Express JS Concept

Concept	Explanation	Usage	Example
express( )	The core function used to create an Express application instance. It initializes the app and allows setting up routes and middleware.	javascript const app = express(); // Initializes Express application	javascript const express = require('express'); const app = express(); // Set up Express app
app.listen( )	Starts the Express server and listens on a specified port for incoming requests.	javascript app.listen(port, callback); // Starts the server on a given port	javascript app.listen(3000, () => { console.log('Server is running on port 3000'); });
app.get( )	Defines a route handler for GET requests to the specified route. Commonly used to handle requests for displaying content or returning data.	javascript app.get(path, callback); // Handles GET requests for a specific route	javascript app.get('/', (req, res) => { res.send('Hello World'); }); // Responds to GET request at '/'
app.post( )	Defines a route handler for POST requests. Used when handling form submissions, file uploads, or any request that sends data to the server.	javascript app.post(path, callback); // Handles POST requests for a specific route	javascript app.post('/submit', (req, res) => { res.send('Form submitted'); });
app.put( )	Defines a route handler for PUT requests, typically used to update an existing resource.	javascript app.put(path, callback); // Handles PUT requests to update existing data	javascript app.put('/update/:id', (req, res) => { res.send('Resource updated'); });
app.delete( )	Defines a route handler for DELETE requests, commonly used to remove resources on the server.	javascript app.delete(path, callback); // Handles DELETE requests for a specific route	javascript app.delete('/delete/:id', (req, res) => { res.send('Resource deleted'); });
req (Request)	An object that contains information about the HTTP request, such as headers, query parameters, body data, and more.	javascript app.get(path, (req, res) => { const param = req.query.param; // Access query parameters from URL });	javascript app.get('/user', (req, res) => { const name = req.query.name; res.send(`Hello ${name}`); });
res (Response)	An object that provides methods for sending a response back to the client. It is used to send data, status codes, or HTML views back to the user.	javascript app.get(path, (req, res) => { res.send('Response data'); }); // Sends response data to client	javascript app.get('/home', (req, res) => { res.json({ message: 'Welcome' }); }); // Sends JSON response
next( )	A function used in middleware to pass control to the next middleware function or route handler. It is essential for chaining multiple middleware functions.	javascript function middleware(req, res, next) { next(); } // Pass control to next middleware	javascript app.use((req, res, next) => { console.log('Request received'); next(); }); // Pass control to next middleware
app.use( )	Registers middleware for a route or application-wide. It's used for tasks like logging, authentication, error handling, and more.	javascript app.use(middleware); // Use a middleware function for all routes or specific ones	javascript app.use(express.json()); // Middleware for parsing JSON bodies in requests
app.all( )	Defines a route handler for all HTTP methods (GET, POST, PUT, DELETE, etc.) on a specific	javascript app.all(path, callback); // Handles all HTTP methods for a specific route	javascript app.all('/resource', (req, res) => { res.send('Request received'); }); // Handles all methods at '/resource' route

	route. Useful for handling multiple types of requests at once.		
req.params	Contains route parameters (like :id) passed in the URL. Used for dynamic route handling, e.g., for fetching specific records from a database.	javascript app.get('/user/:id', (req, res) => { const userId = req.params.id; res.send(`User ID: ${userId}`); });	javascript app.get('/product/:productId', (req, res) => { const productId = req.params.productId; res.send(productId); });
req.query	Contains the query parameters from the URL (e.g., ?name=John). Often used for filtering, sorting, and searching functionality.	javascript app.get('/search', (req, res) => { const searchTerm = req.query.query; res.send(`Searching for ${searchTerm}`); });	javascript app.get('/search', (req, res) => { const filter = req.query.filter; res.send(`Filtering by: ${filter}`); });
req.body	Contains data sent in a POST, PUT, or PATCH request, typically from a form submission or API call. It's used to extract data from the request payload.	javascript app.post('/submit', (req, res) => { const name = req.body.name; res.send(`Received name: ${name}`); });	javascript app.post('/signup', (req, res) => { const { username, password } = req.body; res.send('User signed up'); });
res.send( )	Sends the response to the client. It can send various types of data like strings, HTML, JSON, or buffers.	javascript res.send('Response content'); // Send content back to the client	javascript app.get('/', (req, res) => { res.send('Hello, World!'); }); // Sends a simple string response
res.json( )	Sends a JSON response to the client. The object passed to res.json() will automatically be stringified and returned as JSON.	javascript res.json({ key: 'value' }); // Send JSON response back to the client	javascript app.get('/data', (req, res) => { res.json({ message: 'Here is your data' }); });
res.status( )	Sets the HTTP status code for the response. Often used to indicate whether a request was successful or encountered an error.	javascript res.status(code).send('Message'); // Send response with a specific status code	javascript app.get('/', (req, res) => { res.status(200).send('Success'); }); // Sends success status code
res.redirect()	Redirects the client to a different route or URL. It's used for page redirection after an operation or event.	javascript res.redirect(url); // Redirect to another URL or route	javascript app.get('/old-url', (req, res) => { res.redirect('/new-url'); }); // Redirect from '/old-url' to '/new-url'

JavaScript Asynchronous Concepts

Concept	Explanation	Usage	Example
try	Defines a block of code that will be tested for errors.	try { // code }	javascript try { let x = y + 1; } catch (error) { console.error(error); }
catch	Defines a block of code that handles errors from a try block.	catch (error) { // error handling }	javascript try { throw new Error('Test'); } catch (error) { console.error(error.message); }
async	Declares a function as asynchronous, enabling the use of await.	async functionName() { // async code }	javascript const fetchData = async () => { const data = await fetch('url'); return data; };
await	Pauses execution in an async function until the promise resolves.	const result = await somePromise;	javascript const result = await fetch('url'); console.log(result);
throw	Used to manually throw an error, allowing for custom error handling.	throw new Error('Error message');	javascript if (!data) { throw new Error('Data not found'); }
finally	Executes a block of code after try/catch, regardless of whether an exception occurred.	finally { // cleanup code }	javascript try { // code } catch (error) { console.error(error); } finally { console.log('Cleanup'); }
Promise	Represents a value that may be available in the future.	const promise = new Promise((resolve, reject));	javascript const promise = new Promise((resolve) => { resolve('Success'); }); promise.then((msg) => console.log(msg));
.then()	Executes a callback when a promise resolves successfully.	promise.then(result => { // handle result });	javascript promise.then((result) => { console.log(result); }).catch((error) => console.error(error));
.catch()	Executes a callback when a promise is rejected.	promise.catch(error => { // handle error });	javascript promise.catch((error) => { console.error('Error:', error); });
.finally()	Executes code after a promise settles (resolved or rejected).	promise.finally(() => { // final task });	javascript promise.finally(() => { console.log('Task completed'); });
setTimeout()	Executes code after a specified delay.	setTimeout(() => { // code }, delay);	javascript setTimeout(() => { console.log('Hello after 2 seconds'); }, 2000);
setInterval()	Executes code repeatedly at specified intervals.	setInterval(() => { // code }, interval);	javascript const interval = setInterval(() => { console.log('Repeating...'); }, 1000); clearInterval(interval);
forEach()	Iterates over an array, executing a provided function for each element.	array.forEach(item => { // code });	javascript [1, 2, 3].forEach((num) => { console.log(num); });
map()	Creates a new array with the results of a provided function on every element.	array.map(item => { // transformation });	javascript const nums = [1, 2, 3]; const doubled = nums.map((num) => num * 2); console.log(doubled); // [2, 4, 6]
filter()	Creates a new array with elements that pass the provided test function.	array.filter(item => { // test });	javascript const nums = [1, 2, 3, 4]; const even = nums.filter((num) => num % 2 === 0); console.log(even); // [2, 4]
reduce()	Reduces an array to a single value using a reducer function.	array.reduce((acc, item) => { // reducer }, init);	javascript const nums = [1, 2, 3]; const sum = nums.reduce((acc, num) => acc + num, 0); console.log(sum); // 6
Object.keys()	Returns an array of an object's enumerable property names.	Object.keys(object);	javascript const obj = { a: 1, b: 2 }; const keys = Object.keys(obj); console.log(keys); // ['a', 'b']

Object.values()	Returns an array of an object's enumerable property values.	Object.values(object);	javascript const obj = { a: 1, b: 2 }; const values = Object.values(obj); console.log(values); // [1, 2]
Object.entries()	Returns an array of an object's enumerable key-value pairs.	Object.entries(object);	javascript const obj = { a: 1, b: 2 }; const entries = Object.entries(obj); console.log(entries); // [['a', 1], ['b', 2]]

JavaScript, MongoDB and Cloudinary

Category	Instance	Explanation	Example Usage
JavaScript	.split(delimiter)	Splits a string into an array based on a specified delimiter.	'a,b,c'.split(',') // ['a', 'b', 'c']
	.pop()	Removes and returns the last element of an array.	const arr = [1, 2, 3]; arr.pop(); // 3
	.push(element)	Adds one or more elements to the end of an array.	const arr = [1, 2]; arr.push(3); // [1, 2, 3]
	.map(callback)	Creates a new array by applying a function to each element of an array.	[1, 2, 3].map(x => x * 2); // [2, 4, 6]
	.filter(callback)	Filters elements of an array based on a condition.	[1, 2, 3].filter(x => x > 1); // [2, 3]
	.reduce(callback, initial)	Reduces an array to a single value by applying a function.	[1, 2, 3].reduce((sum, x) => sum + x, 0); // 6
	.forEach(callback)	Executes a function for each element in an array.	[1, 2, 3].forEach(x => console.log(x));
	.join(delimiter)	Joins all elements of an array into a string, separated by a delimiter.	[1, 2, 3].join('-'); // '1-2-3'
	.includes(value)	Checks if a string or array contains a specified value.	[1, 2, 3].includes(2); // true
	.find(callback)	Finds the first element in an array that satisfies a condition.	[1, 2, 3].find(x => x > 1); // 2
	.slice(start, end)	Returns a shallow copy of an array or string from start to end (non-inclusive).	[1, 2, 3, 4].slice(1, 3); // [2, 3]

	.trim()	Removes whitespace from both ends of a string.	' hello '.trim(); // 'hello'
	.sort(callback)	Sorts elements of an array in place.	[3, 1, 2].sort(); // [1, 2, 3]
	.toString()	Returns a string representation of an object.	const obj = { a: 1 }; obj.toString(); // '[object Object]'
	.toFixed(digits)	Formats a number to a fixed number of decimal places.	(123.456).toFixed(2); // '123.46'
	.error(message)	Throws an error with a custom message.	throw new Error('Something went wrong!');
	.console()	Logs data to the console.	console.log('Hello world!');
MongoDB	.connect(uri, options)	Establishes a connection to the database.	await mongoose.connect('mongodb://localhost:27017/mydb');
	.save()	Saves a document to the database.	const doc = new Model(data); await doc.save();
	.validate()	Runs validation on a document manually.	const doc = new Model({ key: 'value' }); await doc.validate();
	.find(query)	Finds documents matching the query.	const docs = await Model.find({ key: 'value' });
	.findOne(query)	Finds a single document matching the query.	const doc = await Model.findOne({ key: 'value' });
	.findById(id)	Finds a document by its unique ID.	const doc = await Model.findById('12345');
	.updateOne(query, update)	Updates a single document matching the query.	await Model.updateOne({ key: 'value' }, { $set: { key: 'newValue' } });
	.updateMany(query, update)	Updates multiple documents matching the query.	await Model.updateMany({ key: 'value' }, { $set: { key: 'newValue' } });
	.deleteOne(query)	Deletes a single document matching the query.	await Model.deleteOne({ key: 'value' });

	.deleteMany(query)	Deletes multiple documents matching the query.	await Model.deleteMany({ key: 'value' });
	.populate(field)	Populates referenced fields in the document.	const doc = await Model.findById(id).populate('relatedField');
	.countDocuments(query)	Counts the number of documents matching the query.	const count = await Model.countDocuments({ key: 'value' });
	.distinct(field, query)	Finds distinct values for a specified field.	const values = await Model.distinct('fieldName', { key: 'value' });
	.aggregate(pipeline)	Performs advanced aggregation operations like grouping and filtering.	await Model.aggregate([{ $group: { _id: '$key', total: { $sum: '$value' } } }]);
	.create(docs)	Creates and inserts one or more documents.	await Model.create([{ key: 'value1' }, { key: 'value2' }]);
	.error()	Returns the error object (in case of a failure).	Model.find({}).catch((err) => console.error(err));
Cloudinary	.config(options)	Configures Cloudinary SDK with credentials and settings.	cloudinary.config({ cloud_name: 'my-cloud', api_key: '123', api_secret: 'abc' });
	.uploader.upload()	Uploads a file to Cloudinary.	const result = await cloudinary.uploader.upload('/path/to/file', { folder: 'my-folder' });
	.uploader.destroy()	Deletes an asset by its public ID.	await cloudinary.uploader.destroy('public_id');
	.uploader.rename()	Renames an uploaded asset.	await cloudinary.uploader.rename('old_id', 'new_id');
	.api.resource()	Fetches metadata for a single asset by public ID.	const result = await cloudinary.api.resource('public_id');
	.api.resources()	Fetches metadata for multiple assets.	const result = await cloudinary.api.resources({ prefix: 'folder/' });

	.api.resources_by_tag()	Fetches assets with a specific tag.	const result = await cloudinary.api.resources_by_tag('tag-name');
	.image()	Generates an optimized URL for an image with transformations.	const url = cloudinary.image('public_id', { width: 300, crop: 'scale' });
	.video()	Generates an optimized URL for a video with transformations.	const url = cloudinary.video('public_id', { width: 300, crop: 'scale' });
	.uploader.explicit()	Explicitly processes an already-uploaded resource.	const result = await cloudinary.uploader.explicit('public_id', { type: 'upload' });
	.error()	Logs an error message for Cloudinary API failures.	cloudinary.error('Error uploading image');

Logical Operators

Operator	Description	Example	Result
=	Assignment operator	x = 5	x is 5
+	Addition	5 + 2	7
-	Subtraction	5 - 2	3
*	Multiplication	5 * 2	10
/	Division	5 / 2	2.5
%	Modulus (remainder)	5 % 2	1
**	Exponentiation	2 ** 3	8
==	Equality (loose, converts types)	5 == '5'	true
===	Strict equality (no type conversion)	5 === '5'	false
!=	Inequality (loose)	5 != '5'	false
!==	Strict inequality	5 !== '5'	true
<	Less than	5 < 10	true
>	Greater than	10 > 5	true
<=	Less than or equal	5 <= 5	true
>=	Greater than or equal	10 >= 5	true
&&	Logical AND	true && false	false
||	Logical OR	true || false; // Returns true false || false; // Returns false	Logical OR - cause at least one is true
!	Logical NOT	!true	false
??	Nullish coalescing (returns first non-null/undefined value)	null ?? 'default'	'default'
?:	Ternary (conditional)	x > 5 ? 'yes' : 'no'	'yes' or 'no'
+ (unary)	Converts operand to a number	+'5'	5
- (unary)	Negates a number	-5	-5

++	Increment	let x = 5; x++	x = 6
--	Decrement	let x = 5; x--	x = 4
<<	Bitwise left shift	5 << 1	10
>>	Bitwise right shift	5 >> 1	2
>>>	Bitwise unsigned right shift	-5 >>> 1	Large positive number
&	Bitwise AND	5 & 1	1
!	Logical NOT	!true	false
??	Nullish coalescing (returns first non-null/undefined value)	null ?? 'default'	'default'
?:	Ternary (conditional)	x > 5 ? 'yes' : 'no'	'yes' or 'no'
+ (unary)	Converts operand to a number	+'5'	5
- (unary)	Negates a number	-5	-5
++	Increment	let x = 5; x++	x = 6
--	Decrement	let x = 5; x--	x = 4
<<	Bitwise left shift	5 << 1	10
>>	Bitwise right shift	5 >> 1	2
>>>	Bitwise unsigned right shift	-5 >>> 1	Large positive number
&	Bitwise AND	5 & 1	1

|	Bitwise OR	const x = 5; // Binary: 0101
		const y = 3; // Binary: 0011
		
		console.log(x | y); // 7 (Binary: 0111)
			Performs OR expression but in binary

^	Bitwise XOR	5 ^ 1	4
~	Bitwise NOT	~5	-6
=	Assignment operator	x = 10	10
+=	Addition assignment	x += 5	x = x + 5
-=	Subtraction assignment	x -= 5	x = x - 5
*=	Multiplication assignment	x *= 5	x = x * 5
/=	Division assignment	x /= 5	x = x / 5
%=	Modulus assignment	x %= 5	x = x % 5
**=	Exponentiation assignment	x **= 3	x = x ** 3
=>	Arrow function	const add = (a, b) => a + b	Defines a function

await	Pauses async function execution until a Promise resolves	await fetch(url)	Returns Promise result
new	Creates an instance of an object	new Date()	Instance of Date
super	Refers to the parent class in inheritance	super.method()	Calls parent method
this	Refers to the current object	this.property	Accesses object props
delete	Deletes a property from an object	delete obj.prop	Removes prop
in	Checks if a property exists in an object	'key' in obj	true or false
instanceof	Checks if an object is an instance of a class	obj instanceof Array	true or false
typeof	Returns the type of a variable	typeof 42	'number'
void	Evaluates expression but returns undefined	void 0	undefined
?.	Optional chaining operator	obj?.prop	Accesses prop safely