VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

WINTER SEMESTER 2020-2021

DESIGN PATTERNS

SWE2019

J – COMPONENT

Title: BLOGSITE

FACULTY: PROF. JYOTISMITA CHAKI

Slot: A2

Team Number: A18

Team Members:

18MIS0332 – Nithish D

18MIS0369 – Srikar Kotra

18MIS0377 – Patil Navaneeth

ABSTRACT

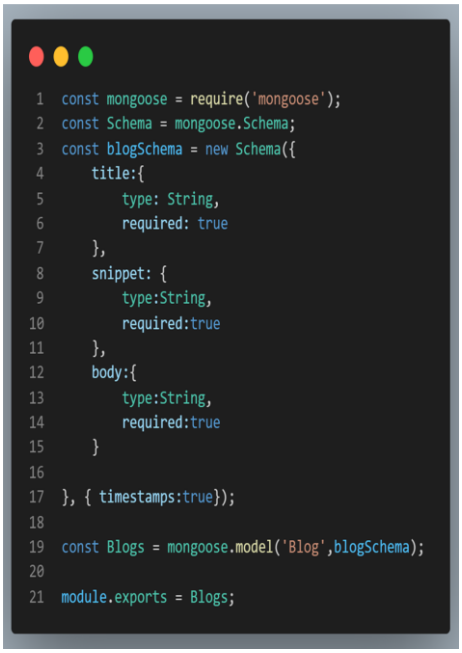
The project is basically a web application. The application deals with the blogging and it allows user to create a simple blog and submit. Once the user submitted the blog it will be added to the database and reflected the in the frontend. Anyone can access the blogs which are created by any user. At any point of time user can delete the posted blog and hence the application we are creating is CRD app i.e., it allows user to Create, Read and Delete. The frontend is going to be supported with HTML, CSS and JAVASCRIPT and the backend technology is NODE JS and for the database Mongodb will be used. Since the frontend view is triggered by the action taken by the user, the architectural pattern we decided to follow is MVC. The problem arises in maintaining the code, the application may look simply but a lot of things are carried from making a request to application to making the appropriate response for the request.

CONTENTS

INTRODUCTION

The application just provides the simple interface for the user to read, post and delete. Behind the scene the application uses dependencies to make the implementation interface simple also for programmers. Nodejs is a runtime environment for javascript in server side. Nodejs uses Express js framework as dependency which makes the request and response simple. For the HTML , and CSS code to run on server, the application uses EJS as framework and to create a schema and model , mongoose will be used and for simplifying the process of user request to the application the middleware called Morgon will be used which is a HTTP request logger for the Node js. And finally to implement utility functions a library called Lodash will be used. The main problem arises for managing these dependencies and to make the code simple and understandable without making it complex i.e. we need only required feature from the dependency.

```
Blogs
{
    "id":1
    "title":"Javascript Design Patterns",
    "snippet":"Intro about various design patterns of JS"
    "body":"There are totally 23 design patterns....."
}
```



```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const blogSchema = new Schema({
    title:{
        type: String,
        required: true
    },
    snippet: {
        type:String,
        required:true
    },
    body:{
        type:String,
        required:true
    }

}, { timestamps:true});

const Blogs = mongoose.model('Blog',blogSchema);

module.exports = Blogs;
```

How did the project evolve?

In review 1 we came under a conclusion on what architecture pattern can be used and we have decided to conclude with MVC architecture.

In review 2 we implemented view component, developed interface and set the server

In review 3 we added the database, backend work and finished controller component. We used facade design pattern to simplify the code and made our work easy using EJS to support web view.

Why is this helpful?

Design Patterns establishes solutions to common problems which helps to keep code maintainable, extensible and loosely coupled. Especially in our project, we have learnt more clearly about maintaining the code neatly and even in the further case, developers can understand the code easily because of the use of facade pattern. We have understood more clearly about design patterns through this project and enables us to make solutions to solve any particular type of problem.

BACKGROUND

LITERATURE SURVEY

Title: CRUD Operations in MongoDB

Link:

https://www.researchgate.net/publication/265160520_CRUD_Operations_in_MongoDB

Literature review:

MongoDB is a superior and truly scalable document-situated data set created in C++ that stores information in a BSON design, a dynamic schema document structured like JSON. Mongo hits a sweet spot between the incredible question capacity of a social information base and the dispersed idea of different data sets like Riak or HBase[1]. MongoDB was created remembering the utilization of the data set in an appropriated architecture, all the more specifically a Shared Nothing Architecture, and that is way it can evenly scale and it upholds Master-Slave replication and Sharding.

MongoDB utilizes BSON to store its documents. BSON keep documents in an arranged rundown of components, each component has three components: a field name, an information type and a worth. BSON was intended to be efficient away space and scan speed which is accomplished for huge components in a BSON document by a prefixed with a length field. All documents should be serialized to BSON prior to being shipped off MongoDB; they're later de-serialized from BSON by the driver into the language's local document portrayal.

Title: Node js challenges in implementation

Link:

https://www.researchgate.net/publication/318310544_Nodejs_Challenges_in_Implementation

Literature review:

Node.js offered ascend to the Full Stack Developers who are presently ready to oversee worker and client side by their own. Node.js is quick and solid for hefty documents and weighty organization load applications because of its occasion driven, non-blocking, and asynchronous approaches, where engineers can likewise keep a complete project in single pages (SPA) and can use for IOT. The aftereffect of study concludes from an overview and from writing survey the execution zones and challenges of the Node.js[2]. Finally, will give idea on the best way to improve to overcome the challenges.

Title: A review and analysis of technologies for developing web applications

Link:

https://www.researchgate.net/publication/230668912_A_review_and_analysis_of_technologies_for_developing_web_applications

Literature review:

While creating business applications, the IT director should make frequently settle on the significant and irreversible decision of which stage the carry out the arrangement (Taudes, 2000). The most common choices were work area or the web stage. Web application advancement has been seeing increasing offer in the by and large IS projects. Be that as it may, web applications do no deliver themselves well to application of the customary frameworks' improvement cycle model (Gellersen and Gaedke, 1999)[3]. The conventional approach additionally sets aside heaps of effort to complete particular web application. There are different kinds of web applications.

Title: Comparison between client-side and server-side rendering in the web development

Link:

https://www.researchgate.net/publication/341880604_Comparison_between_client-side_and_server-side_rendering_in_the_web_development

Literature review:

Compulsory workers for widespread applications that is accessible to number of clients might be an obstruction for the corporation and excessive for little applications despite the fact that it could bring the compatibility benefits. Realizing that request of web application increases to give convenience and usability to the clients, client side delivering comes to create programming all the quicker and more efficient. It has been finished by redirecting the solicitation towards a HTML document then the worker will give messages with no content or a stacking screen until the device takes all JavaScript to permit the program compiling everything prior to showing the content.[4]

Title: MongoDB – a comparison with NoSQL databases

Link:

https://www.researchgate.net/publication/327120267_MongoDB_-_a_comparison_with_NoSQL_databases

Literature review:

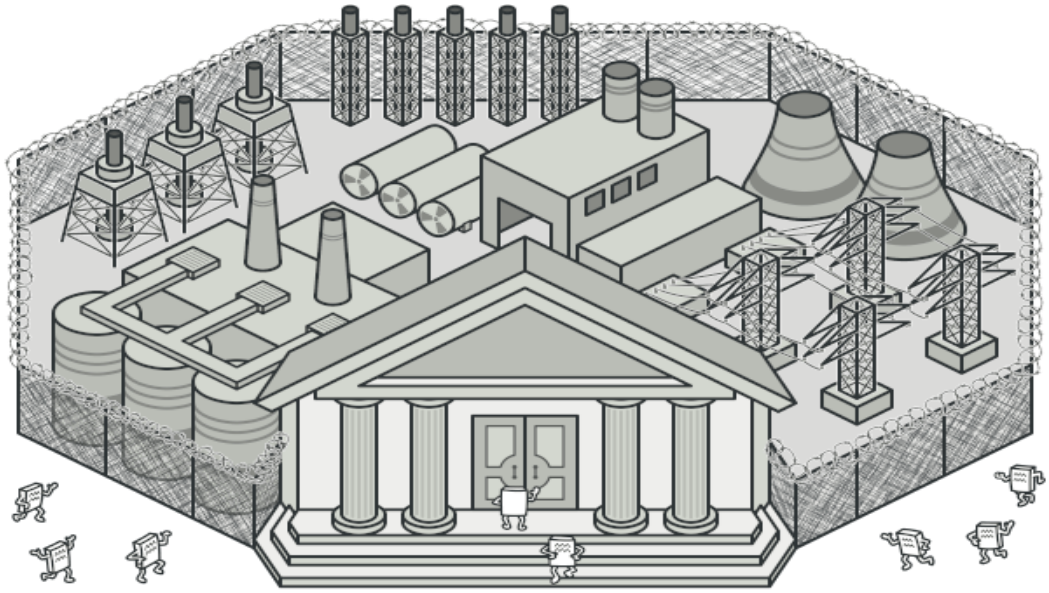
Online applications and their information the board needs have changed dynamically in the previous few years. Assortment of highlights and strict information consistency is given by the social data sets. Because of huge cost of putting away and controlling information in classical social data set frameworks, NoSQL data sets have been created. NoSQL information bases give greater scalability and heterogeneity when compared to RDBMS. MongoDB, a NoSQL information base gives high scalability, performance and accessibility. MongoDB is a document-based NoSQL information base intended for Internet and electronic applications. Information model of MongoDB is not difficult to expand on because of its innate help for unstructured information.[5]

MOTIVATION

At the end the application will be running in the server with the interactive GUI for the client. The request will be processed by the controller and the response will be fetched from the server and at the same time the model will change the view depend on the request. The methodology is CRUD except the update feature will not be present. The acronym CRUD refers to all major operation that are implemented in Relational database application. Each letter in the acronym can map to a standard structured query language, Hypertext protocol method or Data distribution service. The experience of node package manager (NPM), installing required package, building web application needed.

CRUD	SQL	HTTP	DDS
Create	INSERT	PUT	write
Read	DELETE	GET	read
Update	UPDATE	PUT	write
Delete	DELETE	DELETE	dispose

PROPOSED APPROACH



Architectural pattern: MVC

Design pattern: FAÇADE

As we mentioned earlier MVC architectural pattern will be used for the structure of code. We are using various dependency and requiring a particular function from it, the business logic of our application is tightly coupled to the object of the 3rd party package. In order to maintain the code working with the subsystem Facade design pattern will be used. The dependency list will be maintained in the separate file called package.json. And when needed a particular feature the request will be made from the business logic.

For eg. If I need a random number generator from the lodash library, the request for it will be

"const generator = require('lodash');"

"const num = generator.random(1,20);"

Though the library contains various moving parts , we make use of certain feature that needed by using the facade making the code easy to maintain and comprehend.

DESIGN PATTERN

FAÇADE:

Intent -Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Applicability

Use the Facade pattern when

• you want to provide a simple interface to a complex subsystem. Subsystems often get more complex as they evolve. Most patterns, when applied, result in more and smaller classes. This makes the subsystem more reusable and easier to customize, but it also becomes harder to use for clients that don't need to customize it. A facade can provide a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

• there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.

• you want to layer your subsystems. Use a facade to define an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Participants

• Facade (Compiler)
o knows which subsystem classes are responsible for a request.
o delegates client requests to appropriate subsystem objects.
• subsystem classes (Scanner, Parser, ProgramNode, etc.)
o implement subsystem functionality.
o handle work assigned by the Facade object.
o have no knowledge of the facade; that is, they keep no references to it.

Collaborations

• Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem object(s). Although the subsystem objects perform the actual work, the facade may have to do work of its own to translate its interface to subsystem interfaces.
• Clients that use the facade don't have to access its subsystem objects directly.

Consequences

The Facade pattern offers the following benefits:
1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.

2. It promotes weak coupling between the subsystem and its clients. Often the components in a subsystem are strongly coupled. Weak coupling lets you vary the components of the subsystem without affecting its clients. Facades help layer a system and the dependencies between objects. They can eliminate complex or circular dependencies. This can be an important consequence when the client and the subsystem are implemented independently.
Reducing compilation dependencies is vital in large software systems. You want to save time by minimizing recompilation when subsystem classes change. Reducing compilation dependencies with facades can limit the recompilation needed for a small change in an important subsystem. A facade can also simplify porting systems to other platforms, because it's less likely that building one subsystem requires building all others.

3. It doesn't prevent applications from using subsystem classes if they need to. Thus, you can choose between ease of use and generality.

Related Patterns

• Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.
• Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them.

Architectural pattern: MVC

Model

Here we model the database according to our convenience and data field required. We needed three main data field in order to store the blog in the database. They are Blog title, Snippet and body. With these we model the database and inform the server and database that these three should be included.

```
const { truncate } = require('lodash');
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const blogSchema = new Schema({
    title:{
        type: String,
        required: true
    },
    snippet: {
        type:String,
        required:true
    },
    body:{
        type:String,
        required:true
    }
}, { timestamps:true});
const Blog = mongoose.model('Blog',blogSchema);
module.exports = Blog;
```

Controllers

We came to a final and most important step. We have created an interface and modelled and set up the database but how it will be controlled, how the blogs will be fetched from the database and made available in the interface, when creating a new blog how it will be saved, when deleting how the blog will be deleted and how the database will know which blog to delete. All this question can be answered by Controller of MVC architecture. Controller takes the whole control and updates the view.

```
const Blog = require('../models/blog');

const blog_index = (req,res) => {
    Blog.find().sort({ createdAt: -1})
    .then((result)=>{
          res.render('blogs/index',{title:'All blogs',blogs:result});
    }).catch((err)=>{
        console.log(err);
    })
}
const blog_details = (req,res) => {
    const id = req.params.id;
    Blog.findById(id)
    .then((result)=>{
        res.render('blogs/details',{title:'Blog Details',blog:result})
    }).catch((err)=>{
        res.status(404).render('404', {title: 'Blog not found'});
    })
}
const blog_create_get = (req,res) => {
    res.render('blogs/create',{title:'Create a BLog'});
}

const blog_create_post = (req,res) => {
    console.log(req.body);
    const blog = new Blog(req.body);
```

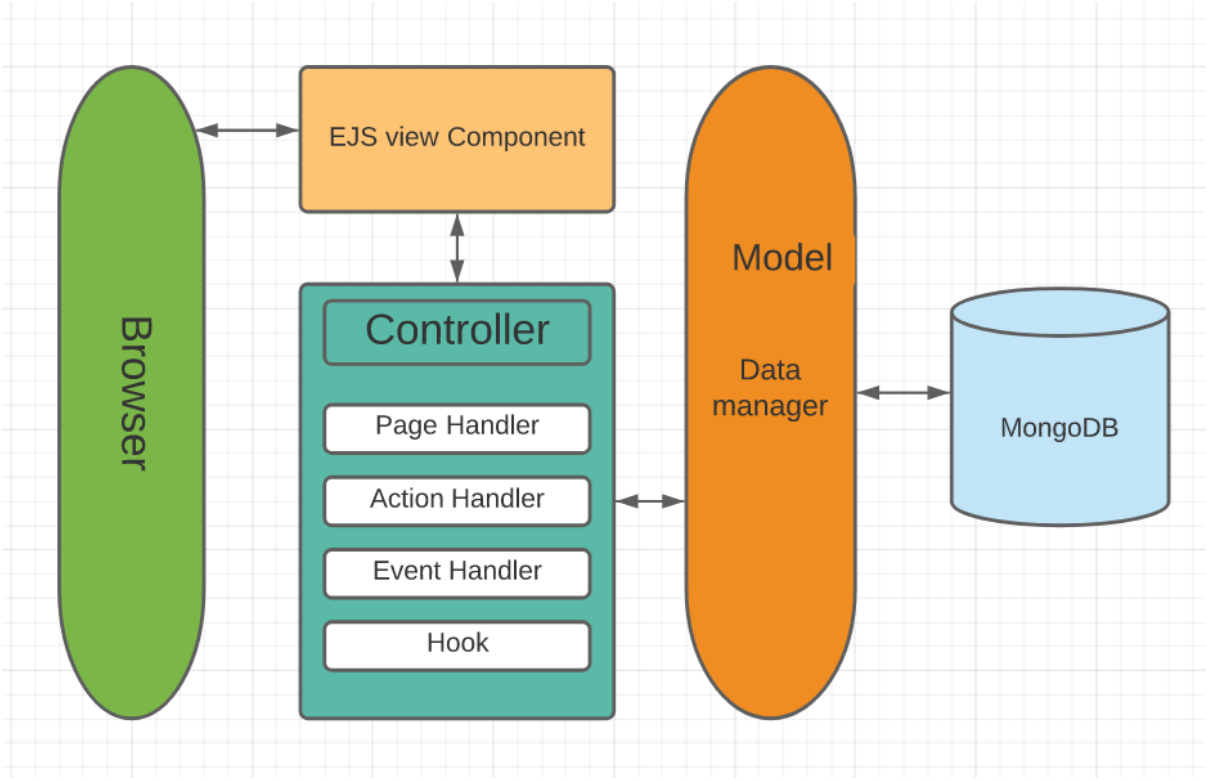
```
    blog.save()
    .then((result)=>{

        res.redirect('/blogs');
    }).catch((err)=>{
        console.log(err);
    })
}
const blog_delete = (req,res) => {
    const id = req.params.id;
    Blog.findByIdAndDelete(id)
      .then(result => {
          res.json({ redirect: '/blogs'});
      })
       .catch(err => {
           console.log(err);
       });
}
module.exports = {
    blog_index,
    blog_details,
    blog_create_get,
    blog_create_post,
    blog_delete
}
```

Blog Routes:

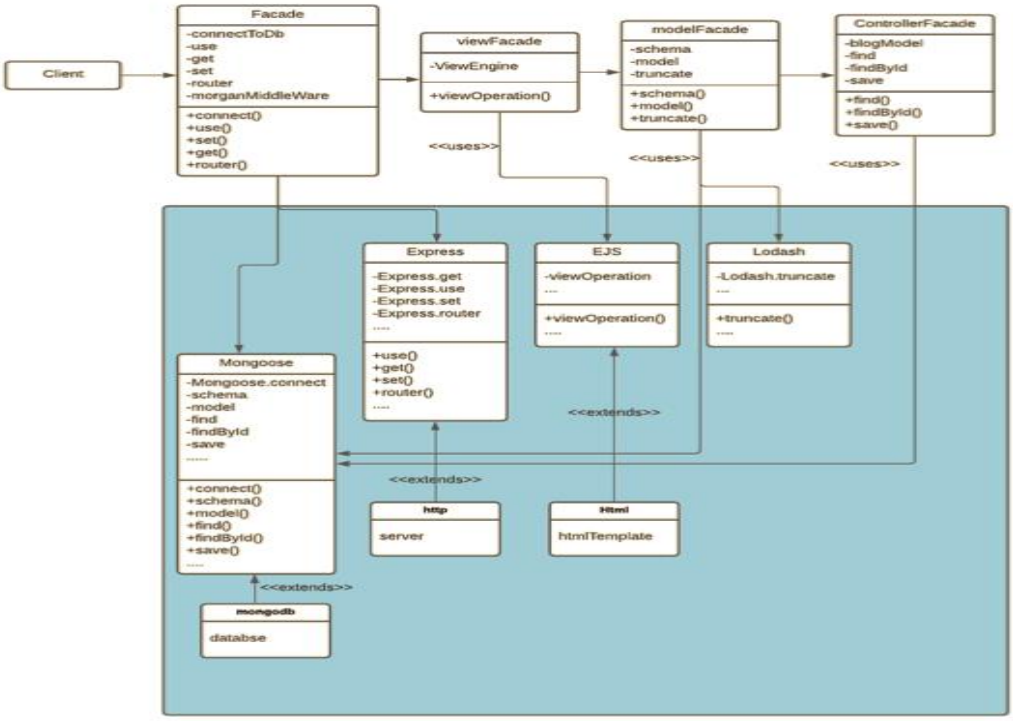
```
const express = require('express');

const blogController = require('../controllers/blogControllers')

const router = express.Router();

router.get('/', blogController.blog_index);

router.post('/', blogController.blog_create_post);

router.get('/create',blogController.blog_create_get)

router.get('/:id',blogController.blog_details);

router.delete('/:id', blogController.blog_delete);

module.exports = router;
```

IMPLEMENTATION

Abstract View of implementation:

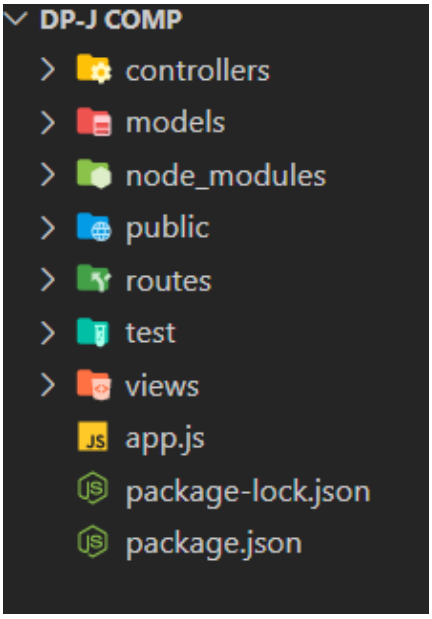


https://lucid.app/publicSegments/view/a93ff8ac-d09b-4bd3-be28-74aef817745b/image.png



We first started with keeping the architecture in mind, we created separate folders called Models, Views, Controllers and started with the View first. We designed the interface and later we prepared to make it dynamic by adding database.

Folder Structure



App.js (root folder)

```
const express = require('express');
const morgan  = require('morgan');
const mongoose = require('mongoose');
const blogRoutes = require('./routes/blogRoutes');
//express app
const app = express();
//In review 2
// const http = require("http")
// const PORT = 3000
// const server = http.createServer()

// server.listen(PORT, error => {
//   if (error) {
//     return console.error(error)
//   }
```

```
//  app.get('/about-me',(req,res)=>{
//     res.redirect('about','301');
// })
// })
// connect to mongo db
const dbURI =
'mongodb+srv://Nithish:welcome123@nodeblog.1kxq2.mongodb.net/note-
tuts?retryWrites=true&w=majority';
mongoose.connect(dbURI, { useNewUrlParser:true, useUnifiedTopology: true})
    .then((result)=>{ app.listen(3000); })
    .catch((err)=> console.log(err));

//register view engine
app.set('view engine','ejs');

app.use(express.static('public'));

app.use(morgan('dev'));

app.use(express.urlencoded({extended:true}));
app.get('/',(req,res)=>{
  res.redirect('/blogs');
});
app.get('/about',(req,res)=>{
  res.render('about',{title:'About'});
});

app.use('/blogs',blogRoutes);

app.get('/about-me',(req,res)=>{
   res.redirect('about','301');
})

app.use((req,res)=>{
   res.status(404).render('404',{title:'404'});
})
module.exports = app;
```

View:

Nav bar

```
<nav>

    <div class="site-title">
        <a href='/'><h1>BLOG</h1></a>
        <p>A node blog post</p>

    </div>
    <ul>
        <li><a href='/'>Blogs</a></li>
        <li><a href='/about'>About</a></li>
        <li><a href='/blogs/create'>New Blogs</a></li>
    </ul>

</nav>
```

2. Footer

```
<footer>
    copywrite &copy; BlogSite
</footer>
```

3.Header

```
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Blog | <%=title%></title>
    <link rel="stylesheet" href="/style.css">
</head>
```

4. 404 page

```
<html lang="en">
    <%-include('./partials/header.ejs') %>
    <body>
        <%-include('./partials/nav.ejs') %>
```

```
        <div class="not-found content">
            oops, page not found!
        </div>
        <%-include('./partials/footer.ejs') %>
    </body>
</html>
```

5. About page

```
<html lang="en">
    <%-include('./partials/header.ejs') %>
    <body>
        <%-include('./partials/nav.ejs') %>
        <div class="about content">
            <h2>BLOG Page</h2>
            <p>Post your blogs</p>
                    </div>
        <%-include('./partials/footer.ejs') %>
    </body>
</html>
```

6.Index

```
<html lang="en">
    <%-include('../partials/header.ejs') %>
    <body>

        <%-include('../partials/nav.ejs') %>
        <div class="blogs content">
            <h2>All blogs</h2>

            <% if(blogs.length>0) { %>
            <%blogs.forEach(blog => { %>
                <a class="single" href="/blogs/<%=blog._id %>">
                <h2 class="title"><%= blog.title %></h2>
                <p class="snippet"><%= blog.snippet %></p>
                </a>
            <% }) %>
            <% } else { %>
                <p>No blogs</p>
            <% } %>
```

```
        </div>
        <%-include('../partials/footer.ejs') %>
    </body>
</html>
```

7. Create

```
<html lang="en">
    <%- include('../partials/header.ejs') %>
    <body>
        <%- include('../partials/nav.ejs') %>
        <div class="create-blog content">
            <form action='/blogs' method="POST">
                <label for="title">Blog title:</label>
                <input type="text" id="title" name="title" required >
                <label for="snippet">Blog snippet</label>
                <input type="text" id="snippet" required name="snippet">
                <label for="body">Blog body:</label>
                <textarea id="body" required name="body"></textarea>
                <button>Submit</button>
            </form>
        </div>
        <%-include('../partials/footer.ejs') %>
    </body>
</html>
```

8. Details

```
<html lang="en">
    <%-include('../partials/header.ejs') %>
    <body>
       <%-include('../partials/nav.ejs') %>
        <div class="details content">
            <h2><%= blog.title %></h2>
            <div class="content">
                <p><%= blog.body %></p>
            </div>
            <a class="delete" data-doc="<%= blog._id %>">
                <img src="/trashcan.svg" alt="detete icon">
            </a>
```

```
    </div>
    <%-include('../partials/footer.ejs') %>
    <script>
        const trashcan = document.querySelector('a.delete');
        trashcan.addEventListener('click',(e)=>{
            const endpoint = `/blogs/${trashcan.dataset.doc}`;
            fetch(endpoint,{
                method:'DELETE'
            }).then((response)=>response.json())
            .then((data)=> window.location.href = data.redirect)
              .catch(err=>console.log(err));
        })</script></body></html>
```

Styling Views with CSS

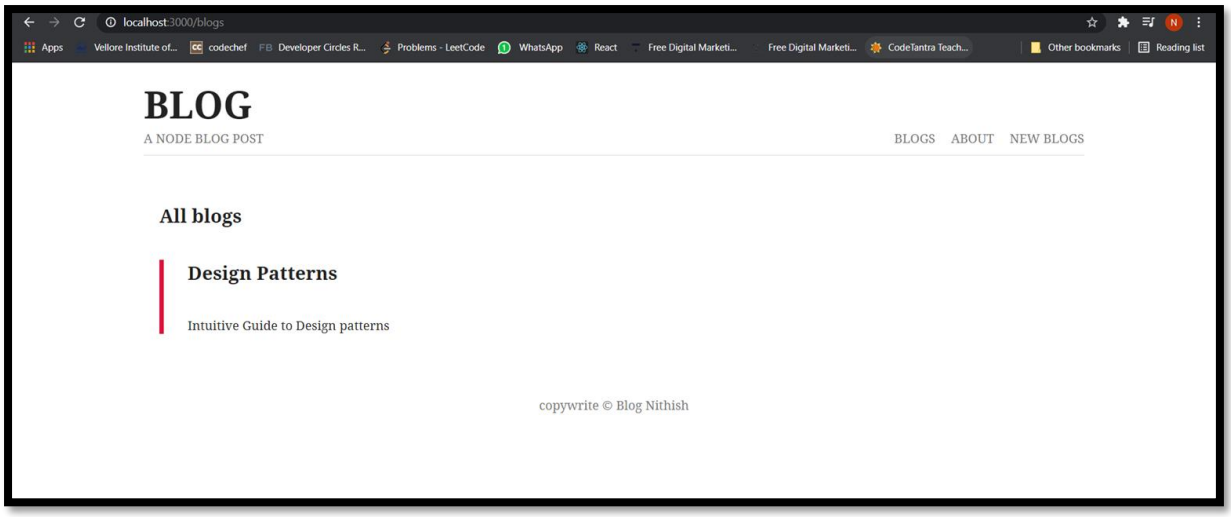
```
@import
url('https://fonts.googleapis.com/css2?family=Noto+Serif:wght@400;700&display=
swap');
body{
  max-width: 1200px;
  margin: 20px auto;
  padding: 0 20px;
  font-family: 'Noto Serif', serif;
  max-width: 1200px;
}
p, h1, h2, h3, a, ul{
  margin: 0;
  padding: 0;
  text-decoration: none;
  color: #222;
}

/* nav & footer styles */
nav{
  display: flex;
  justify-content: space-between;
  margin-bottom: 60px;
  padding-bottom: 10px;
```

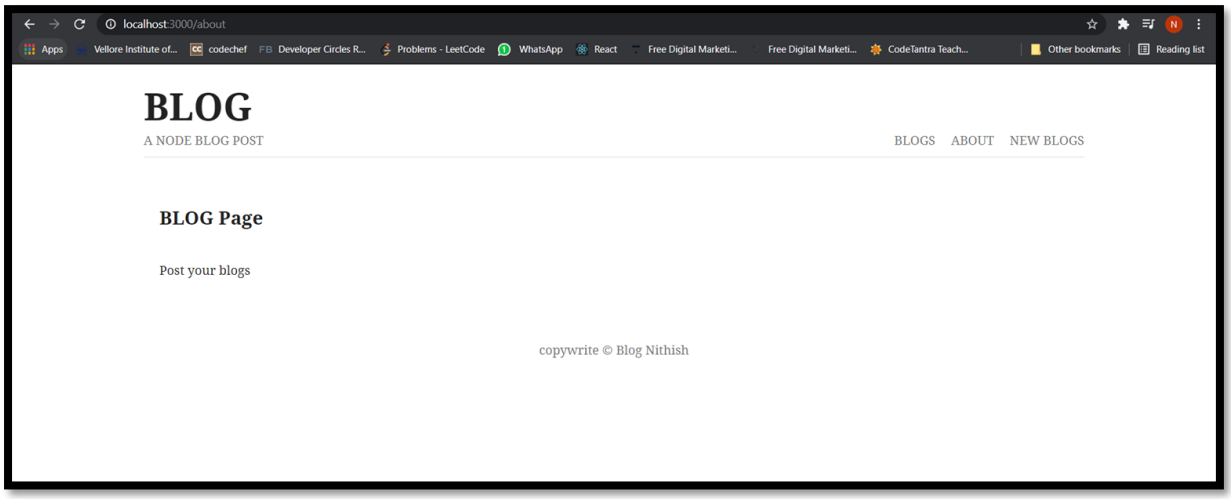
```
  border-bottom: 1px solid #ddd;
  text-transform: uppercase;
}
nav ul{
  display: flex;
  justify-content: space-between;
  align-items: flex-end;
}
nav li{
  list-style-type: none;
  margin-left: 20px;
}
nav h1{
  font-size: 3em;
}

nav p, nav a{
  color: #777;
  font-weight: 300;
}
footer{
  color: #777;
  text-align: center;
  margin: 80px auto 20px;
}
h2{
  margin-bottom: 40px;
}
h3{
  text-transform: capitalize;
  margin-bottom: 8px;
}
.content{
  margin-left: 20px;
}

/* index styles */
.blogs a{
```

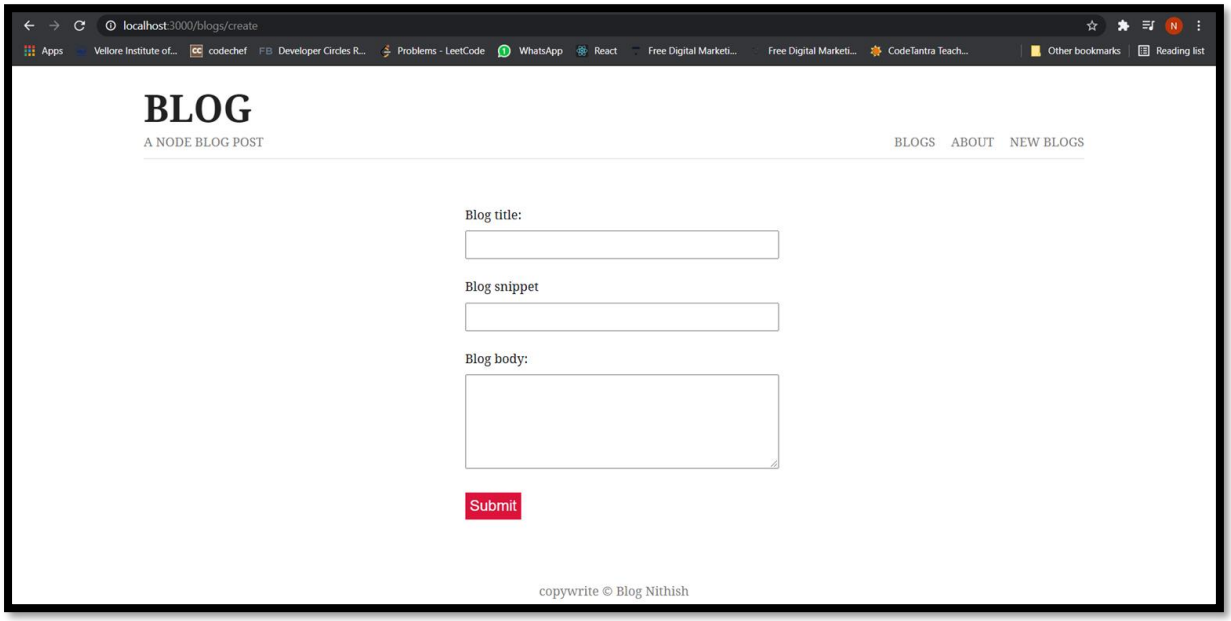
```
  display:block;
  margin: 30px 0px;
  border-left: 6px solid crimson;
  padding-left: 30px;
}
.blogs a:hover h2{
  color: crimson;
}
/* details styles */
.details{
  position: relative;
}
.delete{
  position: absolute;
  top: 0;
  right: 0;
  padding:8px;
  border-radius: 50%;
}
.delete:hover{
  cursor: pointer;
  box-shadow: 1px 2px 3px rgba(0,0,0,0.2);
}
/* create styles */
.create-blog form{
  max-width: 400px;
  margin: 0 auto;
}
.create-blog input,
.create-blog textarea{
  display: block;
  width: 100%;
  margin: 10px 0;
  padding: 8px;
}
.create-blog label{
  display: block;
  margin-top: 24px;
```

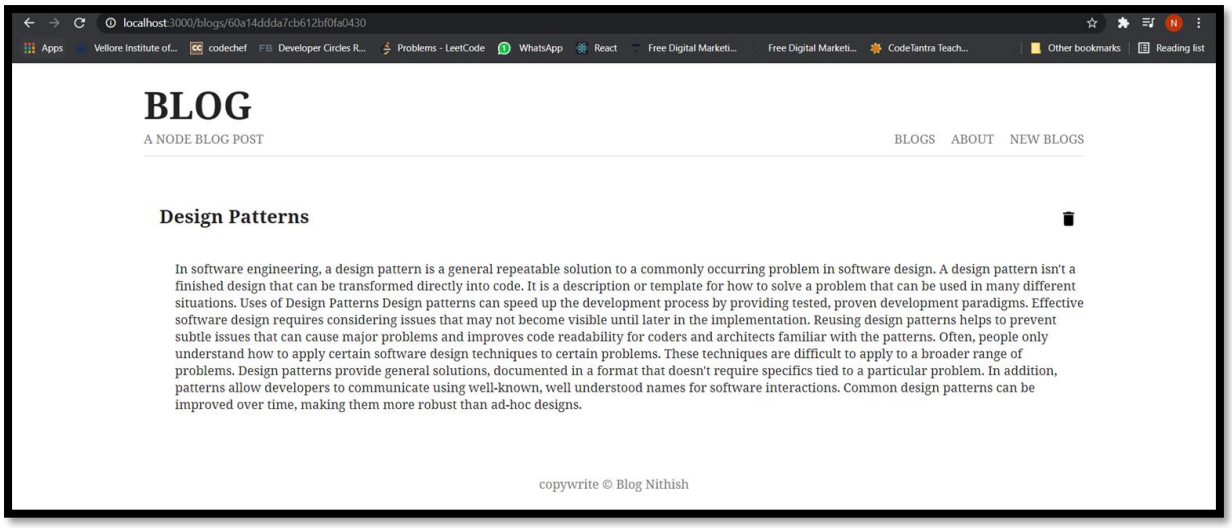
```
}
textarea{
  height: 120px;
}
.create-blog button{
  margin-top: 20px;
  background: crimson;
  color: white;
  padding: 6px;
  border: 0;
  font-size: 1.2em;
  cursor: pointer;
}
```

OBTAINED RESULTS

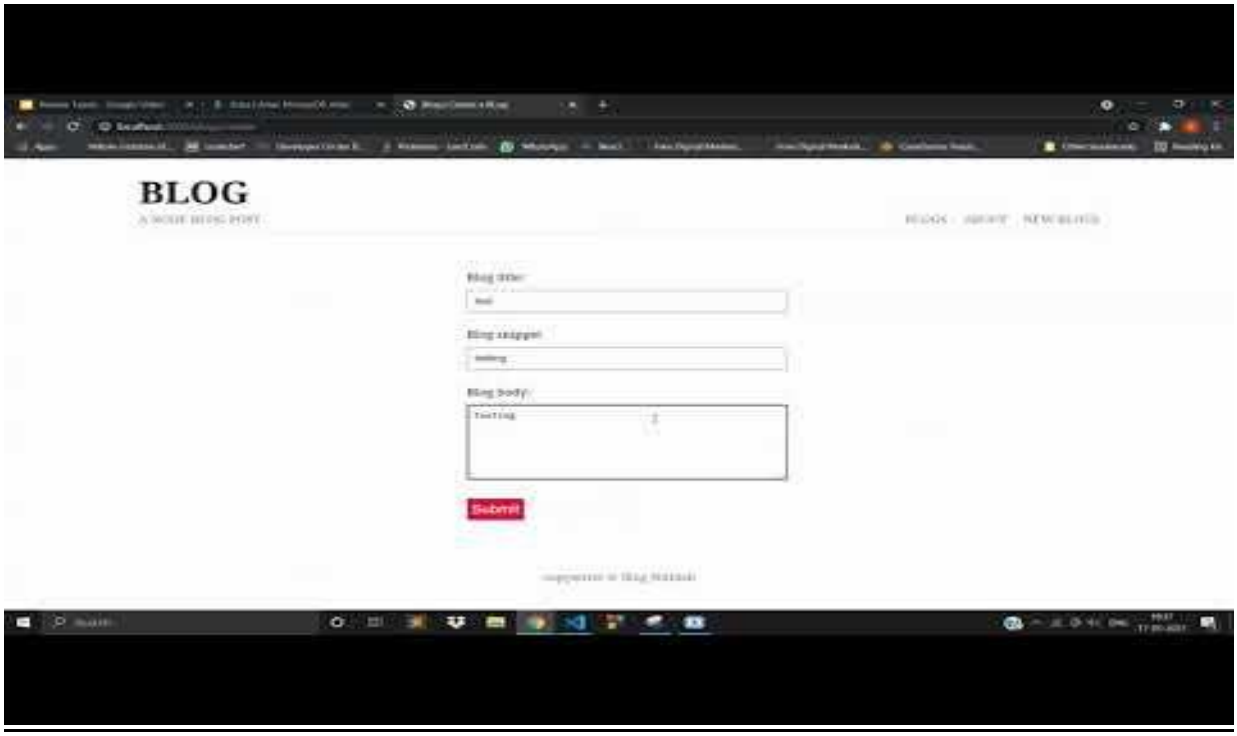








Final Application (Double click on the image to play video):



The results obtained is not only an webpage but also server running in the background which gives the results. Each page is the result of ejs template which we created in the view folder.

Remembering the definition:

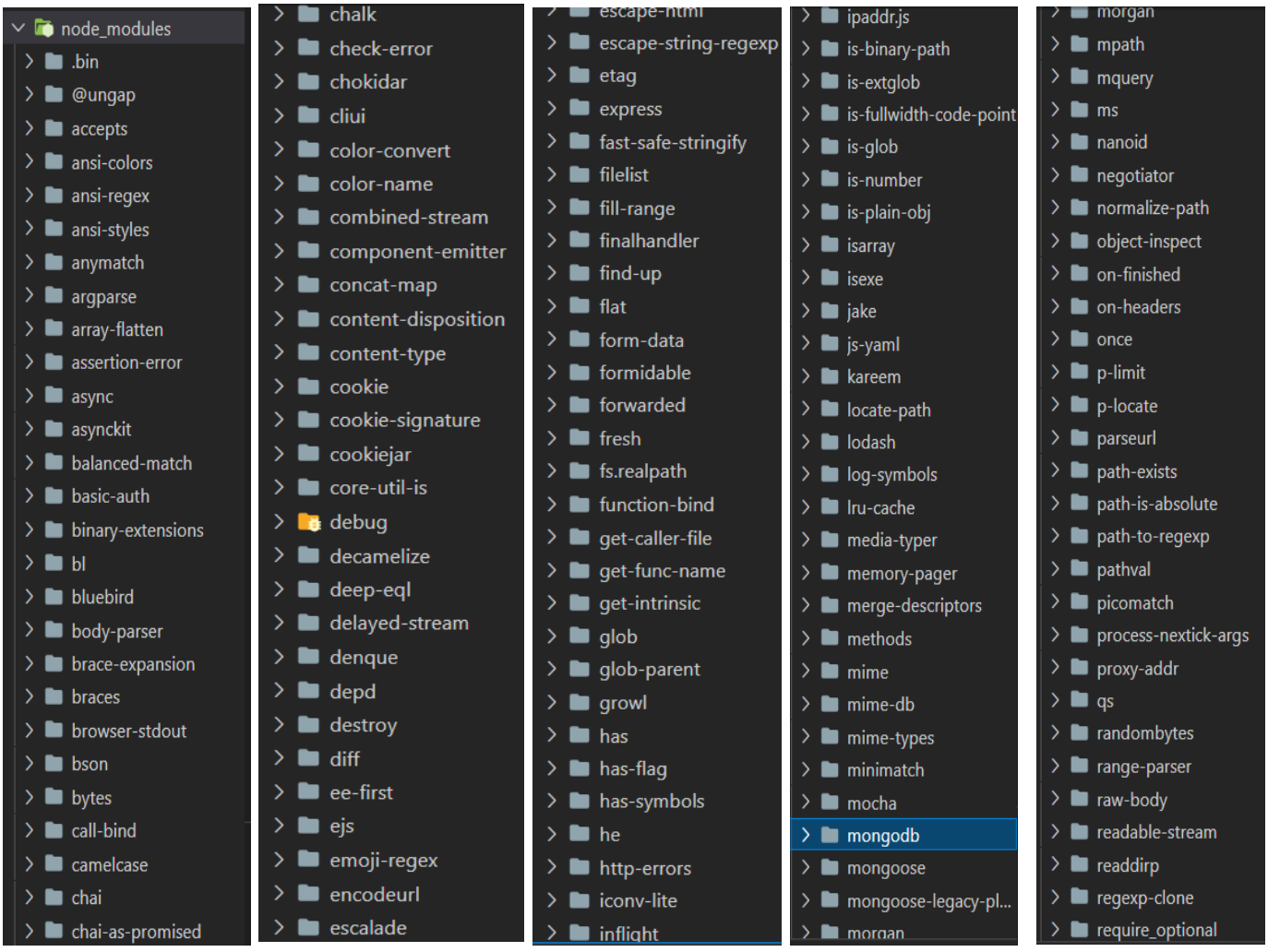
Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.

We Created an MVC architecture and started using the facade design pattern. The huge libraries are maintained in the folder called node_modules. We are requiring the needed interface through the require method. There are many classes and methods in the node modules but we don't need it all. With facade pattern we are making it so simple to implement the required interface even without the Classes and objects.
The server is running through an huge library called express. There are lots of methods in the express but we are requiring few like get(), use().
We required the express library through the code
const express = require('express');

Note: Class and function are almost equivalent in javascript. The Property applies to class also applies to functions.

ANALYSIS

We have the complete application developed with the simple interface and with no complex code. The facade design pattern provided the way of structuring the huge libraries in separate folder and acted as a guide to require what we needed. There are complex classes in the library if we include them all in the main code, it will be a real maintenance problem as the code progress in time. For a simple application itself it will be thousands of codes making it more difficult to maintain. The library contains huge sub folders which is shown below.



Complex classes in the libraries:

```
class OrderedBulkOperation extends BulkOperationBase {
  constructor(topology, collection, options) {
    options = options || {};
    options = Object.assign(options, { addToOperationsList });

    super(topology, collection, options, true);
  }
}

/**
 * Returns an unordered batch object
 * @ignore
 */
function initializeOrderedBulkOp(topology, collection, options) {
  return new OrderedBulkOperation(topology, collection, options);
}

initializeOrderedBulkOp.OrderedBulkOperation = OrderedBulkOperation;
module.exports = initializeOrderedBulkOp;
module.exports.Bulk = OrderedBulkOperation;

class BulkWriteResult {

  constructor(bulkResult) {
    this.result = bulkResult;
  }

  /**
   * Evaluates to true if the bulk operation correctly executes
   * @type {boolean}
   */
  get ok() {
    return this.result.ok;
  }
```

```
/**
 * The number of inserted documents
 * @type {number}
 */
get nInserted() {
  return this.result.nInserted;
}
```

As we can see so much huge code can't be presenting in the main code which will even the simpler things like connecting to database, getting the content, redirecting the route difficult. So with the facade pattern we separated it from the main code and used what is needed.

With the facade pattern We minimize the complexity of the application, Aids principle of loose coupling, Software becomes more flexible and more expandable.

With the MVC architecture we are providing separation of concerns and in future if something need to be updated it won't affect other rest of the code and won't affect the application from running. It provides an isolation of the application's presentation layer that displays the data in the user interface, from the way the data is actually processed.
In other words, it isolates the application's data from how the data is actually processed by the application's business logic layer.
The biggest advantage of the MVC design pattern is that you have a nice isolation of these components/layers and you can change any one of them without the rest being affected.

Here is the list of the major advantages of this pattern.
- It provides a clean separation of concerns.
- It is easier to test code that implements this pattern.
- It promotes better code organization, extensibility, scalability and code reuse.
- It facilitates de-coupling the application layer.

Comparison with other design patterns:

ABSTRACT FACTORY:

Abstract Factory can be used with Facade to provide an interface for creating subsystem objects in a subsystem-independent way. Abstract Factory can also be used as an alternative to Facade to hide platform-specific classes.
In our project, we cannot use Abstract Factory as an alternative to Facade because when you do not want to hide the way the subsystem objects are created from the client code.

MEDIATOR:

Facade and Mediator have similar jobs: they try to organize collaboration between lots of tightly coupled classes.
Facade defines a simplified interface to a subsystem of objects, but it doesn't introduce any new functionality. The subsystem itself is unaware of the facade. Objects within the subsystem can communicate directly.
Mediator centralizes communication between components of the system. The components only know about the mediator object and don't communicate directly.
• Mediator is similar to Facade in that it abstracts functionality of existing classes. However, Mediator's purpose is to abstract arbitrary communication between colleague objects, often centralizing functionality that doesn't belong in any one of them.
In this project, even though the objects are tightly coupled with each other, we also want the objects to communicate with each other without any introduction of mediator.
Henceforth, mediator is not used.

ADAPTER:

Facade defines a new interface for existing objects, whereas Adapter tries to make the existing interface usable. Adapter usually wraps just one object, while Facade works with an entire subsystem of objects. For blog website we want multiple objects that enables us to create new blogs with an entire subsystem of objects. Therefore, facade is preferable to adapter for our project.

SINGLETON:

A Facade class can often be transformed into a Singleton since a single facade object is sufficient in most cases. This project is practically impossible to be executed with a single object and hence singleton can't be used.

CONCLUSION

With the MVC architecture and Facade design pattern, the application was easy to develop without having to interact with the complex classes. The MVC architecture provided a way to separate the presentation layer from the logic and processing layer and helped to achieve separation of concern and work as a team to develop individual layer. Overall, the code is easy to maintain and expandable in the future with the use of appropriate design pattern.

REFERENCES:

[1] Truică, Ciprian-Octavian & Boicea, Alexandru & Trifan, Ionuţ. (2016). CRUD Operations in MongoDB. 10.2991/icacsei.2013.88.

[2]Shah, Hezbullah & Soomro, Tariq. (2017). Node.js Challenges in Implementation. Global Journal of Computer Science and Technology. 17. 72-83.

[3]Mandava, Asha & Antony, Solomon. (2017). A review and analysis of technologies for developing web applications.

[4]Iskandar, Taufan & Lubis, Muharman & Kusumasari, Tien & Lubis, Arif. (2020). Comparison between client-side and server-side rendering in the web development. IOP Conference Series Materials Science and Engineering. 801. 012136. 10.1088/1757-899X/801/1/012136.

[5]Krishnan, Hema & Elayidom, M.Sudheep & Santhanakrishnan, T.. (2016). MongoDB – a comparison with NoSQL databases. International Journal of Scientific and Engineering Research. 7. 1035-1037.