SIT 725 Prac 5
MVC Structure

Contents

- MVC Structure
- Routes
- Controllers
- Services
- Importance of index.js
- Callback Hell
- Questions

MVC Structure

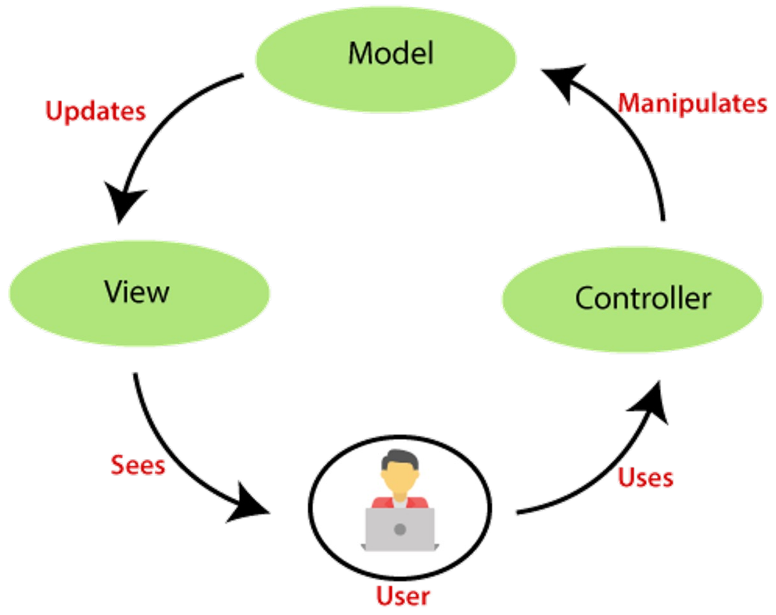
Any software development project's success hinges on good architecture. This enables not just smooth development processes among teams, but also the application's scalability. It ensures that developers will have little trouble refactoring various portions of the code whenever new modifications are required.

In diverse languages, architecture patterns such as MVT in Python, MVVM in Android, and MVC in JavaScript apps exist.

MVC architecture divides the whole application into three parts; the Model, the View and the Controller.

MVC Structure Cont ...

- Model: Our data is defined in this section. It's where we save our schemas and models, or the blueprint for our app's data.
- View: This includes templates as well as any other interaction the user has with the app. It is here that our Model's data is given to the user.
- Controller: This section is where the business logic is handled. This includes database reading and writing, as well as any other data alterations. This is the link between the Model and the View.



Routes

Now lets see how we modify our older structure into the new MVC Structure. You can see in the code that we added three new folders Controllers, Routes and Services.

Routes are the files that are responsible to serve the endpoints of our application to the user. We try to create separate route files for separate functionalities in our application. For eg in the code we have a file projects.js which is responsible to serve all the endpoints related to projects in our application.

This file is appended to our server.js by the command

```
let projectsRoute = require('./routes/projects')
app.use('/api/projects',projectsRoute)
```

Routes Cont ...

You can see that we added a part (‘/api/projects’) with the project routes what this does is that this appends ‘/api/projects’ infront of all the routes inside our projects.js. So the routes in our projects.js looks like something this

- GET - /api/projects
- POST - /api/projects

Suppose we had another GET route in our projects.js which started like /myprojects then the full path of that route would become

- GET - /api/projects/myprojects

Controllers

Controllers are the parts that are responsible to handle the business logic of our application. When we call an endpoint from the user we send a bunch of data with that end point what needs to be done with that data is handle by our controllers. So when ever we create a route file we also create a corresponding controller file which handles all the business logics for all the different route endpoints in that file.

You can see in the example code that we also created a projectsController.js for our routes file projects.js.

Whatever data comes from our routes directly goes into our controllers. Keep in mind we never do any business logic inside our routes file the only part we do in out routes is to take inputs and send it to the controllers and whatever result we get from the controller we then pass it to the user from our routes.

Controllers Cont ....

Controllers are also important because they are a bridge between our routes and models. We make the modifications in the data received from the user in our controllers and then basically send it to the models where they can be either saved or updated or retrieved from our database.

Services

The folder services is mainly an extension of the MVC model that we create in order to make our life easy. Think of the situation where we are trying to save data or get data from our database we would have to write the same code again and again in our controllers. So we create a service file in our project that is responsible to handle the basic CRUD (create, retrieve, update and delete) operations on our models.

Service file would never have any business logic in them there sole purpose is to just perform basic CRUD operations.

In our code base you can see we created projectService.js which has all CRUD operations for our project model and because we created this file we can reuse these functions wherever we need in out projectController.js

Services Cont ...

Another good reason of creating a service file is that suppose in future we see that the npm we are using has changed and we have to do modifications in our project we would only have to change this service file and it would bring out the change everywhere these functions are being used.

Now think if we wouldn't have created this file we would have to go through the whole project and make these changes everywhere we are using these CRUD operations.

Importance of index.js

We also made a very big change in how we import the files into other files when we start using MVC. You can see that each folder we created also has a new file called index.js in it.

In javascript when we are importing and we don't specify the exact path of a file and instead we specify the folder javascript knows that by default it needs to read the index.js file.

Index.js file responsible to handle all the indexing of the files in that folder. How this makes our life easy is that we can specify all the different files in the folder in our index.js file and index them by providing them a name. This in future helps us to use the functions of that file with the help of a dot operator.

Have a look at the index.js we created in our Controller folder. It looks something like this

Importance of index.js Cont ...

```
module.exports={
    projectsController:require('./projectsController')
}
```

And now when we call this folder in our routes files projects.js we can directly use the keyword projectController.<Function name> to call a particular function inside our projectsController.js file.

```
router.get('/', (req, res) => {
    Controllers.projectsController.getProjects(res);
})
```

Callback Hell

Callback Hell, also known as Pyramid of Doom, is an anti-pattern seen in code of asynchronous programming. It is a slang term used to describe and unwieldy number of nested “if” statements or functions.

A few callbacks appear harmless if you don't expect your application logic to become too sophisticated. However, as your project's requirements grow, you'll quickly find yourself with layers upon layers of nested callbacks. Next you know you have successfully created a Callback Hell.

An example of callback hell would something like this.

Callback Hell Cont ...

```
fs.readdir(source, function (err, files) {
  if (err) {
   console.log('Error finding files: ' + err)
  } else {
   files.forEach(function (filename, fileIndex) {
    console.log(filename)
    gm(source + filename).size(function (err, values) {
     if (err) {
      console.log('Error identifying file size: ' + err)
     } else {
      console.log(filename + ' : ' + values)
      aspect = (values.width / values.height)
      widths.forEach(function (width, widthIndex) {
       height = Math.round(width / aspect)
       console.log('resizing ' + filename + 'to ' + height + 'x' + height)
       this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
        if (err) console.log('Error writing file: ' + err)
       })
      }.bind(this))
     }
    })
   })
  }
 })
```

Callback Hell Cont ...

You don't need to understand what's happening in that function but instead understand that in that function we start one function then call another function inside it and then another and keep going. This is called a callback hell.

So how do we fix it. To answer your question we have already fixed it. The MVC structure is a fix to solve the problems of callback hell. Modularising your application solve the issue of callback hell as we dont create a function inside a function but instead we create event that never cause a callback hell.

Thanks