INDIAN INSTITUTE OF TECHNOLOGY MADRAS • सिद्धिर्भवति कर्मजा •

IIT Madras

ONLINE DEGREE

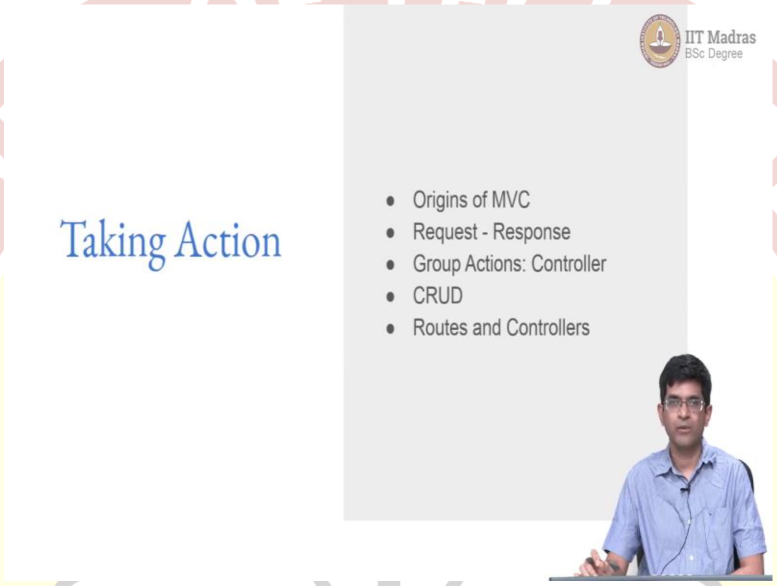
Modern Application Development 1

Professor. Nitin Chandrachoodan
Department of Electrical Engineering
Indian Institute of Technology, Madras

MVC Origins

Hello everyone, and welcome to this course on Modern Application Development. Hello everyone. Today we are going to look at the topic of controllers.

(Refer Slide Time: 0:22)



Taking Action

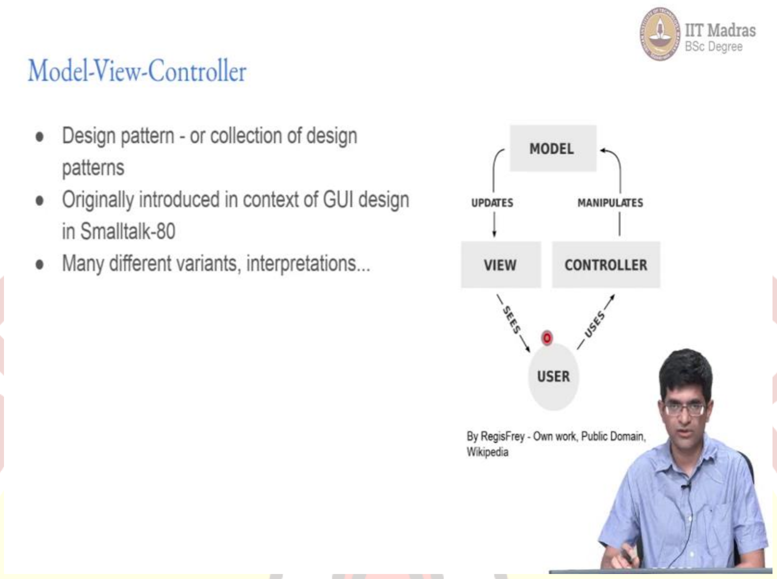
- Origins of MVC
- Request - Response
- Group Actions: Controller
- CRUD
- Routes and Controllers

And this, in some sense, the reason why I have written taking action over here is because controllers are all about actions, about responding to something that the user has requested, making changes if necessary on the model side. But finally, also making changes on the view that the user sees.

So, before we get into this, this is potentially a slightly controversial topic and the reason is, because if you search online for MVC and especially in the context of something like flask, you will see any number of people sort of vehemently objecting to even calling flask an MVC framework and pretty much any definition of MVC that you come across, you are likely to find people finding some fault with it.

So, what I am going to do is take a slight detour into the origins of the MVC concept, just to sort of set context over here and then I will sort of go further and explain how I plan to use the concept of the controller and in what sense, we are really using the MVC concept over here and the main thing that I want you to take away from this set of lectures on the controller part of the application development is that it is more a way of thought, rather than one specific way of doing things. So with that in mind, let us get into the origins of MVC.

(Refer Slide Time: 1:47)



Model-View-Controller

- Design pattern - or collection of design patterns
- Originally introduced in context of GUI design in Smalltalk-80
- Many different variants, interpretations...

Now, MVC stands for Model-View-Controller, it can be thought of as either a design pattern, as I mentioned earlier, a design pattern is essentially something which over years of experience in implementing certain kinds of designs, people have come up with as a nice consistent way of implementing certain kinds of logic.

On the other hand, MVC involves at least three concepts that we can see in the title itself and there are a number of cases, where you would actually see this referred to as a collection of design patterns, rather than a single design pattern in itself and there are various names given to the patterns.

Now, for somebody who wants to go deep into the study of software engineering, those make a lot of sense, because ultimately the reason why we study these things is they help us to better understand where we came from and more importantly, what to do next. So, that is the reason for trying to sort of systematise, the study of these kinds of topics, it is not just that people are, do not have anything better to do than give it names.

Giving it these names helps us to understand what is special about it and therefore, what parts of it are good and should be taken forward versus what parts of it are probably best left behind for the particular context in which you are working. Now, one important thing to understand in this discussion of MVC is that it was originally introduced in the context of

GUI design or GUI design, graphical user interface design, in the Smalltalk-80 programming language.

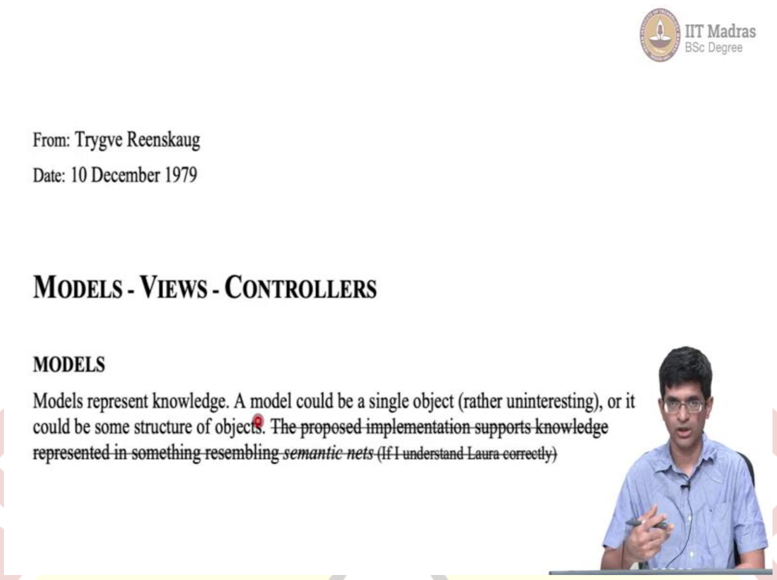
So, Smalltalk-80, as the name suggests, was somewhere around 1980, it was developed and it was a heavily object oriented language, meaning that the whole concept of objects was built into the language right from the start and because of that, there were certain ways of thinking about various parts of programmes that lend themselves naturally to the context of that language.

In particular, so this is what the Wikipedia page for MVC has a picture, you have the model and the model basically updates the view, which is what the user sees. Now the user might want to change the view, might want to look at something else and to do that, they cannot directly go back and change the view, they need to use a controller, which will go back either request data or manipulate in some ways the model and get back to you with a view.

So, this is the loop that essentially happens over here. So, as you can see over here, all of this is completely centred around the user and it is about user interface design and what we are talking about is the fact that the user should not directly manipulate the data. In other words, the user to model there is no connection. I cannot go in and directly make changes in the database, I have to use a controller, which means that the kinds of changes I can make are automatically constrained.

And in turn, the model will then determine what the user gets to see, by means of controlling the view. Now the problem with this approach, although it looks simple on the face of it, is that there are many different variants and many different interpretations of this idea.

(Refer Slide Time: 5:02)



From: Trygve Reenskaug

Date: 10 December 1979

MODELS - VIEWS - CONTROLLERS

MODELS

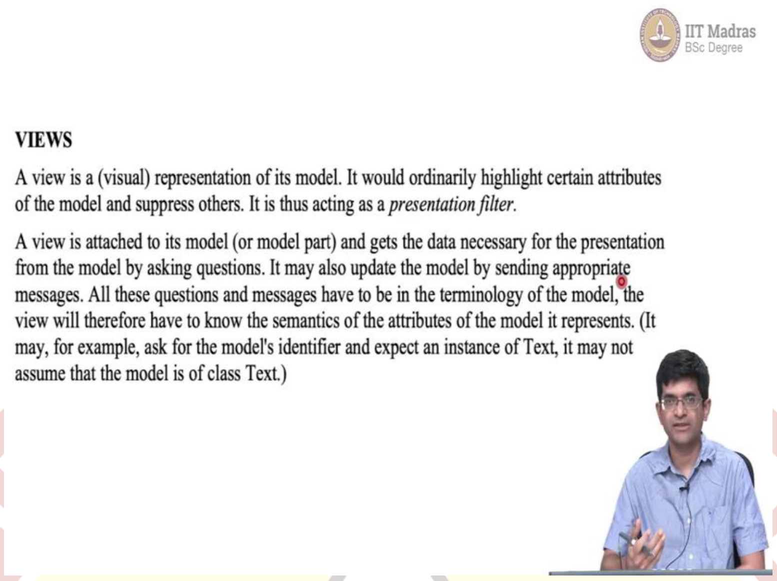
Models represent knowledge. A model could be a single object (rather uninteresting), or it could be some structure of objects. ~~The proposed implementation supports knowledge represented in something resembling *semantic nets* (If I understand Laura correctly)~~

So, what I am going to very briefly do is just put up, a couple of slides over here, which are literally from the person who is credited with having come up with this idea in the first place, Trugve Reenskaug, he was working at the Xerox Palo Alto Research Centre, during the time that Smalltalk-80 was being developed and this was part of the work that he came up with over there.

And this, what I have put up over here are just snippets from an internal note, essentially it is sort of an email that he sent to other people in the group or there, which sort of described his way of systematically structuring different components of GUI kind of system and if you look at it, it is worth sort of reading, it is a very short note, just like one and a half pages. So, from that point of view, what he is saying is, the models represent knowledge, it could be either about a single object or it could be some structure of objects.

Now, that is like sufficiently vague and general. But it also gives you an idea of what is happening, it essentially encapsulates information about an object. In the example that we were looking at earlier, that could be details about a student or it could be structure of objects, you have students courses and the connections between them, all of those are captured as either a model or multiple models that interact with each other.

(Refer Slide Time: 6:25)


VIEWS

A view is a (visual) representation of its model. It would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a *presentation filter.*

A view is attached to its model (or model part) and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. All these questions and messages have to be in the terminology of the model, the view will therefore have to know the semantics of the attributes of the model it represents. (It may, for example, ask for the model's identifier and expect an instance of Text, it may not assume that the model is of class Text.)

The view is a visual representation of the model and the important thing to notice over here is this, it is acting as a presentation filter. So, what presentation filter in this case means is that the database contains data just as, large blobs of binary information and obviously, that is not how you want to look at it, you would rather look at it in a way that the human being can understand. So, that is a presentation filter, it shows you the information that you need in the present context, so that you can make sense of it and make something useful out of it.

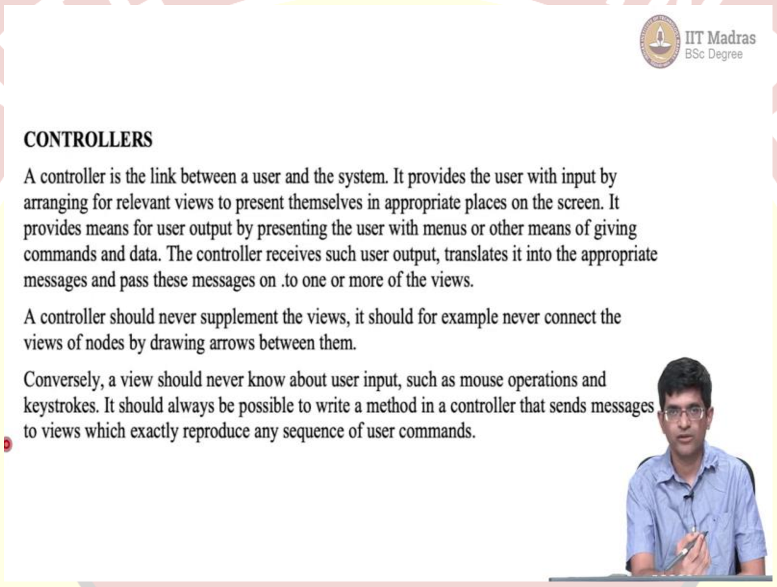
And there are a few more things over here that, it is would be a good idea to sort of, look up this paper and read through it. It is available in fact, on his website. It is not that it is too much detail, as I said, it is a very short note, but it sort of gives you an idea of what was the thought process that went into defining the model view controller idea when it started out. Remember, this was 40 plus years ago.

Now, the important things, as far as we are concerned is that the view is attached to the model, it gets the data necessary for the presentation from the model by asking questions. So, you would remember that in previous lectures, we had looked at essentially models and how we can query using SQL, the Structured Query Language, we can query information, querying basically means asking questions. So, the queries are used in order to gather data about the model, which is then presented to the user in the form of a view.

Now, it may also update the model by sending appropriate messages and this is an important thing to keep in mind. This, in fact is one of the core ideas of object orientation. So, object orientation is not so much about classes and inheritance and polymorphism. Although that is not what we normally think of, the most fundamental core idea of object orientation is the fact that there are models of objects that then you can communicate with by sending appropriate messages.

That encapsulation and the fact that everything, any modification to the object or getting data out of it, happens through the process of these messages is at the heart of object orientation. So you can see that, so far at least the model and view put together are very clearly, they have their roots in object oriented programming. So, this one is reasonably simple and easy to understand.

(Refer Slide Time: 8:59)



CONTROLLERS

A controller is the link between a user and the system. It provides the user with input by arranging for relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. The controller receives such user output, translates it into the appropriate messages and pass these messages on .to one or more of the views.

A controller should never supplement the views, it should for example never connect the views of nodes by drawing arrows between them.

Conversely, a view should never know about user input, such as mouse operations and keystrokes. It should always be possible to write a method in a controller that sends messages to views which exactly reproduce any sequence of user commands.

The tricky part then comes what is the control and this is slightly less clearly defined. As you can see, it is also a longer description. But more importantly, it also starts falling apart later in our context, we will get to that later. But right now, the original definition at least said it is the link between a user and the system.

Now, what is the view, the view is what the user gets out of the system. But the controller in some sense is the other part of the story, it provides the user with input, now look at this. Now, suddenly, we have done things. The way that this is describing it, is that the controller is providing the user with input and normally when we say input, we are always talking about input to a computer.

This is turning things around and saying the computer is generating output, which is input to the user. So, the controller, in some sense provides input to the user by arranging for relevant views to present themselves. What does that mean? It means that the controller finally decides what gets displayed on screen. Not just that in appropriate places on the screen, it could be for example, things like a menu bar or toolbar or some button that needs to be pressed. That is what we mean when we say that we are arranging for relevant views to present themselves in appropriate places on the screen.

Now, the second part is that the controller receives such user output. Once again, it is been turned around, so that the user is now generating output, which becomes the input to the system, what the controller does is it receives that, translates it into appropriate messages. Remember, you can only communicate with the model through messages and then passes them to the model and eventually back to the views.

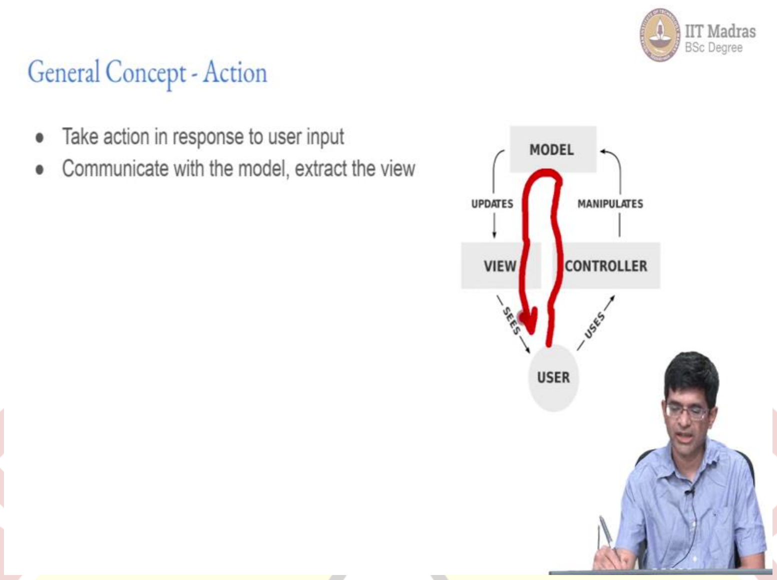
So, the controller, in some sense is very closely tied with the view, because it is deciding what views to present to the user, but is also making sure that the user is never directly interfacing with the model, it takes care of that communication. There are a couple of other interesting things to note over here, one of them is a controller should never supplement the views. What that means is, if a view has been defined in a certain way, all that the controller can decide is what are the values of various things on the view, it cannot.

For example add anything new, it cannot construct a new view on the fly, it should not exist. And we want to keep that separation of concerns. Avoid separation of concerns is something that I have mentioned earlier and I am going to be repeating multiple times today and the other interesting thing over here is, the view should never know about user inputs, such as mouse operations and keystrokes.

What exactly does that mean? It means that in the view, you should not, for example hard code in something that says if this button was pressed or if this key was typed in, all that is handled by the controller. So, there is something, whenever you have binding a mouse click or a key press to an action, you are binding it to an action, not a view. The action essentially is what we are thinking of as the controller.

So, please keep that in mind. At the end of the day, what we are trying very rigorously as far as possible to do over here, is to separate out the model, which only stores data, the view which shows data to the user and the controller, which actually takes care of action and how do we sort of go about implementing that kind of action, is what this entire set of lectures is going to be about.

(Refer Slide Time: 12:37)



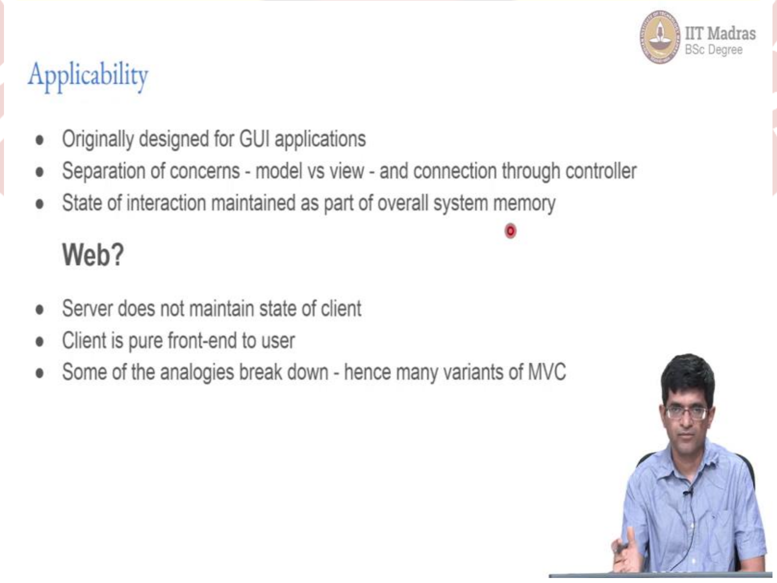
General Concept - Action

- Take action in response to user input
- Communicate with the model, extract the view

So, the general concept that we are talking about is there has to be an action, which is taken, which basically takes user input does something and most likely modifies the model or may not modify, it may only read data from the model without modifying it and finally gives it back to the view, it communicates with the model, extracts the view and then finally brings it back to the user.

So, that kind of action is what we mean by controller. However, when we get sort of deeper into this, we will realise that there are sort of slightly more nuanced meanings given to that term controller that we should probably think about a little bit more.

(Refer Slide Time: 13:17)



Applicability

- Originally designed for GUI applications
- Separation of concerns - model vs view - and connection through controller
- State of interaction maintained as part of overall system memory

Web?

- Server does not maintain state of client
- Client is pure front-end to user
- Some of the analogies break down - hence many variants of MVC

So, let us think about the applicability, the MVC model was originally designed for GUI applications and the main idea as I said, was this separation of concerns, we have a model, we have a view and they are connected through a controller and what happens in a typical GUI application, let us say running on your Windows laptop or on a Linux desktop or laptop, is that there is some concept of the state or the interaction, after all whatever is being displayed on the screen is being done by a programme which knows exactly what is there and what you are likely to click on and there are certain limits on what you can do at that point in time.

So, in other words, there is some concept of a state that the overall system is in and the MVC model was built around that that concept, the fact that there is a state, which is determined by the underlying system and that the controller, there are limitations on what you can do and therefore the controller can directly be the link between the view and the model, is what got implemented in the MVC structure.

Now, when we come to the web, things change a little bit. One major change is that the server and the client are now in most cases of interest, at least on different machines. Which means that in general, the server cannot know the state of the client. Because in between the server having sent one response to the client and the next request coming from the client, the client may have rebooted the client may have moved to a different place and picked up a different IP address.

The client may have gone into a low power mode, lots of other things could have happened, which the server knows nothing about. That is not the case in a typical desktop GUI application. But it is very often the case on the web or at least has to be the default state that we assume that the server cannot know the state of the client.

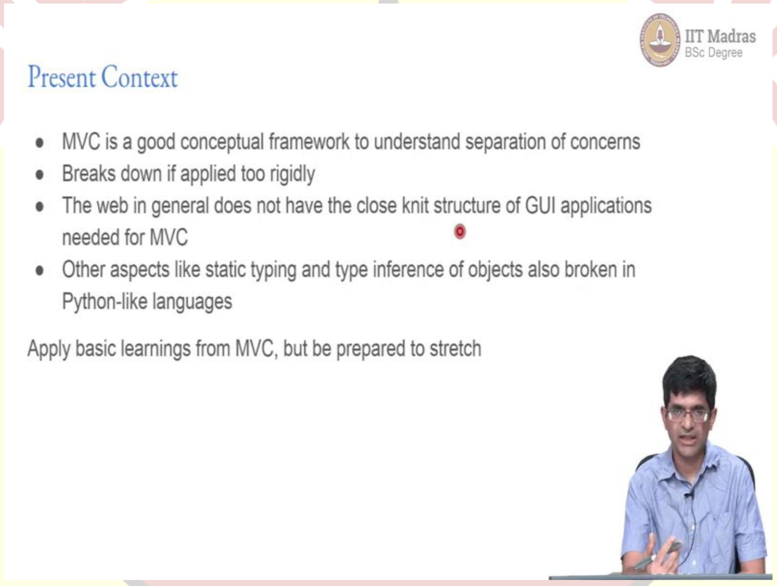
The client in turn is purely a front end to the user, it is pretty much just a browser, you are looking at it and saying, this is what I am seeing in front of me, this is how I need to interact with it. Because of this, many of the analogies that are used in order to understand MVC breakdown and hence, we come up with any number of different variants that are used to understand how MVC is applied.

And you will, in fact come across, a whole lot of comments on StackOverflow and various other places, which say that this is not MVC or that is not MVC or how could you possibly describe something like this as MVC? And in fact, I would not be surprised if some of you right now are going to look through this entire series of lectures and say, but what you have described is not MVC, quite possible.

What I am trying to convey over here is that there is a frame of mind or rather thought process involved in MVC, the separation of concerns that I am repeatedly stressing on, that is useful from the point of view of designing an application. Whether you choose to call that MVC at the end, or some other model may determine how you go about designing your system.

But at the end of the day, what you need to keep in mind is that you should try and have as clear as possible architecture of your application, which separates out the different parts that need to be implemented and that is what MVC ultimately is trying to do.

(Refer Slide Time: 16:44)



Present Context

- MVC is a good conceptual framework to understand separation of concerns
- Breaks down if applied too rigidly
- The web in general does not have the close knit structure of GUI applications needed for MVC
- Other aspects like static typing and type inference of objects also broken in Python-like languages

Apply basic learnings from MVC, but be prepared to stretch

So, like I said MVC is a good conceptual framework. But it breaks down if it is applied too rigidly. So, be aware of that do not try and say, is this MVC? Or is this not MVC? That is not a useful question to ask in general. Part of the reason for that is, especially in the context of web apps, they do not have the close knit structure of GUI applications, the server and client sort of being on the same system and therefore the client server, knowing what state the client is in and so on.

There are also a few other things. So for example, you will find that ASP, the programming language from the ASP.net from Microsoft, actually has a lot of I mean, they have tried to stick to the MVC concept quite closely. But what happens there is they also sort of take it further and in fact, infer some information from model types that are directly influenced certain views and certain controller architectures and so on.

So, there is this concept of the type of underlying object, which can be used in a language like C sharp, which is used for ASP.net, which does not really exist in Python. So, Python does not have a concept of static data types, that is to say, data types that are declared upfront before the programme starts running and because of that, there is a lot of type inference and things that happen in Python, which tends to break some of those ideas from ASP.net MVC when you try applying them in a Python like framework.

Now, why are we still choosing to use Python because it is a relatively easy language to programme and developing and even though I said that there are certain issues such as static typing and so on, as long as you understand that MVC is more of a conceptual framework and not something to rigidly stick to, you should still be able to benefit from the ideas. So, apply the basic learnings from the MVC model, but be prepared to stretch them in different ways.