NODE.JS

Part III

WRITING A CHAT APPLICATION

- Create a package.json file to store all the dependency

```
{
"name": "socket-chat-example",
"version": "0.0.1",
"description": "my first socket.io app",
"dependencies": {}
}
```

- dependencies will be populated automatically in the json file and packages will be installed in a sub directory called node_modules
- `npm install express`
- sample index.js to create a server
- `npm install -g nodemon`

SOCKET.IO

- ❑Express initializes app to be a function handler that you can supply to an HTTP server
- ❑We define a route handler / that gets called when we hit our website home.
- ❑We make the http server listen on port 3000.

```
const express = require('express');
const app = express();
const http = require('http');
const server = http.createServer(app);

app.get('/', (req, res) => {
  res.send('<h1>Hello world</h1>');
});

server.listen(3000, () => {
  console.log('listening on *:3000');
});
```

https://socket.io/get-started/chat#:~:text=The%20main%20idea%20behind%20Socket,binary%20data%20is%20supported%20too.

SOCKET.IO

```
app.get('/', (req, res) => {
res.sendFile(__dirname + '/index.html');
});
```

```
<html>
<head>Add CSS</head>
<body>
<ul id="messages"></ul>
<form id="form" action="">
<input id="input" autocomplete="off"
/><button>Send</button>
</form>
</body>
</html>
```

- ❑ Socket.IO is composed of two parts:
- ❑ A server that integrates with (or mounts on) the Node.JS HTTP Server socket.io
- ❑ A client library that loads on the browser side socket.io-client
- ❑ To indicate connected users the index.js is updated as follows.

```
io.on('connection', (socket) => {
console.log('a user connected');
});
```

```
const express =
require('express');
const app = express();
const http = require('http');
const server =
http.createServer(app);
const { Server } =
require("socket.io");
const io = new Server(server);
```

A new instance of socket.io by passing the server (the HTTP server) object

CLIENT AND SERVER

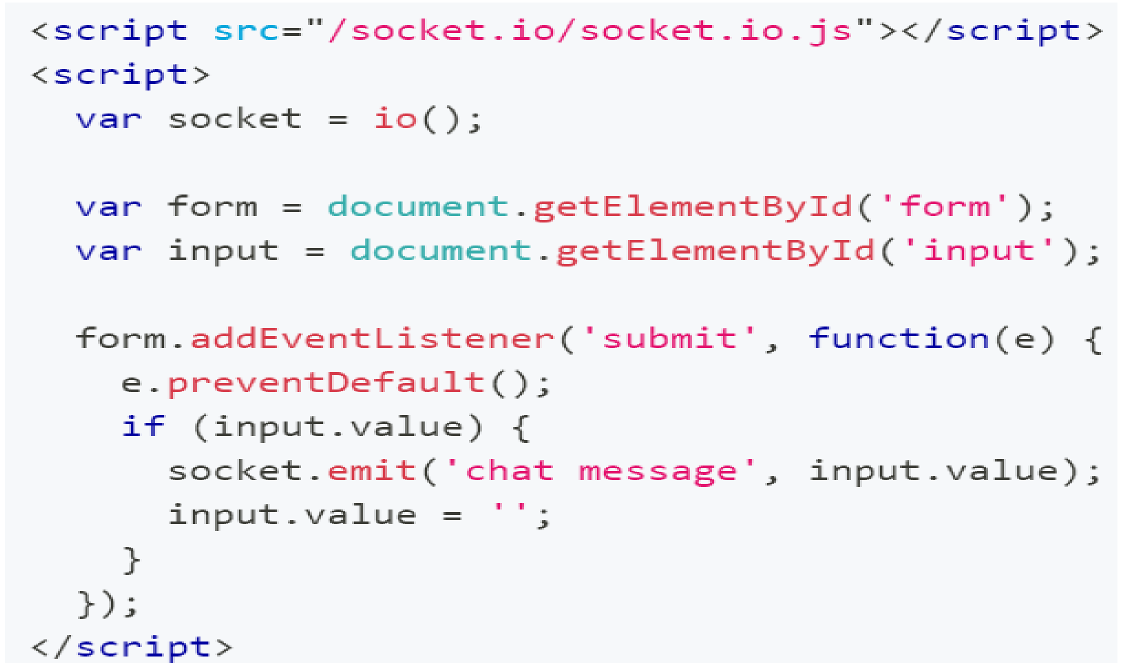
```
<script src="/socket.io/socket.io.js"></script>
<script>
var socket = io();
</script>
```

- ❑ index.html should be modified to include the io client
- ❑ If you would like to use the local version of the client-side JS file, you can find it at node_modules/socket.io/client-dist/socket.io.js.
- ❑ No URLs are specified when io() is called, since it defaults to trying to connect to the host that serves the page
- ❑ loading socket.io client from content delivery network
  - ❑`<script src="https://cdn.socket.io/4.5.0/socket.io.min.js" integrity="…" crossorigin="anonymous"></script>`
- ❑ events can be added for individual sockets

```
io.on('connection', (socket) => {
console.log('a user connected');
socket.on('disconnect', () => {
console.log('user disconnected');
});
});
```

CLIENT SENDS DATA

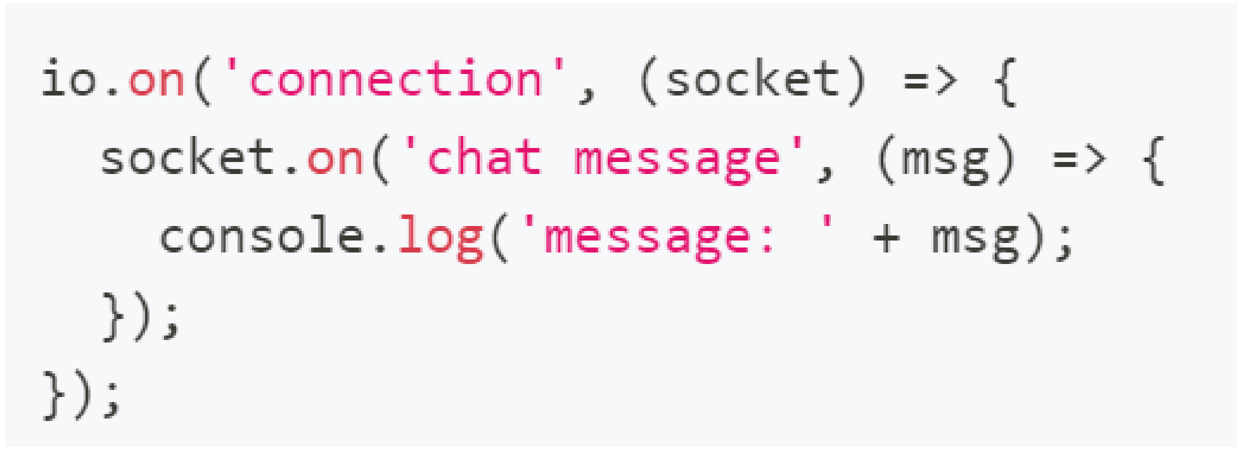
- ❑ The main idea behind Socket.IO is that you can send and receive any events you want, with any data you want. Any objects that can be encoded as JSON will do, and binary data is supported too.
- ❑ Let’s make it so that when the user types in a message, the server gets it as a chat message event



```
<script src="/socket.io/socket.io.js"></script>
<script>
  var socket = io();

  var form = document.getElementById('form');
  var input = document.getElementById('input');

  form.addEventListener('submit', function(e) {
    e.preventDefault();
    if (input.value) {
      socket.emit('chat message', input.value);
      input.value = '';
    }
  });
</script>
```



```
io.on('connection', (socket) => {
  socket.on('chat message', (msg) => {
    console.log('message: ' + msg);
  });
});
```

});

BROADCAST A MESSAGE

```
 io.emit('some event', { someProperty: 'some value',
otherProperty: 'other value' });
```

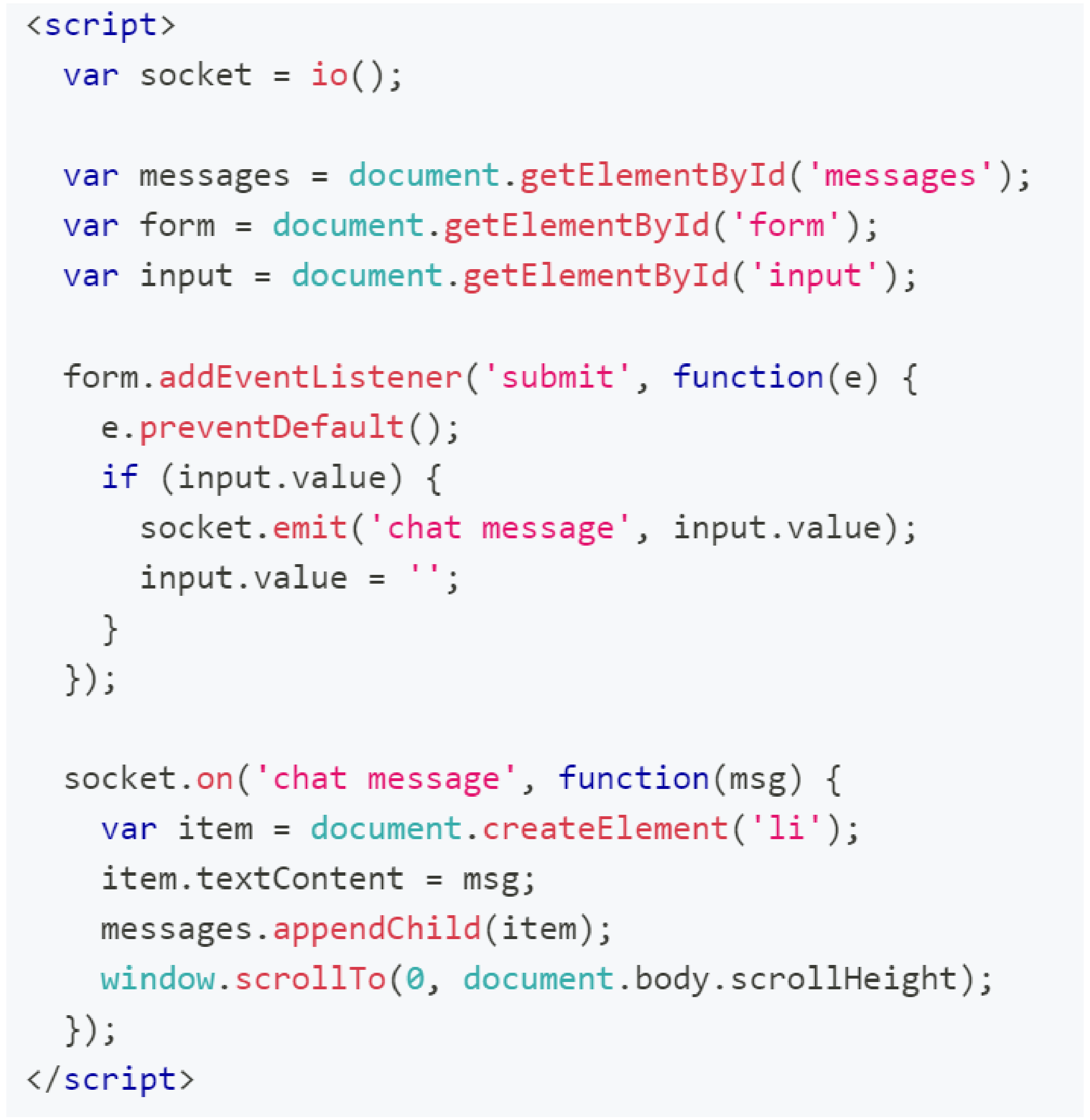
- ❑Broadcasts a data to everyone including the sender
- ❑If you want to send a message to everyone except for a certain emitting socket, we have the broadcast flag for emitting from that socket:

```
io.on('connection', (socket) => {
socket.broadcast.emit('hi');
```

To broadcast the received message, at index.js the following is included

```
io.on('connection', (socket) => {
  socket.on('chat message', (msg) => {
    io.emit('chat message', msg);
  });
});
```

on the client side when we capture a chat message event we'll include it in the page



```
<script>
  var socket = io();

  var messages = document.getElementById('messages');
  var form = document.getElementById('form');
  var input = document.getElementById('input');

  form.addEventListener('submit', function(e) {
    e.preventDefault();
    if (input.value) {
      socket.emit('chat message', input.value);
      input.value = '';
    }
  });

  socket.on('chat message', function(msg) {
    var item = document.createElement('li');
    item.textContent = msg;
    messages.appendChild(item);
    window.scrollTo(0, document.body.scrollHeight);
  });
</script>
```

https://socket.io/get-started/chat

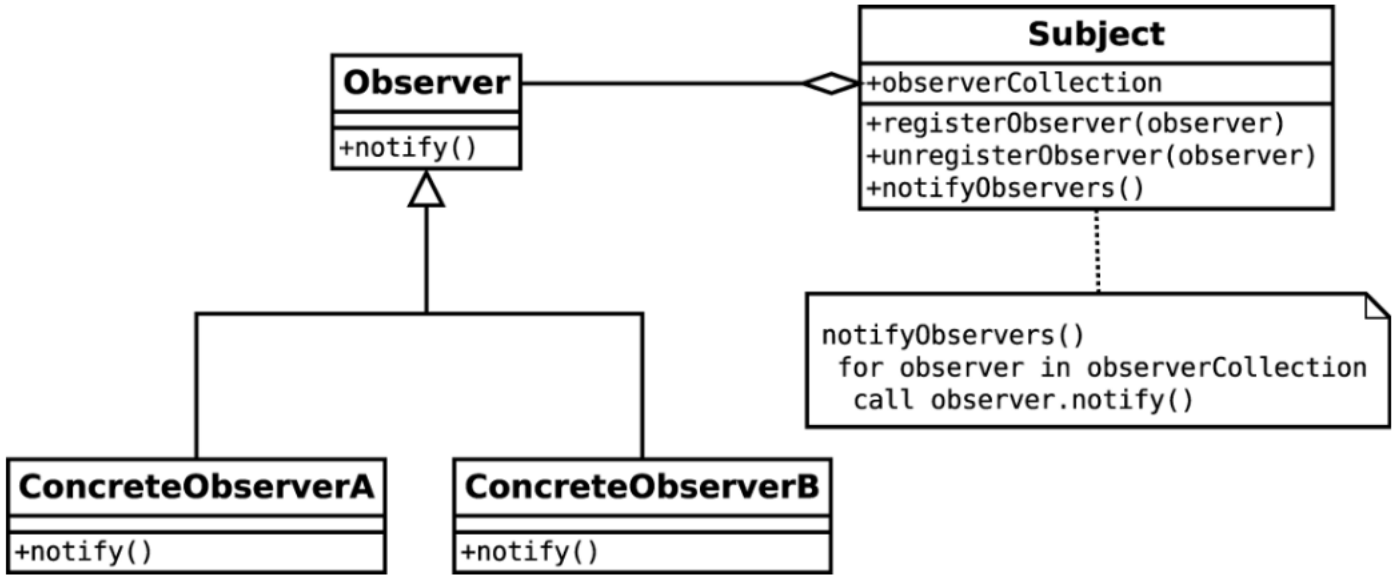
THE OBSERVER PATTERN

- ❑Together with reactor, callbacks, and modules, this is one of the pillars of the platform and an absolute prerequisite for using many node-core and userland modules
- ❑Pattern (observer): defines an object (called subject), which can notify a set of observers (or listeners), when a change in its state happens.
- ❑The main difference from the callback pattern is that the subject can actually notify multiple observers, while a traditional continuation-passing style callback will usually propagate its result to only one listener, the callback
- ❑ No need to define interfaces as required in Java to implement this pattern

The observer pattern is already built into the core and is available through the **EventEmitter** class.

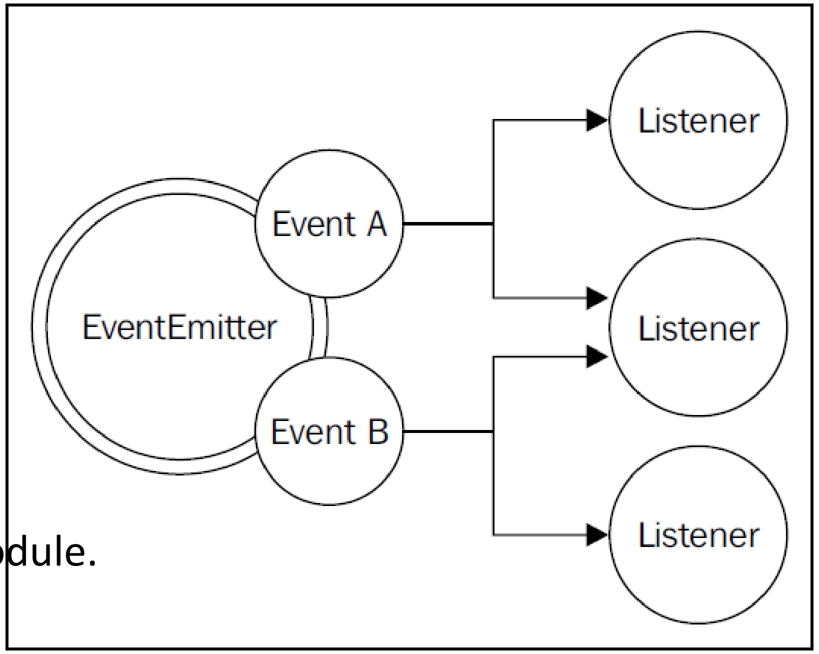
The `EventEmitter` class allows us to register one or more functions as listeners, which will be invoked when a particular event type is fired

THE OBSERVER PATTERN



```
notifyObservers()
 for observer in observerCollection
  call observer.notify()
```

EVENT EMITTER



- The `EventEmitter` is a prototype, and it is exported from the `events` core module.
- `Obtaining a reference of the EventEmitter`
  - `var EventEmitter = require('events').EventEmitter;`
  - `var eeInstance = new EventEmitter();`
- The essential methods of the `EventEmitter` are given as follows.
  - `on(event, listener)`: This method allows you to register a new listener (a function) for the given event type (a string)
  - `once(event, listener)`: This method registers a new listener, which is then removed after the event is emitted for the first time
  - `emit(event, [arg1], […])`: This method produces a new event and provides additional arguments to be passed to the listeners
  - `removeListener(event, listener)`: This method removes a listener for the specified event type

EVENT EMITTERS

- ❑All the preceding methods will return the `EventEmitter` instance to allow chaining.
- ❑ there is a big difference between a listener and a traditional Node.js callback; in particular, the first argument is not an error, but it can be any data passed to emit() at the moment of its invocation.
- ❑The `listener` function has the signature, `function([arg1], […])`, so it simply accepts the arguments provided the moment the event is emitted.
- ❑Inside the listener, `this` refers to the instance of the `EventEmitter` that produces the event

EMITTING EVENTS

- The `EventEmitter` - as it happens for callbacks - cannot just throw exceptions when an error condition occurs, as they would be lost in the event loop if the event is emitted asynchronously.
- Instead, the convention is to emit a special event, called `error`, and to pass an `Error` object as an argument.
- A common dilemma when defining an asynchronous API is to check whether to use an `EventEmitter` or simply accept a callback.
- The general differentiating rule is semantic: callbacks should be used when a result must be returned in an asynchronous way; events should instead be used when there is a need to communicate that something has just happened

REQUIRE SYNCHRONY

- ❑ The essential concept to remember is that everything inside a module is private unless it's assigned to the `module.exports` variable.
- ❑ The contents of this variable are then cached and returned when the module is loaded using `require()`.

- ❑ `require` function is synchronous.
- ❑ In fact, it returns the module contents using a simple direct style, and no callback is required
- ❑ In its early days, Node.js used to have an asynchronous version of `require()`, but it was soon removed because it was overcomplicating a functionality that was actually meant to be used only at initialization time, and where asynchronous I/O brings more complexities than advantages.
- ❑ The term *dependency hell*, describes a situation whereby the dependencies of a software, in turn depend on a shared dependency, but require different incompatible versions.
- ❑ Node.js solves this problem elegantly by loading a different version of a module depending on where the module is loaded from.
- ❑ All the merits of this feature go to `npm` and also to the resolving algorithm used in the `require` function

RESOLVING DEPENDENCIES

- The `resolve()` function takes a module name (which we will call here, `moduleName`) as input and it returns the full path of the module.
- This path is then used to load its code and also to identify the module uniquely.
- The resolving algorithm can be divided into the following three major branches:
- • **File modules**: If `moduleName` starts with "`/`" it's considered already an absolute path to the module and it's returned as it is. If it starts with "`./`", then `moduleName` is considered a relative path, which is calculated starting from the requiring module.
- • **Core modules**: If `moduleName` is not prefixed with "`/`" or "`./`", the algorithm will first try to search within the core Node.js modules.
- • **Package modules**: If no core module is found matching `moduleName`, then the search continues by looking for a matching module into the first `node_modules` directory that is found navigating up in the directory structure starting from the requiring module.

RESOLVING DEPENDENCIES

- The algorithm continues to search for a match by looking into the next `node_modules` directory up in the directory tree, until it reaches the root of the filesystem.
- The resolving algorithm is applied transparently for us when we invoke require(); however, if needed, it can still be used directly by any module by simply invoking require.resolve().

- myApp, depB, and depC all depend on depA; however,they all have their own private version of the dependency

- Calling `require('depA')` from `/myApp/foo.js` will load `/myApp/node_modules/depA/index.js`

- Calling `require('depA')` from `/myApp/node_modules/depB/bar.js` will load `/myApp/node_modules/depB/node_modules/depA/index.js`

```
myApp
├── foo.js
└── node_modules
    ├── depA
    │   └── index.js
    ├── depB
    │   ├── bar.js
    │   └── node_modules
    │       └── depA
    │           └── index.js
    └── depC
        ├── foobar.js
        └── node_modules
            └── depA
                └── index.js
```

CONNECTING A DATABASE

- ❑MongoDB is a NoSQL database used to store large amounts of data without any traditional relational database table.
- ❑Instead of rows & columns, MongoDB used collections & documents to store data.
- ❑A collection consists of a set of documents & a document consists of key-value pairs which are the basic unit of data in MongoDB

CONNECTING TO A DATABASE

Step 1: Initialize npm on the directory and install the necessary modules. Also, create the index file

```
$ npm i express mongoose
```

Step 2: Initialise the express app and make it listen to a port on localhost.

```
const express = require("express");

const app = express();

app.listen(3000, () => console.log("Server is
running"));
```

- ❑To connect a Node.js application to MongoDB, we have to use a library called **Mongoose**
- ❑`const mongoose = require("mongoose");`
- ❑Connect method is invoked with URL and user credentials

```
mongoose.connect("mongodb://localhost:27017/newCollection
", {

      useNewUrlParser: true,

      useUnifiedTopology: true

});
```

CONFIGURING THE DATABASE

A schema is defined

A schema is a structure, that gives information about how the data is being stored in a collection.

```
const contactSchema = {
email: String,
query: String,
};
```

Then we have to create a model using that schema which is then used to store data in a document as objects

```
const Contact = mongoose.model("Contact", contactSchema);
```

When you call mongoose.model() on a schema, Mongoose compiles a model for you.

The first argument is the singular name of the collection your model is for.

Mongoose automatically looks for the plural, lowercased version of your model name.

```
Contact.create({ query: 'small' }, function (err, small) {
  if (err) return handleError(err);
  // saved!
});

// or, for inserting large batches of documents
Contact.insertMany([{ query: 'small' }], function(err) {

});
```

CRUD

Contact.find()

Contact.delete;

Contact.update();

```
async function run() {
// Create a new mongoose model
const personSchema = new mongoose.Schema({
name: String });
const Person = mongoose.model('Person',
personSchema);
// Create a change stream. The 'change'
event gets emitted when there's a // change
in the database
Person.watch(). on('change', data =>
console.log(new Date(), data));
```

STORING DATA

we are able to store data in our document

```
app.post("/contact", function (req, res) {
    console.log(req.body.email);
const contact = new Contact({
    email: req.body.email,
    query: req.body.query,
});
```

```
contact.save(function (err) {
    if (err) {
        throw err;
    } else {
        res.render("contact");
    }
});
```

MONGO DB CONNECTION

By using Mongoose we’re able to access the MongoDB database in an object-oriented way.

This means that we need to add a Mongoose schema

MongoDB's collections, by default, do not require their documents to have the same schema. That is:

- The documents in a single collection do not need to have the same set of fields and the data type for a field can differ across documents within a collection.
- To change the structure of the documents in a collection, such as add new fields, remove existing fields, or change the field values to a new type, update the documents to the new structure.

This flexibility facilitates the mapping of documents to an entity or an object. Each document can match the data fields of the represented entity, even if the document has substantial variation from other documents in the collection.

QUERYING THE DATABASE

```
 app.get("/show",
function(req,res) {

     Contact.find(function(err,
contacts) {

        if (err) {

            console.log(err);

        } else {

            res.json(contacts);

        }

     });

 });
```

```
app.get('/:id', function(req, res)
{
    let id = req.params.id;
    Contact.findById(id,
function(err, contact) {
        res.json(contact);
    });
});
```

STORING FORM DATA WITH MULTIPLE FIELDS

```
app.post("/contact", function (req, res) {
     console.log(req.body.email);
const contact = new Contact({
     email: req.body.email,
     query: req.body.query,
});
```

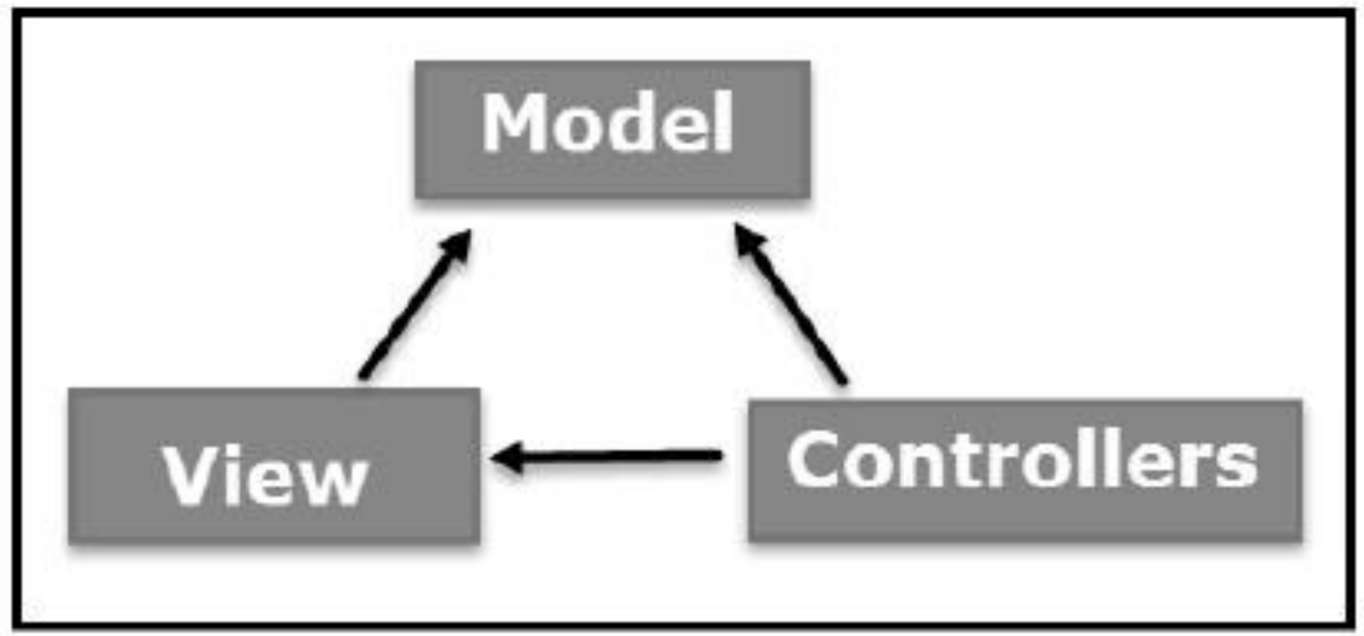
contact = new Contact(req.body);

```
contact.save(function (err) {
    if (err) {
        throw err;
    } else {
        res.render("contact");
    }
});
```

STORING FORM DATA WITH MULTIPLE FIELDS

```
 let contact = new Contact(req.body);
 contact.save()
 .then(contact => {
            res.status(200).json({'contact': 'contact added
successfully'});
        })
        .catch(err => {
            res.status(400).send('adding new contact failed');
        });
```

SOFTWARE DESIGN PATTERNS



MODEL VIEW CONTROLLER

It is an architecture that aims to separate out the data (model), the display (view) and the application logic (controller).

1. A request comes into your application
2. The request gets routed to a controller
3. The controller, if necessary, makes a request to the model
4. The model responds to the controller
5. The controller sends a response to a view
6. The view sends a response to the original requester

HAPPY HOLIDAYS