PERSEVERE

Student Guide to Coding

APIs and Microservices Certification (300 hours)

Introduction to Managing Packages with npm

The Node Package Manager (npm) is a command-line tool used by developers to share and control modules (or packages) of JavaScript code written for use with Node.js.

When starting a new project, npm generates a package.json file. This file lists the package dependencies for your project. Since npm packages are regularly updated, the package.json file allows you to set specific version numbers for each dependency. This ensures that updates to a package don't break your project.

npm saves packages in a folder named node_modules. These packages can be installed in two ways:

1. globally in a root node_modules folder, accessible by all projects.
2. locally within a project's own node_modules folder, accessible only to that project.

Most developers prefer to install packages local to each project to create a separation between the dependencies of different projects. Working on these challenges will involve you writing your code on Glitch on our starter project. After completing each challenge you can copy your public Glitch url (to the homepage of your app) into the challenge screen to test it! Optionally you may choose to write your project on another platform but it must be publicly visible for our testing.

How to Use package.json, the Core of Any Node.js Project or npm Package

The package.json file is the center of any Node.js project or npm package. It stores information about your project, similar to how the <head> section of an HTML document describes the content of a webpage. It consists of a single JSON object where information is stored in key-value pairs. There are only two required fields; "name" and "version", but it’s good practice to provide additional information about your project that could be useful to future users or maintainers.

If you look at the file tree of your project, you will find the package.json file on the top level of the tree. This is the file that you will be improving in the next couple of challenges.

One of the most common pieces of information in this file is the author field. It specifies who created the project, and can consist of a string or an object with contact or other details. An object is recommended for bigger projects, but a simple string like the following example will do for this project. *"author": "Jane Doe",*

Add a Description to Your package.json

The next part of a good package.json file is the description field; where a short, but informative description about your project belongs.

If you some day plan to publish a package to npm, this is the string that should sell your idea to the user when they decide whether to install your package or not. However, that’s not the only use case for the description, it’s a great way to summarize what a project does. It’s just as important in any Node.js project to help other developers, future maintainers or even your future self understand the project quickly.

Regardless of what you plan for your project, a description is definitely recommended. Here's an example:

```
"description": "A project that does something awesome",
```

Add Keywords to Your package.json

The keywords field is where you can describe your project using related keywords. Here's an example:

```
"keywords": [ "descriptive", "related", "words" ],
```

As you can see, this field is structured as an array of double-quoted strings.

Add a License to Your package.json

The license field is where you inform users of what they are allowed to do with your project.

Some common licenses for open source projects include MIT and BSD. License information is not required, and copyright laws in most countries will give you ownership of what you create by default. However, it’s always a good practice to explicitly state what users can and can’t do. Here's an example of the license field:

```
"license": "MIT",
```

Add a Version to Your package.json

A version is one of the required fields of your package.json file. This field describes the current version of your project. Here's an example:

```
"version": "1.2.0",
```

Expand Your Project with External Packages from npm

One of the biggest reasons to use a package manager is their powerful dependency management. Instead of manually having to make sure that you get all dependencies whenever you set up a project on a new computer, npm automatically installs everything for you. But how can npm know exactly what your project needs? Meet the dependencies section of your package.json file.

In this section, packages your project requires are stored using the following format:

```
"dependencies": {
  "package-name": "version",
  "express": "4.14.0"
}
```

Manage npm Dependencies By Understanding Semantic Versioning

Versions of the npm packages in the dependencies section of your package.json file follow what's called Semantic Versioning (SemVer), an industry standard for software versioning aiming to make it easier to manage dependencies. Libraries, frameworks or other tools published on npm should use SemVer in order to clearly communicate what kind of changes projects can expect if they update.

Knowing SemVer can be useful when you develop software that uses external dependencies (which you almost always do). One day, your understanding of these numbers will save you from accidentally introducing breaking changes to your project without understanding why things that worked yesterday suddenly don't work today. This is how Semantic Versioning works according to the official website:

```
"package": "MAJOR.MINOR.PATCH"
```

The MAJOR version should increment when you make incompatible API changes. The MINOR version should increment when you add functionality in a backwards-compatible manner. The PATCH version should increment when you make backwards-compatible bug fixes. This means that PATCHes are bug fixes and MINORs add new features but neither of them break what worked before. Finally, MAJORs add changes that won't work with earlier versions.

Use the Tilde-Character to Always Use the Latest Patch Version of a Dependency

In the last challenge, you told npm to only include a specific version of a package. That's a useful way to freeze your dependencies if you need to make sure that different parts of your project stay compatible with each other. But in most use cases, you don't want to miss bug fixes since they often include important security patches and (hopefully) don't break things in doing so.

To allow an npm dependency to update to the latest PATCH version, you can prefix the dependency's version with the tilde (~) character. Here's an example of how to allow updates to any 1.3.x version.

"package": "~1.3.8"

Use the Caret-Character to Use the Latest Minor Version of a Dependency

Similar to how the tilde we learned about in the last challenge allows npm to install the latest PATCH for a dependency; the caret (^) allows npm to install future updates as well. The difference is that the caret will allow both MINOR updates and PATCHes.

Your current version of moment should be "~2.10.2" which allows npm to install to the latest 2.10.x version. If you were to use the caret (^) as a version prefix instead, npm would be allowed to update to any 2.x.x version.

"package": "^1.3.8"
This would allow updates to any 1.x.x version of the package.

Remove a Package from Your Dependencies

You have now tested a few ways you can manage dependencies of your project by using the package.json's dependencies section. You have also included external packages by adding them to the file and even told npm what types of versions you want, by using special characters such as the tilde or the caret.

But what if you want to remove an external package that you no longer need? You might already have guessed it; just remove the corresponding key-value pair for that package from your dependencies.
This same method applies to removing other fields in your package.json as well

Basic Managing Packages with npm Review Questions

1. What does API stand for?

2. What does npm stand for?

3. What does SemVar stand for?

4. To allow an npm dependency to update to the latest PATCH version, you can prefix the dependency's version with the____________.

5. the_________allows npm to install future updates as well. The difference is that this will allow both MINOR updates and PATCHes.

6. npm saves packages in a folder named node_modules. These packages can be installed in two ways. What are the two ways?
1.

2.

7. What file is the center of any Node.js project or npm package?

8. Regardless of what you plan for your project, what description is definitely recommended?

9. Some common licenses for open source projects include

10. Write this out using a MIT license.

Introduction to Basic Node and Express

Node.js is a JavaScript runtime that allows developers to write backend (server-side) programs in JavaScript. Node.js comes with a handful of built-in modules - small, independent programs - that help facilitate this purpose. Some of the core modules include:

- HTTP: a module that acts as a server
- File System: a module that reads and modifies files
- Path: a module for working with directory and file paths
- Assertion Testing: a module that checks code against prescribed constraints

Express, while not included with Node.js, is another module often used with it. Express runs between the server created by Node.js and the frontend pages of a web application. Express also handles an application's routing. Routing directs users to the correct page based on their interaction with the application. While there are alternatives to using Express, its simplicity makes it a good place to begin when learning the interaction between a backend powered by Node.js and the frontend.

Meet the Node Console

During the development process, it is important to be able to check what’s going on in your code. Node is just a JavaScript environment. Like client side JavaScript, you can use the console to display useful debug information. On your local machine, you would see the console output in a terminal. On Glitch you can open the logs in the lower part of the screen. You can toggle the log panel with the button ‘Logs’ (lower-left, inside the tools menu).

We recommend you to keep the log panel open while working at these challenges. By reading the logs, you can be aware of the nature of errors that may occur.

Start a Working Express Server

In the first two lines of the file myApp.js, you can see how easy it is to create an Express app object. This object has several methods, and you will learn many of them in these challenges. One fundamental method is app.listen(port). It tells your server to listen on a given port, putting it in running state. You can see it at the bottom of the file. It is inside comments because, for testing reasons, we need the app to be running in the background. All the code that you may want to add goes between these two fundamental parts. Glitch stores the port number in the environment variable process.env.PORT. Its value is 3000. Let’s serve our first string! In Express, routes take the following structure: app.METHOD(PATH, HANDLER). METHOD is an http method in lowercase. PATH is a relative path on the server (it can be a string, or even a regular expression). HANDLER is a function that Express calls when the route is matched.

Handlers take the form function(req, res) {...}, where req is the request object, and res is the response object. For example, the handler

```
function(req, res) {
  res.send('Response String');
}
```
will serve the string 'Response String'.

Serve an HTML File

You can respond to requests with a file using the res.sendFile(path) method. You can put it inside the app.get('/', ...) route handler. Behind the scenes, this method will set the appropriate headers to instruct your browser on how to handle the file you want to send, according to its type. Then it will read and send the file. This method needs an absolute file path. We recommend you to use the Node global variable __dirname to calculate the path like this:

```
absolutePath = __dirname + relativePath/file.ext
```

Serve Static Assets

An HTML server usually has one or more directories that are accessible by the user. You can place the static assets there needed by your application (stylesheets, scripts, images). In Express, you can put in place this functionality using the middleware express.static(path), where the path parameter is the absolute path of the folder containing the assets. If you don't know what middleware is... don't worry, we will discuss it in detail later. Basically, middleware are functions that intercept route handlers, adding some kind of information. A middleware needs to be mounted using the method app.use (path, middlewareFunction). The first path argument is optional. If you don't pass it, the middleware will be executed for all requests.

Serve JSON on a Specific Route

While an HTML server serves (you guessed it!) HTML, an API serves data. A REST (REpresentational State Transfer) API allows data exchange in a simple way, without the need for clients to know any detail about the server. The client only needs to know where the resource is (the URL), and the action it wants to perform on it (the verb). The GET verb is used when you are fetching some information, without modifying anything. These days, the preferred data format for moving information around the web is JSON. Simply put, JSON is a convenient way to represent a JavaScript object as a string, so it can be easily transmitted.

Let's create a simple API by creating a route that responds with JSON at the path /json. You can do it as usual, with the app.get() method. Inside the route handler, use the method res.json(), passing in an object as an argument. This method closes the request-response loop, returning the data. Behind the scenes, it converts a valid JavaScript object into a string, then sets the appropriate headers to tell your browser that you are serving JSON, and sends the data back. A valid object has the usual structure {key: data}; data can be a number, a string, a nested object or an array; data can also be a variable or the result of a function call, in which case it will be evaluated before being converted into a string.

Use the .env File

The .env file is a hidden file that is used to pass environment variables to your application. This file is secret. No one but you can access it, and it can be used to store data that you want to keep private or hidden. For example, you can store API keys from external services or your database URI. You can also use it to store configuration options. By setting configuration options, you can change the behavior of your application, without the need to rewrite some code.

The environment variables are accessible from the app as process.env.VAR_NAME. The process.env object is a global Node object, and variables are passed as strings. By convention, the variable names are all uppercase, with words separated by an underscore. The .env is a shell file, so you don’t need to wrap names or values in quotes. It is also important to note that there cannot be space around the equals sign when you are assigning values to your variables, e.g. VAR_NAME=value. Usually, you will put each variable definition on a separate line.

Implement a Root-Level Request Logger Middleware

Earlier, you were introduced to the express.static() middleware function. Now it’s time to see what middleware is in more detail. Middleware functions are functions that take 3 arguments: the request object, the response object, and the next function in the application’s request-response cycle. These functions execute some code that can have side effects on the app, and usually add information to the request or response objects. They can also end the cycle by sending a response when some condition is met. If they don’t send the response when they are done, they start the execution of the next function in the stack. This triggers calling the 3rd argument, next().

Look at the following example:

```
function(req, res, next) {
  console.log("I'm a middleware...");
  next();
}
```

Let’s suppose you mounted this function on a route. When a request matches the route, it displays the string “I’m a middleware…”, then it executes the next function in the stack. In this exercise, you are going to build root-level middleware. As you have seen in challenge 4, to mount a middleware function at root level, you can use the app.use(<mware-function>) method. In this case, the function will be executed for all the requests, but you can also set more specific conditions. For example, if you want a function to be executed only for POST requests, you could use app.post(<mware-function>). Analogous methods exist for all the HTTP verbs (GET, DELETE, PUT, …).

Chain Middleware to Create a Time Server

Middleware can be mounted at a specific route using app.METHOD(path, middlewareFunction). Middleware can also be chained inside route definition.

Look at the following example:

```
app.get('/user', function(req, res, next) {
  req.user = getTheUserSync();  // Hypothetical synchronous operation
  next();
}, function(req, res) {
  res.send(req.user);
});
```

This approach is useful to split the server operations into smaller units. That leads to a better app structure, and the possibility to reuse code in different places. This approach can also be used to perform some validation on the data. At each point of the middleware stack you can block the execution of the current chain and pass control to functions specifically designed to handle errors. Or you can pass control to the next matching route, to handle special cases. We will see how in the advanced Express section.

Get Route Parameter Input from the Client

When building an API, we have to allow users to communicate to us what they want to get from our service. For example, if the client is requesting information about a user stored in the database, they need a way to let us know which user they're interested in. One possible way to achieve this result is by using route parameters. Route parameters are named segments of the URL, delimited by slashes (/). Each segment captures the value of the part of the URL which matches its position. The captured values can be found in the req.params object.

```
route_path: '/user/:userId/book/:bookId'
actual_request_URL: '/user/546/book/6754'
req.params: {userId: '546', bookId: '6754'}
```

Get Query Parameter Input from the Client

Another common way to get input from the client is by encoding the data after the route path using a query string. The query string is delimited by a question mark (?), and includes field=value couples. Each couple is separated by an ampersand (&). Express can parse the data from the query string, and populate the object req.query. Some characters, like the percent (%), cannot be in URLs and have to be encoded in a different format before you can send them. If you use the API from JavaScript, you can use specific methods to encode/decode these characters.

```
route_path: '/library'
actual_request_URL: '/library?userId=546&bookId=6754'
req.query: {userId: '546', bookId: '6754'}
```

Use body-parser to Parse POST Request

Besides GET, there is another common HTTP verb, it is POST. POST is the default method used to send client data with HTML forms. In REST convention, POST is used to send data to create new items in the database (a new user, or a new blog post). You don’t have a database in this project, but you are going to learn how to handle POST requests anyway.

In these kinds of requests, the data doesn’t appear in the URL, it is hidden in the request body. This is a part of the HTML request, also called payload. Since HTML is text-based, even if you don’t see the data, it doesn’t mean that it is secret. The raw content of an HTTP POST request is shown below:

```
POST /path/subpath HTTP/1.0
From: john@example.com
User-Agent: someBrowser/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 20
name=John+Doe&age=25
```

As you can see, the body is encoded like the query string. This is the default format used by HTML forms. With Ajax, you can also use JSON to handle data having a more complex structure. There is also another type of encoding: multipart/form-data. This one is used to upload binary files. In this exercise, you will use a urlencoded body. To parse the data coming from POST requests, you have to install the body-parser package. This package allows you to use a series of middleware, which can decode data in different formats.

Get Data from POST Request

Mount a POST handler at the path /name. It’s the same path as before. We have prepared a form in the html frontpage. It will submit the same data of exercise 10 (Query string). If the body-parser is configured correctly, you should find the parameters in the object req.body. Have a look at the usual library example:

route: POST '/library'
urlencoded_body: userId=546&bookId=6754
req.body: {userId: '546', bookId: '6754'}

Respond with the same JSON object as before: {name: 'firstname lastname'}. Test if your endpoint works using the html form we provided in the app frontpage.

Tip: There are several other http methods other than GET and POST. And by convention there is a correspondence between the http verb, and the operation you are going to execute on the server. The conventional mapping is:

POST (sometimes PUT) - Create a new resource using the information sent with the request,

GET - Read an existing resource without modifying it,

PUT or PATCH (sometimes POST) - Update a resource using the data sent,

DELETE => Delete a resource.
There are also a couple of other methods which are used to negotiate a connection with the server. Except from GET, all the other methods listed above can have a payload (i.e. the data into the request body). The body-parser middleware works with these methods as well.

Basic Node and Express Review Questions

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

Introduction to MongoDB and Mongoose

MongoDB is a database that stores data records (documents) for use by an application. Mongo is a non-relational "NoSQL" database. This means Mongo stores all associated data within one record instead of storing it across many preset tables as in a SQL database. Some benefits of this storage model are:

- Scalability: by default, non-relational databases are split (or "shared") across many systems instead of only one. This makes it easier to improve performance at a lower cost.
- Flexibility: new datasets and properties can be added to a document without the need to make a new table for that data.
- Replication: copies of the database run in parallel so if one goes down, one of the copies becomes the new primary data source.

While there are many non-relational databases, Mongo's use of JSON as its document storage structure makes it a logical choice when learning backend JavaScript. Accessing documents and their properties is like accessing objects in JavaScript.

Mongoose.js is an npm module for Node.js that allows you to write objects for Mongo as you would in JavaScript. This can make it easier to construct documents for storage in Mongo.

Working on these challenges will involve you writing your code on Glitch on our starter project. After completing each challenge you can copy your public glitch url (to the homepage of your app) into the challenge screen to test it! Alternatively, you may choose to write your project on another platform, but it must be publicly visible for our testing.

Start this project on Glitch using this link or clone this repository on GitHub! If you use Glitch, remember to save the link to your project somewhere safe!

Use MongoDB Atlas to host a free mongodb instance for your projects

For the following challenges, we are going to start using MongoDB to store our data. To simplify the configuration, we are going to use MongoDB Atlas.

MongoDB Atlas is a MongoDB Database-as-a-Service platform, which basically means that they configure and host the database for you, making it so that the only responsibility you have is to populate your database with what matters: data! We are going to show you how to:

- Create a MongoDB Atlas account.
- Create a new cluster.
- Create a new user on the database.
- Whitelist your IP address.
- Connect to your cluster.

Create a MongoDB Atlas account

Let's start by going to MongoDB Atlas.

Once you open the MongoDB Atlas page, you should sign up for a new account.

- Click the Sign In button in the top right corner to open the registration page.
- Click the Register for a new account link at the bottom of the sign in page.
- Fill the registration form with your information and press continue.
- You should now be logged into your new account and see a modal with a green “Build my first cluster” button; click on it.

Create a new cluster

- Go through the steps of building your first cluster by following the instructions they provide and clicking next after each step.

    - **Choose your cloud provider and region,** you can leave this as the default provided (typically AWS).
    - **Customize your cluster's specs,** you can also leave this as the default provided, M0 Sandbox (Shared RAM, 512 MB Storage) Encrypted.
    - **Give your cluster a name**, you can also leave this as the default provided, Cluster 0.
- Now click the green **Create Cluster** button at the bottom of the screen and verify the image captions they provide.
- You should now see the message. Your cluster is being created - New clusters take between 7-10 minutes to provision. Wait until the cluster is created before going to the next step.

Create a new user on the database

- You should be able to see the green **Get Started** button on the bottom left of your screen; you can click this button to see at which step of the process you are in. If you click on it now, you can see the next step is to **Create your first database user**. Go ahead and click on that step.

    - Follow the instructions by clicking on the Security tab.
    - Click on the green **ADD NEW USER** button.
    - Enter a user name and password and then select **Read or write to any database** under user privileges, remember to store your username and password somewhere safe.
    - Click on the **ADD USER** green button in the bottom right of the modal.

Note: You can always upgrade your privileges to the Admin level; however, it is best practice to give permissions to a user on an as-needed basis for security reasons.

Whitelist your IP address

- If you now click on the green **Get Started** button in the bottom left of your screen, you should see the next step to take highlighted, **Whitelist your IP address**, click on it.

    - Follow the instructions by clicking on the IP Whitelist tab under the Security tab.

    - Click on the green **ADD IP ADDRESS** button.
    - In the modal, click the **ALLOW ACCESS FROM ANYWHERE** button and you should see 0.0.0.0/0 pre-filled for the whitelist entry field, click the green **Confirm** button.

Connect to your cluster

- Clicking on the green **Get Started** button in the bottom left of your screen should now show you the final step, **Connect to your cluster**. Click on it.

    - Follow the instructions by clicking on the Connect button in the Sandbox section.
    - In the pop-up modal, click on **Connect Your Application**, a connection string will be displayed, you can copy that connection string by clicking on the copy button.
    - This will be the final URI that you will use to connect to your db, it will look something like this mongodb+srv://<user>:<password>@<cluster#-dbname>.mongodb.net/test?retryWrites=true, notice that the user and cluster#-dbname fields are already filled out for you, all you would need to replace is the password field with the one that you created in the previous step.
- That's it! You now have the URI you will add to your application to connect to your database. Keep this URI safe somewhere, so you can use it later!
- Feel free to create separate databases for different applications if they need a separate database. You just need to create a new project under your current MongoDB Atlas account, build a new cluster, add a new user, whitelist your IP addresses and finally connect to your cluster to obtain the new URI.

Install and Set Up Mongoose

Add mongodb and mongoose to the project's package.json. Then require mongoose. Store your MongoDB Atlas database URI in the private .env file as MONGO_URI. Surround the the URI with single or double quotes and make sure no space exists between both the variable and the `=` and the value and `=`. Connect to the database using the following syntax:

mongoose.connect(<Your URI>, { useNewUrlParser: true, useUnifiedTopolog

Create a Model

CRUD Part I - CREATE

First of all we need a Schema. Each schema maps to a MongoDB collection. It defines the shape of the documents within that collection. Schemas are a building block for Models. They can be nested to create complex models, but in this case we'll keep things simple. A model allows you to create instances of your objects, called documents.

Glitch is a real server, and in real servers the interactions with the db happen in handler functions. These functions are executed when some event happens (e.g. someone hits an endpoint on your API). We'll follow the same approach in these exercises. The done() function is a callback that tells us that we can proceed after completing an asynchronous operation such as inserting, searching, updating or deleting. It's following the Node convention and should be called as done(null, data) on success, or done(err) on error. Warning - When interacting with remote services, errors may occur!

```
/* Example */
var someFunc = function(done) {
  //... do something (risky) ...
  if(error) return done(error);
  done(null, result); };
```

Create Many Records with model.create( )

Sometimes you need to create many instances of your models, e.g. when seeding a database with initial data. Model.create() takes an array of objects like [{name: 'John', ...}, {...}, ...] as the first argument, and saves them all in the db.

Use model.find( ) to Search Your Database

Find all the people having a given name, using Model.find() -> [Person]

In its simplest usage, Model.find() accepts a query document (a JSON object) as the first argument, then a callback. It returns an array of matches. It supports an extremely wide range of search options. Check it in the docs. Use the function argument personName as a search key.

Use model.findOne( ) to Return a Single Matching Document from Your Database

Model.findOne() behaves like .find(), but it returns only one document (not an array), even if there are multiple items. It is especially useful when searching by properties that you have declared as unique.

Use model.findById( ) to Search Your Database By -id

When saving a document, mongodb automatically adds the field _id, and sets it to a unique alphanumeric key. Searching by _id is an extremely frequent operation, so mongoose provides a dedicated method for it.

Perform Classic Updates by Running Find, Edit, then Save

In the good old days this was what you needed to do if you wanted to edit a document and be able to use it somehow, e.g. sending it back in a server response. Mongoose has a dedicated updating method : Model.update(). It is bound to the low-level mongo driver. It can bulk edit many documents matching certain criteria, but it doesn't send back the updated document, only a 'status' message. Furthermore, it makes model validations difficult, because it just directly calls the mongo driver.

Perform New Updates on a Document Using model.findAndUpdate( )

Recent versions of mongoose have methods to simplify documents updating. Some more advanced features (i.e. pre/post hooks, validation) behave differently with this approach, so the Classic method is still useful in many situations. findByIdAndUpdate() can be used when searching by Id.

Delete One Document Using model.findByIdAndRemove

Delete one person by the person's _id. You should use one of the methods findByIdAndRemove() or findOneAndRemove(). They are like the previous update methods. They pass the removed document to the cb. As usual, use the function argument personId as the search key.

Delete Many Documents with model.remove( )

Model.remove() is useful to delete all the documents matching given criteria.

Chain Search Query Helpers to Narrow Search Results

If you don't pass the callback as the last argument to Model.find() (or to the other search methods), the query is not executed. You can store the query in a variable for later use. This kind of object enables you to build up a query using chaining syntax. The actual db search is executed when you finally chain the method .exec(). You always need to pass your callback to this last method. There are many query helpers. Here we'll use the most 'famous' ones.

Basic MongoDB and Mongoose Review Questions

1. What is MongoDB?

2. What is MongoDB Atlas?

3. The___________function is a callback that tells us that we can proceed after completing an asynchronous operation such as inserting, searching, updating or deleting.

4. It's following the Node convention and should be called as__________ on success, or____________ on error.

5. ____________ takes an array of objects like [{name: 'John', ...}, {...}, ...] as the first argument, and saves them all in the db.

6. ____________accepts a query document (a JSON object) as the first argument, then a callback.

7. When saving a document, mongodb automatically adds the field _id, and sets it to a unique alphanumeric key. Searching by _id is an extremely frequent operation, so mongoose provides a dedicated method for it. What is this method?

8. Mongoose has a dedicated updating method. What is it?

9. What does the model.update( ) send back?

10. Perform New Updates on a Document Using the ____________.