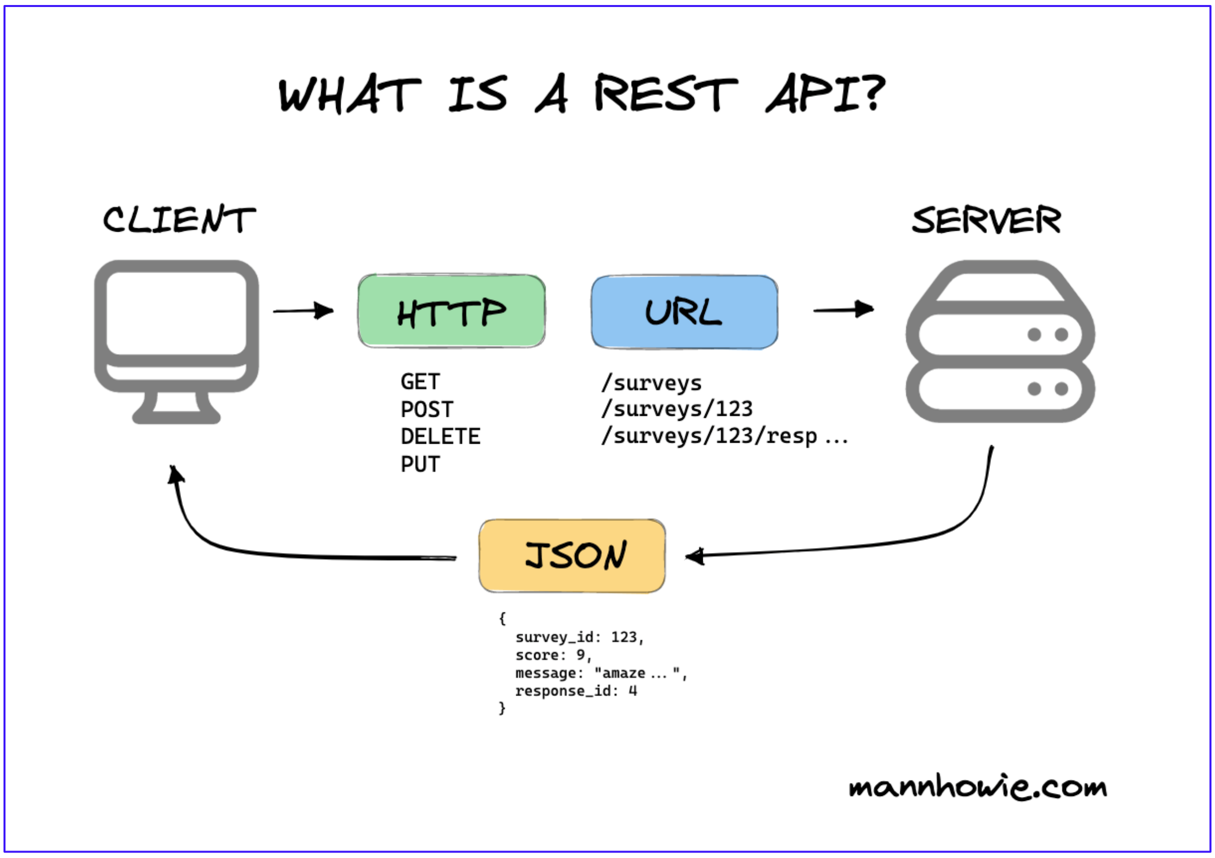
```
{
    "employee": {
        "name":       "Bob",
        "salary":     56000,
        "comments":    "He is a nice man"
    }
}
```

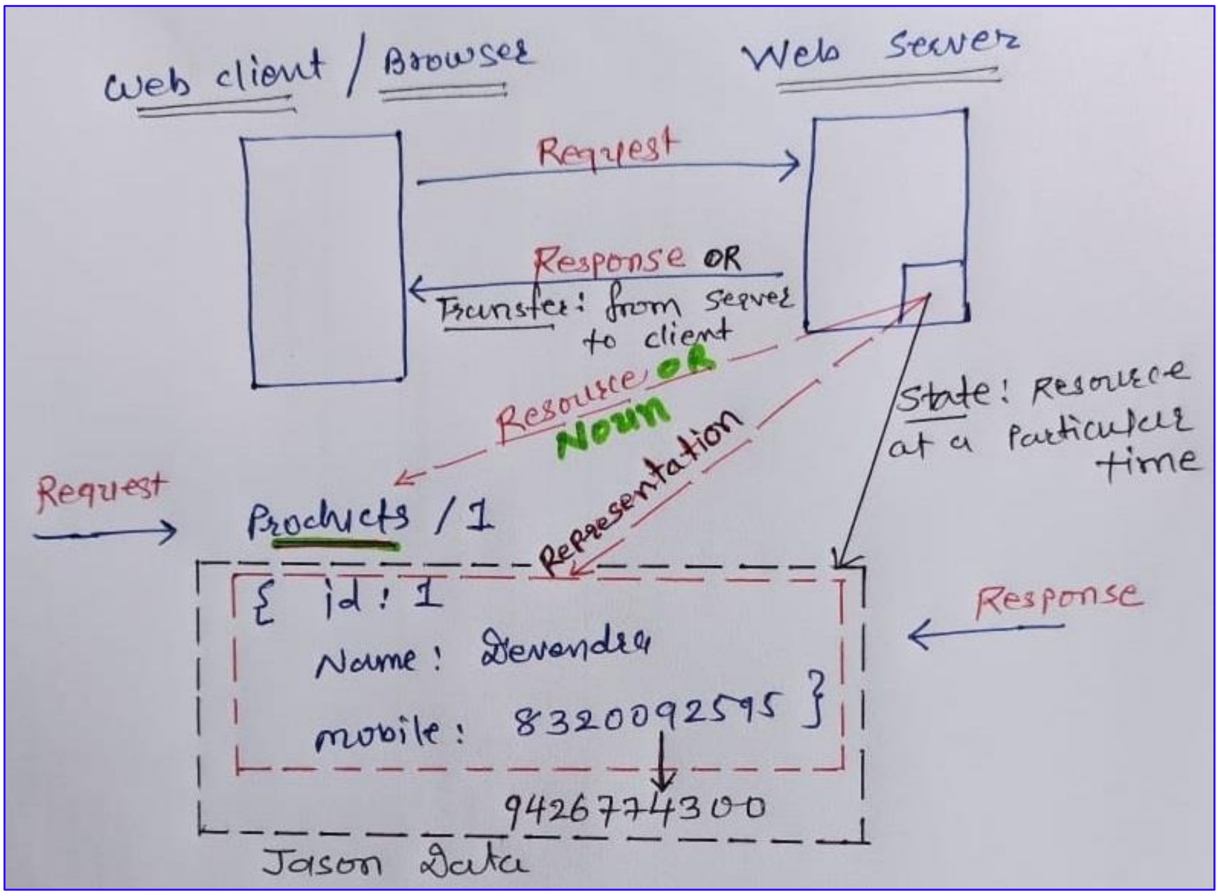
Here, "comments" attribute can be treated as comment.

RESTAPIs:

A. Understanding REpresentational State Transfer (REST)







REpresentational **S**tate **T**ransfer:

REST, or REpresentational State Transfer, is an architectural **style** for providing standards between computer systems on the web, making it easier for systems to communicate with each other. REST-compliant systems, often called **RESTful** systems, are characterized by how they are **stateless** and separate the concerns of client and server. We will go into what these terms mean and why they are beneficial characteristics for services on the Web.

B. The main constraints of of a RESTful API:

1. Separation of Client and Server:

In the REST architectural style, the implementation of the client and the implementation of the server can be done independently without each knowing about the other. This means that the code on the client side can be changed at any time without affecting the operation of the server, and the code on the server side can be changed without affecting the operation of the client.

As long as each side knows what format of messages to send to the other, they can be kept modular and separate. Separating the user interface concerns from the data storage concerns, we improve the flexibility of the interface across platforms and *improve scalability* (system's ability to handle an increasing amount of work, usually by accommodating a growing number of users, transactions, or data volume.) by simplifying the server components. Additionally, the separation allows each component the ability to evolve independently.

By using a REST interface, different clients hit the same REST endpoints, perform the same actions, and receive the same responses.

2. Statelessness:

Systems that follow the REST paradigm are stateless, meaning that the server does not need to know anything about what state the client is in and vice versa. In this way, both the server and the client can understand any message received, even without seeing previous messages. This constraint of statelessness is enforced through the use of resources rather than commands. **Resources** are the nouns of the Web - they describe any object, document, or thing that you may need to store or send to other services.

Because REST systems interact through standard operations on resources, they do not rely on the implementation of interfaces.

3. Cacheability: Responses from the server can explicitly indicate whether they are cacheable or not. This allows clients to cache responses, reducing the need for repeated requests to the server and improving performance.

4. Uniform Interface: The uniform interface constraint is further broken down into several sub-constraints:

I. **Identification of Resources:** Resources (entities or services) are identified by unique URIs (Uniform Resource Identifiers).

II. **Resource Manipulation through Representations:** Resources are manipulated through representations, such as XML or JSON. Clients interact with these representations rather than directly with the resources.

III. **Self-Descriptive Messages:** Each message sent from the server to the client (or vice versa) includes all the information needed to understand and process the request or response. This includes information about how to process the data (e.g., media types).

IV. **Hypermedia as the Engine of Application State (HATEOAS)**: The server provides hypermedia links within the response that guide the client's transitions to related resources. Clients navigate the application's functionality through these links.

These **constraints** help RESTful applications achieve reliability, quick performance, and scalability, as components that can be managed, updated, and reused without affecting the system as a whole, even during operation of the system.

Now, we'll explore how the communication between the client and server actually happens when we are implementing a RESTful interface.

C. Communication between Client and Server:

In the REST architecture, **clients** send requests to retrieve or modify resources, and **servers** send responses to these requests. Let's take a look at the standard ways to make requests and send responses.

Making Requests:

REST requires that a client make a request to the server in order to retrieve or modify data on the server. *A request generally consists of:*

I. an HTTP verb, which defines what kind of operation to perform
II. a header, which allows the client to pass along information about the request
III. a path to a resource
IV. an optional message body containing data

I. HTTP Verbs

There are 4 basic HTTP verbs we use in requests to interact with resources in a REST system:

- GET — retrieve a specific resource (by id) or a collection of resources
- POST — create a new resource
- PUT — update a specific resource (by id)
- DELETE — remove a specific resource by id

II. Headers and Accept parameters

In the header of the request, the client sends the type of content that it is able to receive from the server. This is called the Accept field, and it ensures that the server does not send data that cannot be understood or processed by the client. The options for types of content are MIME Types (or Multipurpose Internet Mail Extensions),

MIME Types, used to specify the content types in the Accept field, consist of a type and a subtype. They are separated by a slash (/).

For ***example***, a text file containing HTML would be specified with the type text/html. If this text file contained CSS instead, it would be specified as text/css. A generic text file would be denoted as text/plain. This default value, text/plain, is not a catch-all, however. If a client is expecting text/css and receives text/plain, it will not be able to recognize the content.

Other types and commonly used subtypes:

image — image/png, image/jpeg, image/gif
audio — audio/wav, audio/mpeg
video — video/mp4, video/ogg
application — application/json, application/pdf, application/xml, applicatio n/octet-stream

For example, a client accessing a resource with id 23 in an articles resource on a server might send a GET request like this:

```
GET /articles/23
Accept: text/html, application/xhtml
```

The Accept header field in this case is saying that the client will accept the content in text/html or application/xhtml.

III. Paths

Requests must contain a path to a resource that the operation should be performed on. In RESTful APIs, paths should be designed to help the client know what is going on.

Conventionally, the first part of the path should be the plural form of the resource. This keeps nested paths simple to read and easy to understand.

A **path** like *fashionboutique.com/customers/223/orders/12* is clear in what it points to, even if you've never seen this specific path before, because it is hierarchical and descriptive. We can see that we are accessing the *order with id 12* for the *customer with id 223.*

Paths should contain the information necessary to locate a resource with the degree of specificity needed. When referring to a list or collection of resources, it is not always necessary to add *an id*. For example, a POST request to the *fashionboutique.com/customers* path would not need an extra identifier, as the server will *generate an id* for the new object.

If we are trying to access a single resource, we would need to append an id to the path. For example: GET fashionboutique.com/customers/:id — retrieves the item in the customers resource with the id specified. *DELETE fashionboutique.com/customers/:id* — deletes the item in the customers resource with the id specified.

IV. Sending Responses

1) Content Types

In cases where the server is sending a data payload to the client, the server must include a *content-type* in the header of the response. This content-type header field alerts the client to the type of data it is sending in the response body. These content types are MIME Types, just as they are in the accept field of the request header. The *content-type* that the server sends back in the response should be one of the options that the client specified in the *accept* field of the request.

For example, when a client is accessing a resource with id 23 in an articles resource with this GET Request:

```
GET /articles/23 HTTP/1.1
Accept: text/html, application/xhtml
```

The server might send back the content with the response header:

```
HTTP/1.1 200 (OK)
Content-Type: text/html
```

This would signify that the content requested is being returned in the response body with a *content-type* of *text/html*, which the client said it would be able to accept.

2) Response Codes

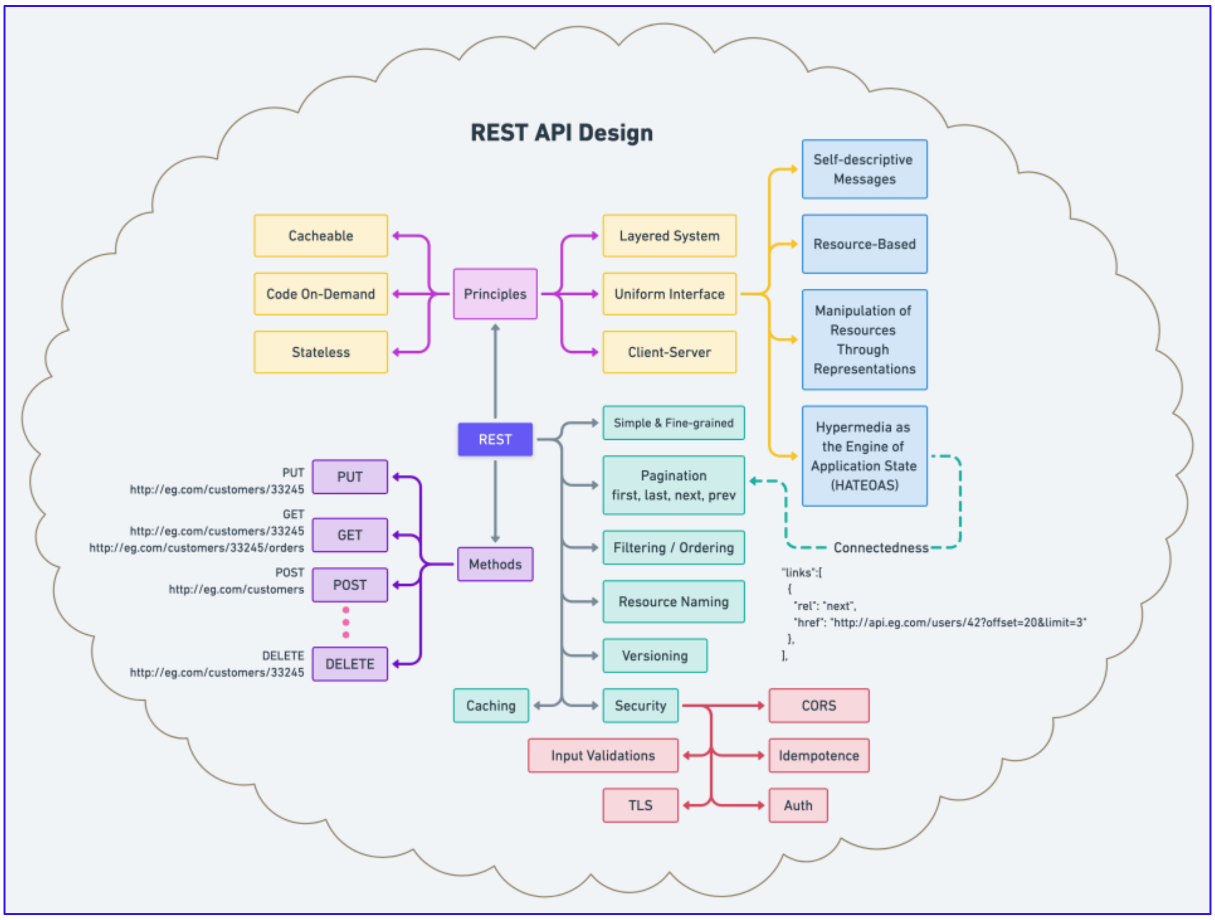
Responses from the server contain status codes to alert the client to information about the success of the operation. As a developer, you do not need to know every status code (there are many of them), but you should know the most common ones and how they are used:

Status code	Meaning
200 (OK)	This is the standard response for successful HTTP requests.
201 (CREATED)	This is the standard response for an HTTP request that resulted in an item being successfully created.

Status code	Meaning
204 (NO CONTENT)	This is the standard response for successful HTTP requests, where nothing is being returned in the response body.
400 (BAD REQUEST)	The request cannot be processed because of bad request syntax, excessive size, or another client error.
403 (FORBIDDEN)	The client does not have permission to access this resource.
404 (NOT FOUND)	The resource could not be found at this time. It is possible it was deleted, or does not exist yet.
500 (INTERNAL SERVER ERROR)	The generic answer for an unexpected failure if there is no more specific information available.

For each HTTP verb, there are expected status codes a server should return upon success:

GET — return 200 (OK)
POST — return 201 (CREATED)
PUT — return 200 (OK)
DELETE — return 204 (NO CONTENT) If the operation fails, return the most specific status code possible corresponding to the problem that was encountered.



```
"links":[
  {
    "rel": "next",
    "href": "http://api.eg.com/users/42?offset=20&limit=3"
  },
],
```