

# **IIT Madras**

## **ONLINE DEGREE**

**Modern Application Development – I**  
**Professor Nitin Chandrachoodan**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Madras**  
**Group Actions by Controller**

(Refer Slide Time: 00:16)

Controllers?



Hello, everyone, and welcome to this course on Modern Application Development. So, now let us get to the topic of what exactly is a controller. So, far, we have had a lot of discussion about actions, requests, and responses, APIs, and the crud as an example. But it is still sort of bouncing around the topic of what exactly is the controller? Why do we have that specific? What do we need to infer over there?

(Refer Slide Time: 00:37)

Actions vs Controllers?



- CRUD etc are a set of actions
- Other actions:
  - send email
  - update logs
  - send alert on WhatsApp / Telegram
- Can actions be groups together logically?

**Controller!**



And this is where it becomes a little bit tricky. As I said, lot of the, if you stick too closely to what exactly is the definition of a controller, you might sort of get lost in what you are trying to implement. The first thing to understand is what are actions? And then why would you think of the controller as something specific, so the CRUD as an example, set of actions, Create, Read, Update, Delete, they are actions, they are things that you would request the server to do on your behalf.

And they may or may not result in changes to the database, but will sort of result in something coming back from the database, which then needs to get displayed appropriately. But there are certain other actions also that can be performed by requesting something in the server. So, for example, I might have something which sends an email when I invoke a certain action, or it might write some information to a log file.

Or it might send an alert, using some other kind of API on WhatsApp, telegram, something like that. Or in the case of a quiz or assessment. I mean, it might do something like, checking a particular answer or requesting additional time for the quiz, various other things that are not directly creating, reading, updating, or deleting a specific object within the database. That could be sort of side actions that are happening out there.

So, one question that we can sort of ask is, other actions that can be grouped together logically, and if so, it then starts to make sense to think of that as that set of actions as a controller for a particular type of model. And that is the most common way in which you will see controllers used in practice. Having said that, in general controllers are used to implement any kind of action.

So, almost anything that you want to ultimately be called a controller, but they become useful when you are thinking in terms of one common set of actions, which is logically connected together. And which relates to one particular data model, perhaps.

(Refer Slide Time: 02:55)

### Example: Laravel PHP framework

# Actions Handled By Resource Controller

Verb	URI	Action	Route Name
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy



So, this is an example from a framework, which is in the PHP programming language. And it is a framework called Laravel. Some of you may be familiar with it. And Laravel tries to implement the MVC framework over there. And in fact, they have this thing called a resource controller, which is, once again, one of the design patterns that they provide, you can essentially implement something called a resource controller.

And let us say you define a resource controller to handle a certain class of objects called photos. What it says is that just by essentially associating the resource controller with the object photos, you automatically sort of benefit from a large number of actions that are sort of predefined for you. What kind of actions? Index. What do you think index would do?

To me, when I hear the term index, it is sort of at least given the context in which we have looked at different things, index probably would be a list of all the photos that I have. Create, nice, simple name. Create a new photo. So, does that mean click something on a camera? Probably not. In this case, we are talking more about the database end of things, not about the action of taking a photograph, but about storing it somewhere.

So, create in this case, most likely means create the storage space to save a photograph, and take the binary data from the camera or wherever else it is, and put it into that database. Now, there is a distinction between create and store, as you can see. That is a slightly subtle distinction, but it depends on the framework itself. And what it usually means in a context like this is this create is just the first part of the story.

It has created an object in the memory of the server. And you can now manipulate that object, you can assign certain binary data for the photograph, you can give it a title, you can give it a caption, you can give it additional maybe geographic information, where was it shot time of shooting, maybe tags, who are the people, associated people, in the photograph, things like that.

All of that is part of the creation. But there is a separate step which stores the information. And that, as you can see the differences. It comes in POST. So now, what is the difference over here, you can see that if I use the verb, GET, look at this term, they are using verb. And as you can see, the verb essentially refers to the HTTP request parameter. Is it a GET or a POST or two new things out here? PUT or three new things PUT or PATCH or DELETE.

Things that we have not seen before. So, what do you think would happen if you called GET /PHOTO/CREATE? It looks as though it would probably just create information. But most likely, it also requires some additional information out here, which tells you a little bit about what to create, because otherwise, it is just going to create a new photo. And it is not quite clear what to put over there.

Because a GET request is just that it is a HTTP URL, I could literally call server name slash(/) PHOTO slash(/) CREATE, it would create something for me. But what, I am not given the information. On the other hand, when I have a POST, inside the post, it would correspond to some kind of a form, most likely, I would have a file upload field, where I would specify the file to be uploaded and stored. That is when it stores it somewhere in the database.

Now, I could further actions, I could request a show, which basically says, take some photo ID, you can see this parameter and the curly brackets over here, which is most likely going to be a number, it might be some other kind of identifier that uniquely identifies the photo. The reason why I say number is because that typically corresponds to the primary key used inside most databases.

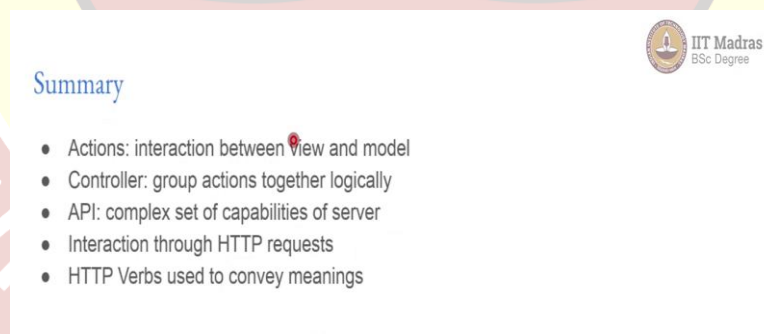
But some databases might not use a number they might use some kind of combination of alphabets or something like that, some kind of a unique key, uniquely identify, unique identifier, that is all you need. And if I GET slash(/) PHOTO slash(/) PHOTO ID, that is great. It basically should just show me that photo. How exactly does it display? Well, you need some view corresponding to that.

So, what you can see over here is each of these actions can essentially be thought of as a controller. Why? Because a user could invoke an action by calling one of these URLs, that would go into the server, the server would then do something with the database, either create a new object or retrieve an object. And it would then invoke a view, which comes back to the user, the view might be in the case of created says, successful, great, you stored it, or you saved it.

Or in the case of a GET, the show, it will probably show you the picture. So, what kind of view comes back as a result of invoking a certain controller, that is also some information that is known to the controller. It can be decided by the controller at the end of the day. There are more such actions, write, edit. And similarly, update, I might want to update, let us say, the metadata or I might want to update the picture itself, I change the lighting, I do a few edits to it, and save it back.

And finally, I might just want to delete it. So, you can see that all of these actions are implicitly created for you the moment that you use one kind of standardized controller in this framework. So, that is what I meant by saying that controllers have their maximum utility when you are trying to do things which are logically related and have some commonality to them.

(Refer Slide Time: 08:57)



**Summary**

- Actions: interaction between view and model
- Controller: group actions together logically
- API: complex set of capabilities of server
- Interaction through HTTP requests
- HTTP Verbs used to convey meanings



So, to summarize all of this. Actions essentially are encoding the interaction between the view and the model, the view essentially presents you with certain links or buttons that you can click, they invoke actions on the server, those actions could either modify the database or



just pull data out of the database from the model and will then send back a new view for the user PC.

When you can group some of these actions together logically is usually when you call it a controller, although you might also entirely well find that you have a controller which just performs a single action. So, controller and action can sort of be thought of as very similar concepts. And for all practical purposes, at least in the context of this course, I will not be differentiating too much between them.

Now, when you start grouping certain actions or controllers together, you can construct a complex set of capabilities for the server, which is usually exposed in the form of an API, an Application Programming Interface. And most important part of all of this is for the web applications that we are looking at, all of this interaction is going to happen through HTTP requests. Purely through HTTP requests, not through any other means.

And the HTTP verbs, so GET, POST, PUT, PATCH, DELETE, I think there is an UPDATE also, are used in order to convey certain meanings. But if you look at the underlying ideas, over there, those verbs are not fundamental to the, what is more, important is, as long as both sides have agreed that a certain request in a certain form to a server will result in the certain kind of action.

As long as the client and the server both know and understand that, that defines your API, whether you choose a GET verb or a PUT verb or something else over there is not the most important part. HTTP just provides you certain mechanisms, it does not enforce, it does not say, for example, that you have to use only get in order to retrieve data. Or for that matter, that you need to use delete in order to delete a certain object, I could use a GET request to actually invoke an action that deletes an object in the database, that is perfectly valid and perfectly fine. As long as both sides know that this is what is expected.

(Refer Slide Time: 11:36)



### Rules of Thumb

- Should be possible to change views without the model ever knowing
- Should be possible to change underlying storage of model without views ever knowing
- Controllers / Actions should generally NEVER talk to a database directly

In practice:

- Views and Controllers tend to be more closely interlinked than with Models
  - More about a way of thinking than a specific rule of design



So, what is most important over here is that you sort of think of this in terms of certain rules of thumb. Rather than saying, this is precisely how something needs to be implemented. One rule of thumb is, it should be possible to change a view and the model should never need to know about it.

Another way of thinking about it is that, if I have gone from a mobile phone display to a desktop, the underlying model should never know or care, the only thing that could possibly changes which controllers are used in order to are invoked in order to get data back, and more importantly, which views are sent back, the controller has a part to play over there. The controller definitely needs to know what kind of view to send back, after retrieving certain data from the database.

And similarly, it should also be possible to change the underlying storage model. So, for example, if I want to go from an SQL lite database to a MySQL database, or PostgreSQL database, the views should never know that there was a difference. And this is, in fact, where the ORM, the object relation manager, like what we are going to be using as SQL alchemy, in the implementations comes into play.

The ORM, that mapper essentially sort of abstracts away the database from the user. So, that you do not need to worry about which is the underlying database or what kind of requirements it has, the views do not change based on that. So, both of these are important sort of rules of thumb to keep in mind while designing the system. One other thing is that the actions or the controllers, generally never talk to a database directly.



What I mean by that is, if you find yourself writing an SQL query directly inside what looks like controller logic, you should probably pause and say, no, this is not a clean separation of functionality. The only thing that is allowed to talk to a database is a model. So, that is where the functionality sort of that separation happens very cleanly.

Now, this part of it actions never talk to a database directly is very important to keep in mind, that keeps one part of the separation very clear. On the other hand, what you will very often find in practice is that the views and controllers tend to be a lot more closely linked. What I mean by that is, you cannot, for example, say that the controller should never know which view to send back, that does not make sense, the controller has to know which view to send back.

Which means that if I change your view, I might potentially need to go and change controller code as well in order to know that this is the right view to invoke given a certain action or a certain result of an action. And you will often find that people tend to link these a little bit more closely the views and controllers are sort of more tightly coupled, controllers need to invoke certain views depending on what response has come from the model and bottom line is this.

At the end of the day, this is more about a way of thinking rather than specific rules of design. You cannot say that I am following MVC therefore, I cannot do this, or I must do it in a certain way. If it turns out that that is making the design too complicated, it may not be the best way of doing it.