1. **What is ExpressJS explain features of express?**

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.

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.

1. **What is middleware? What are the types of middleware present in ExpressJS explain briefly?**

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application’s request-response cycle. The next middleware function is commonly denoted by a variable named next.

Middleware functions can perform the following tasks:

Execute any code.

Make changes to the request and the response objects.

End the request-response cycle.

Call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

An Express application can use the following types of middleware:

Application-level middleware

Router-level middleware

Error-handling middleware

Built-in middleware

Third-party middleware

1. **What is template engine, explain different types of template engines in express?**

A template engine facilitates you to use static template files in your applications. At runtime, it replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. So this approach is preferred to design HTML pages easily.

Following is a list of some popular template engines that work with Express.js:

Pug: Haml-inspired template engine (formerly Jade).

Haml.js: Haml implementation.

EJS: Embedded JavaScript template engine.

hbs: Adapter for Handlebars.js, an extension of Mustache.js template engine.

Squirrelly: Blazing-fast template engine that supports partials, helpers, custom tags, filters, and caching. Not white-space sensitive, works with any language.

Eta: Super-fast lightweight embedded JS template engine. Supports custom delimiters, async, whitespace control, partials, caching, plugins.

React: Renders React components on the server. It renders static markup and does not support mounting those views on the client.

combyne.js: A template engine that hopefully works the way you’d expect.

Nunjucks: Inspired by jinja/twig.

marko: A fast and lightweight HTML-based templating engine that compiles templates to CommonJS modules and supports streaming, async rendering and custom tags. (Renders directly to the HTTP response stream).

whiskers: Small, fast, mustachioed.

Blade: HTML Template Compiler, inspired by Jade & Haml.

Haml-Coffee: Haml templates where you can write inline CoffeeScript.

express-hbs: Handlebars with layouts, partials and blocks for express 3 from Barc.

express-handlebars: A Handlebars view engine for Express which doesn’t suck.

express-views-dom: A DOM view engine for Express.

rivets-server: Render Rivets.js templates on the server.

LiquidJS: A simple, expressive and safe template engine.

express-tl: A template-literal engine implementation for Express.

Twing: First-class Twig engine for Node.js.

Sprightly: A very light-weight JS template engine (45 lines of code), that consists of all the bare-bones features that you want to see in a template engine.

1. **Explain briefly about POST, PUT, GET, DELETE requests with examples?**

**HTTP POST**

Use POST APIs to create new subordinate resources, e.g., a file is subordinate to a directory containing it or a row is subordinate to a database table. When talking strictly in terms of REST, POST methods are used to create a new resource into the collection of resources.

Ideally, if a resource has been created on the origin server, the response SHOULD be HTTP response code 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header.

Many times, the action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either HTTP response code 200 (OK) or 204 (No Content) is the appropriate response status.

Responses to this method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields.

POST is neither safe nor idempotent, and invoking two identical POST requests will result in two different resources containing the same information (except resource ids).

Example request URIs

HTTP POST http://www.appdomain.com/users

HTTP POST http://www.appdomain.com/users/123/accounts

**HTTP PUT**

Use PUT APIs primarily to update existing resource (if the resource does not exist, then API may decide to create a new resource or not). If a new resource has been created by the PUT API, the origin server MUST inform the user agent via the HTTP response code 201 (Created) response and if an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

The difference between the POST and PUT APIs can be observed in request URIs. POST requests are made on resource collections, whereas PUT requests are made on a single resource.

Example request URIs

HTTP PUT http://www.appdomain.com/users/123

HTTP PUT http://www.appdomain.com/users/123/accounts/456

**HTTP GET**

Use GET requests to retrieve resource representation/information only – and not to modify it in any way. As GET requests do not change the state of the resource, these are said to be safe methods. Additionally, GET APIs should be idempotent, which means that making multiple identical requests must produce the same result every time until another API (POST or PUT) has changed the state of the resource on the server.

If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process.

For any given HTTP GET API, if the resource is found on the server, then it must return HTTP response code 200 (OK) – along with the response body, which is usually either XML or JSON content (due to their platform-independent nature).

In case resource is NOT found on server then it must return HTTP response code 404 (NOT FOUND). Similarly, if it is determined that GET request itself is not correctly formed then server will return HTTP response code 400 (BAD REQUEST).

Example request URIs

HTTP GET http://www.appdomain.com/users

HTTP GET http://www.appdomain.com/users?size=20&page=5

HTTP GET http://www.appdomain.com/users/123

HTTP GET http://www.appdomain.com/users/123/address

**HTTP DELETE**

As the name applies, DELETE APIs are used to delete resources (identified by the Request-URI).

A successful response of DELETE requests SHOULD be HTTP response code 200 (OK) if the response includes an entity describing the status, 202 (Accepted) if the action has been queued, or 204 (No Content) if the action has been performed but the response does not include an entity.

DELETE operations are idempotent. If you DELETE a resource, it’s removed from the collection of resources. Repeatedly calling DELETE API on that resource will not change the outcome – however, calling DELETE on a resource a second time will return a 404 (NOT FOUND) since it was already removed. Some may argue that it makes the DELETE method non-idempotent. It’s a matter of discussion and personal opinion.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

Example request URIs

HTTP DELETE http://www.appdomain.com/users/123

HTTP DELETE <http://www.appdomain.com/users/123/accounts/456>

1. **What are node mailer, multer and bcrypt explain briefly?**

Nodemailer is a Node.js module that allows you to send emails from your server with ease. Whether you want to communicate with your users or just notify yourself when something has gone wrong, one of the options for doing so is through mail.

Its main features include (but are not limited to):

* platform-independence
* security, in particular, email delivery with TLS/STARTTLS and DKIM email authentication
* Unicode support
* HTML content and embedded image attachments
* different transport methods besides SMTP support.

Multer is a node.js middleware for handling multipart/form-data, which is primarily used for uploading files. It is written on top of busboy for maximum efficiency. Multer adds a body object and a file or files object to the request object. The bodyobject contains the values of the text fields of the form, the file or files object contains the files uploaded via the form.

Bcrypt is a popular and trusted method for salt and hashing passwords.The bcrypt hashing function allows us to build a password security platform that scales with computation power and always hashes every password with a salt.