Architecture Pattern

Group 98's application will follow a Model-view-controller (MVC) architectural pattern. This is primarily because Django is an MVC framework, although they define things differently. In Django, the controller is the "view" and the view is the "template". Django calls their framework MTV (Model-template-view), but admits that it is an MVC framework. The interaction between the layers is the same in Django as it is in other MVC frameworks.

Traditional web development often resulted in a glorified CRUD (Create-Read-Update-Destroy) interface to a database, and in cases such as this little is gained from isolating a controller layer. In cases where the application is expected to carry out changes that do not have a direct effect on the database, it is better to separate this logic into another layer. This idea (MVC) is the current prevailing architecture for developing complex web applications.

MVC has many benefits, these are probably best explained through an example. Imagine a form on a webpage that creates quote for blinds for a user. It is necessary for the application to take this form input and do something with it before creating an invoice record in the database. The controller here would be responsible for interpreting the parameters from the form and executing some logic associated with these parameters. The controller might need to determine what type of invoice it is, who the user is, who the recipient is, the due date, etc.

For example, the form might give the option for the user to select what type of blinds they want. The controller takes these parameters and executes the business logic related to them. If a user wants a new type of blinds, the controller might pick up on that and automatically send an email to that user containing useful information about the blinds. The controller might also detect that this is the first invoice that has been created for this user, so it might send a welcome email to them too. While executing this logic the controller sends commands to the model to create the necessary records in the database, the controller doesn't know anything about how the data is persisted in the database.

Without this separation, this complex business logic might not be possible, or would need to be done in the "view" or "model" layer. In any of these cases, the code in those layers would become overly complex and difficult to separate logically.

A primary benefit of this controller layer is that the same behaviour can be repeated elsewhere in the application using the same method defined in the controller. It isn't necessary for complex code to be duplicated across multiple views, multiple views can utilize the same controller methods.

Additionally, controllers simplify interaction between multiple models. Views get their information from the models through the controller. This means that the view layer can focus on presenting something to user, and rely on the controller to collect all the information from various models in the correct way.

Compared to older methods of web development the separation of the controller layer is the main difference, and hence the main benefit to utilizing an MVC pattern. The model and view layers are also different because of this abstraction, in that they are greatly simplified so that it is easy to determine their purpose and figure out what is going on.

MVC separation also makes development by multiple developers easier, which is another of the main reasons why this is the choice for group 98. One developer can work on one view, while

another can work on another view, while another can work on the controller, and it can all be brought together easily, even though each piece might rely on one another.

The decoupling offered by MVC also allows easy scaling and manipulation of one of the layers without affecting another. For example, the database could be changed from mySQL to Mongoid without effecting the controller or view layers.