HANDLING POST REQUEST

POST

- Request method
- Request asks the server to accept/store data which is enclosed in the body of the request.
- Often used when submitting a form

POST

- app.post
- Form – method= ‘post’ action=destination.
- The rest of the work is in .js file
- Handle the post request
- When request object is used we require additional middleware for handling such post request.

POST

- Body-parser – npm body parser, which helps the npm to carry out many functionalities.
- Parses incoming request bodies in a middleware before handler.
- Available under req.body.
- Check npm-body parser.

BODY-PARSER

- Express-route specific
  - When posting in a specific route.
  - Eg:/contact
  - Invoke body parser

BODY-PARSER

- Var bodyparser= require('body-parser');
- Var url=bodyparser.urlencodedparser({});
- Console.log(req.body);

BODY-PARSER

- The requested data is posted with the help of req.body
- Middleware- gives access to the body property on the request object.
- Example.

MODEL VIEW CONTROL

- Architectural Design pattern
- Most frequently used
- Separates application functionality
- Promotes organized programming

- Ruby on rails
- Express
- Flask
- Django
- angular

- There are many ways to do it.

MODEL

- Responsible for getting and manipulating the data
- Brain of the application
- Interacts with the DB
- Communicates with the controller

VIEW

- The user interacts with the application
- UI
- Html+css+dynamic values send from the controller
- Controller Communicates with the view as well as model
- Template Engine may vary

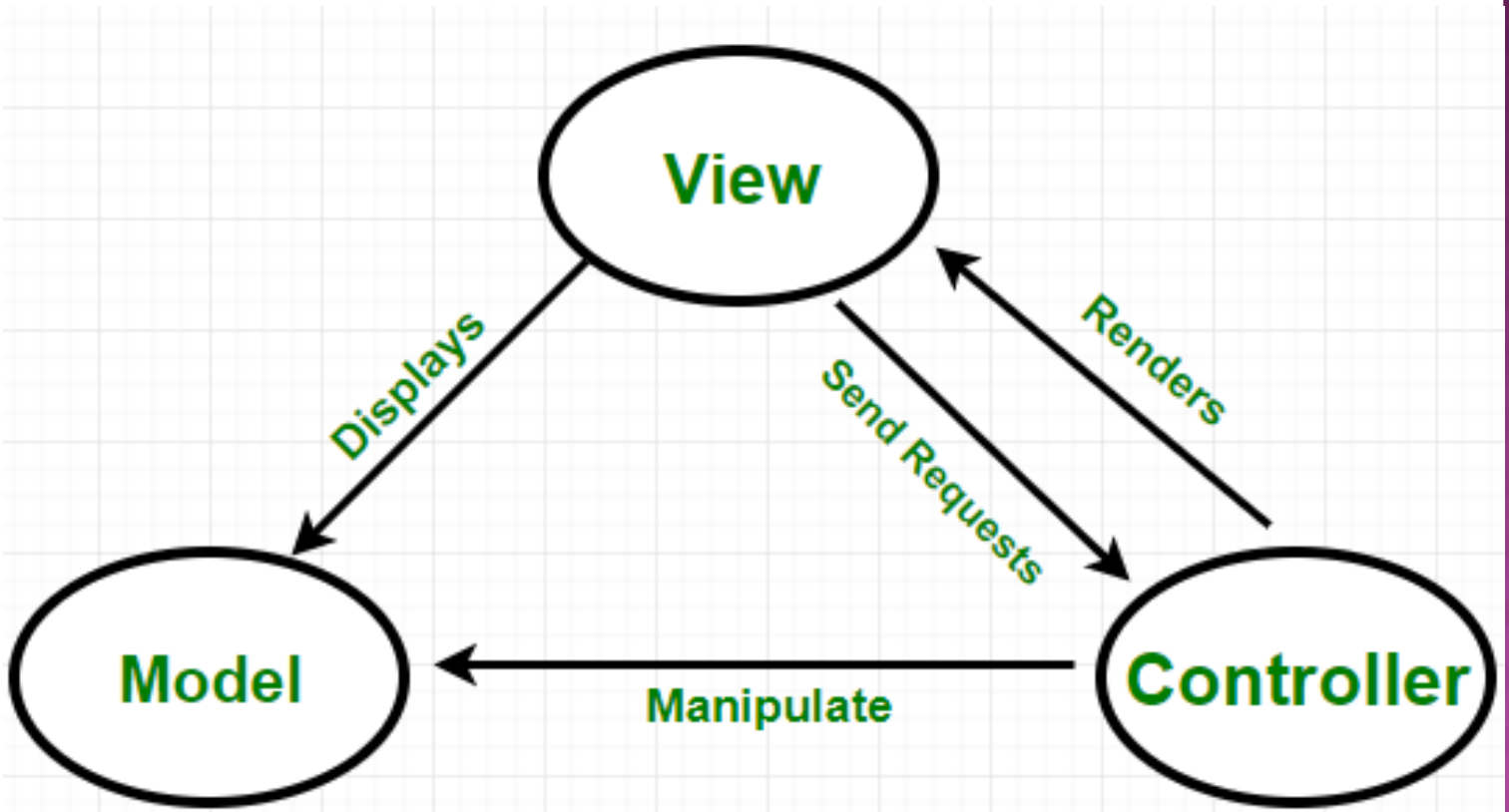
CONTROLLER

- Acts as a middle man
- Takes in user input
- Process request
- Gets data from model
- Passes data to view

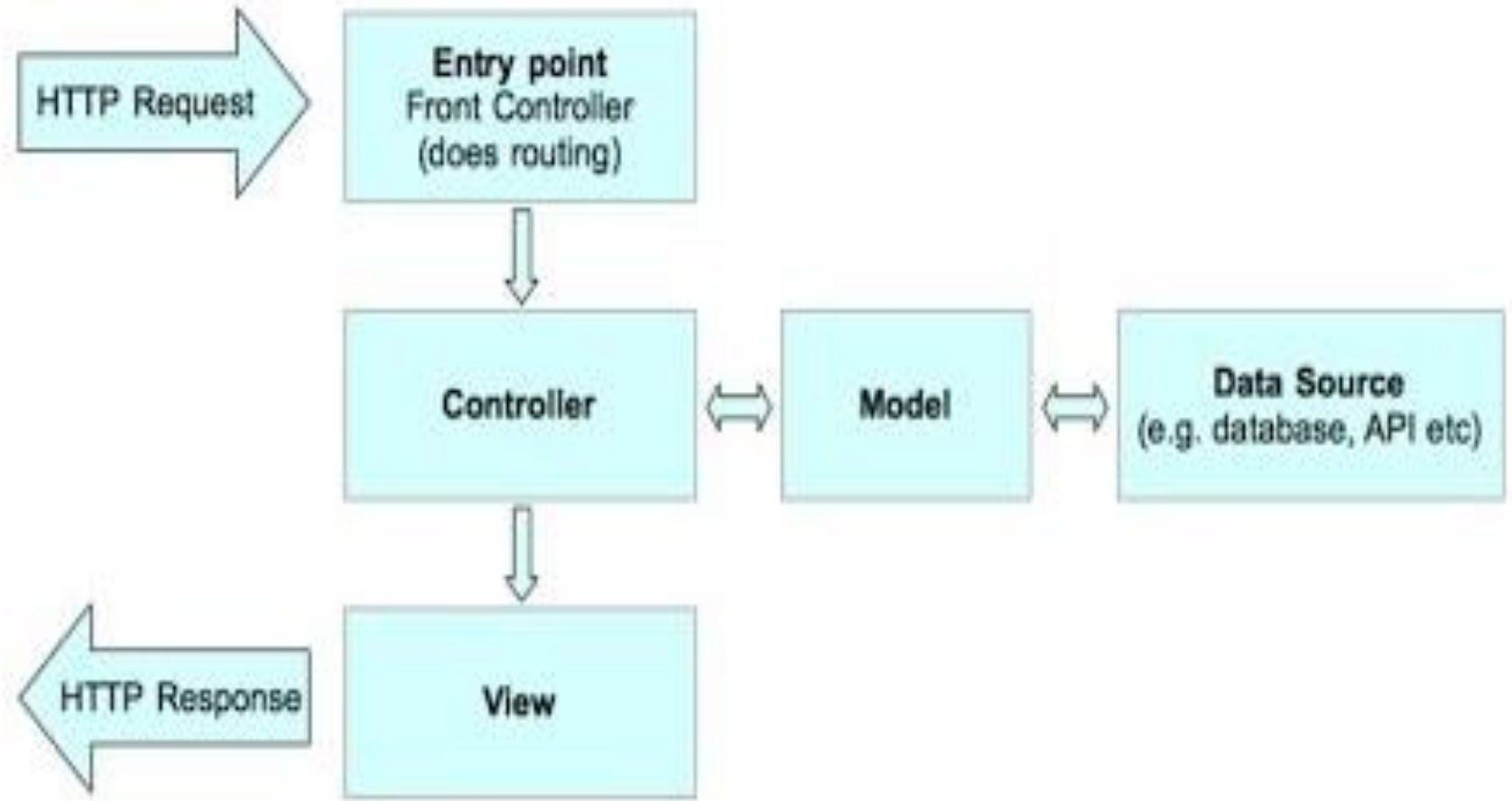
CONTROLLER

- The controller asks the model for data from DB.
- The controller takes that data and loads the view.
- Finally pass those data through view.
- The controller can also load a view without passing data , so just plain web pages can also be passed.

MVC


View
Displays
Send Requests
Renders
Model
Manipulate
Controller

MVC- NODE JS


HTTP Request
Entry point
Front Controller
(does routing)
Controller
Model
Data Source
(e.g. database, API etc)
HTTP Response
View

EXAMPLE

- /routes

user/profile/:id=users . getProfile(id)

//when the route is set to this particular path the control and the particular function is called.

EXAMPLE

- /controllers

```
class user{ function getProfile(id)
{
Profile=this.UserModel.getProfile(id);
Renderview('user/profile',profile)}
```

//calling getprofile function in view

//data will be returned to profile and this profile will be send to view.

EXAMPLE

- /models

```
class usermodel
{
Function getProfile(id)
{
Data=this.db.get('select* from users where id='id');
Return data;
```

EXAMPLE

- /view

  /users/profile

  <h1><%= proile.name %>

  Email : <%= profile.email %>

  < li>Email : <%= profile.phone %>

  //based on view engine.

MAKING AN APPLICATION

- To-do Appliaction
- Steps:
  - Create a new folder – public – assets- (style.css, images, basic operation files)
  - Create our package.json file
    - Install all dependencies
    - Npm init
  - Install require packages
    - Express
    - EJS
    - Body-parser

SAMPLE FILE

- App.js

//this is going to be an express application so we require .

```
Var express= require('express');
```

//accesing the express functionalities in a variable

```
Var app=express();
```

//setting a template Engine(ejs)

App.set('view engine', 'ejs');

SAMPLE FILE

```
//static file loader
App.use('/assets',express.static('./public'));
//listening to port
App.listen(3000);

//next step will be create folders
  according to mvc pattern.
```

THE MVC ARCHITECTURE AND DESIGN PRINCIPLES

1. *Divide and conquer*: Three components can be somewhat independently designed.

2. *Increase cohesion*: Components have **stronger** layer cohesion than if the view and controller were together in a single UI layer.

3. *Reduce coupling*: **Minimal** communication channels among the three components.

4. *Increase reuse*: The **view** and **controller** normally make ***extensive*** use of reusable components for various kinds of UI controls.

5. *Design for flexibility*: It is usually quite easy to change the UI by changing the **view**, the **controller**, or both.

6. *Design for **testability***: Can *test* application separately from the UI.