Avery Peiffer

COE 1530 HW 2

Our movie recommendation algorithm fits well into the pattern of layered architecture. Layered architecture involves adjacent layers in a sequential manner, with information flowing in both directions between them. The layers are usually drawn from the following: the user interface layer, which is concerned with the presentation of the application; the application layer, where application functions determined by use cases are executed; the domain layer, where business use cases are carried out; and the service/support layer, where general but essential services are handled. The layers are built with increasing layers of abstraction, where the lower layers are seen as a set of services. This approach is useful because individual layers can be swapped out relatively painlessly; that is, their separation allows for a layer to be removed without causing significant consequences to the rest of the application. Additionally, the layers are decoupled, in that they can operate without knowledge of any other layer. This means that the layers can be independently designed, further adding to the flexibility of the product. The product’s lower layers can also be designed generically, since they operate at a lower level of abstraction. A downside to the layered architecture pattern is that the infrastructure is localized on one machine, which can limit scalability and increase the possibility for issues.

This pattern fits our project because of the natural hierarchy of ReactJS projects. The highest level of abstraction is represented by the user interface and decreases as data moves through the layers. The rest of the front-end code is concerned with executing functions in accordance with the application’s use cases, sending requests to the back-end when necessary. The back-end of the application consists of relatively simple Python functions, mostly concerned with keeping track of data in a Sqlite database. These back-end functions send data back to the front-end, keeping with this pattern’s theme of bidirectional data flow. Additionally, the main layers can be developed independently of one another, because their interactions will come through the use of an API that serves as a bridge between them. This allows for each layer of our project to be developed separately and simultaneously, increasing productivity. The project’s infrastructure will have to be localized; however, this issue is somewhat mitigated by the relatively small database of movies that we will have to maintain. Though scalability could theoretically become an issue as the number of users increase, it should not factor into this smaller project.

The Model-View-Controller (MVC) pattern is another architectural pattern that can be used to characterize our project. The MVC pattern is similar to the multilayered pattern in that it aims to decouple various layers of an application; however, the MVC pattern is more concerned with decoupling the user interface layer from the rest of the application. The MVC pattern separates the model, which is the application’s functional layer; the view, which is the user interface; and the controller, which directs activities in response to the user. The controller creates the view and updates both the view and the model in response to actor events. Additionally, the model can notify the view about changes. These components reside in different architectural layers, meaning they can be somewhat independently designed, and the view and controller can easily reuse components for various user interface controls. The application can also be designed in a very flexible and testable manner using the MVC pattern.

Again, the architecture of ReactJS projects lends itself well to a MVC pattern. The templates and style sheets included in various components constitute the application’s view. The controller is represented by the various observables and observers that are set up to track and respond to user behavior. These observers then call the various functions that make up the model, which is represented by the application’s back-end. The functional parts of the application are executed in the back-end, and data is sent back to the view after it has been updated or changed. Though the view and controller are tightly knit together, in that they reside in the same files, they are still relatively decoupled in the sense that they do not need to be developed simultaneously. However, there is more tangible decoupling that occurs between the view/controller and the model, as these two sets of components do not really need to know about each other as long as the requisite conditions are met to send requests between them. Using this pattern allows for our project to be developed from multiple angles simultaneously. Some of us are currently developing the back-end, which consists of using Python to execute database operations and perform linear algebra calculations. While that is happening, others are developing the responsive user interface that makes up the front-end.