Code School: Digging into Django

by Sarah Holderness

Templates & Inheritance

Delving into Grids

(Music) Deep within the jungle lie treasures most unknown Track will we uncover, we'll need to use Django AJAX, forms and grids, we'll cross the pyramids Dust off pure old Durango, We're digging into Django. - Hi, I'm Sarah Holderness and this is Digging into Django. This course builds upon the concepts taught in Try Django. And for Django in general, it's helpful to know basic Python, HTML and database concepts, which you can brush up on in these courses. In Try Django, we created the TreasureGram site, which displayed a list of the treasures we've collected along our expeditions. In this course, we'd like to add the following new features to our TreasureGram app: a grid layout, a detail page, a form for adding treasures, image uploads, user authentication, user profile pages and a like button with Ajax. To start, let's look at creating a new look for our app with a grid layout. So the plan here is to use Bootstrap so we can create a grid with columns of width four, and then we can create a new row every three treasures. If we look at the current code in index. html, we'll want to start by getting rid of the table that stored our treasure's details like the value, material and location since those will now be on the detail page. And we'll want to keep the homepage simple with just images. Then if we look at the remaining code in our forloop that displays each treasure, we have a panel with the heading and the image displayed. That whole piece of code is what we'll want to put in each of our rows and columns. We've collapsed

the code for displaying the treasure to make it a little easier to read, but what we've done is add a row outside of the forloop and a column inside of the forloop. If we take a look at our grid right now, we can see there's a good start with the column set up but there are a few problems. The problem is since some of our images have different heights, letting the columns overflow doesn't really work. To fix this, we can create a new row every three treasures. But how would we do that? We can check and see how many iterations of the forloop we've done and see if that number is divisible by three, then we know we want to create a new row. To see the number of iterations of the forloop, we can use this special variable inside of the forloop. Forloop and then. counter will give us the number of iterations so far. So for the first generation we'll have one, as the counter then two, then three. Then we want to check if the counter is divisible by three, and Django provides special filters that we can use. You use a filter by writing the value on the left, |, then the filter name, which here it's divisible by, and the argument which is three. This will return true or false, depending on whether it's divisible by three or not. So if we put this all together in an if statement, we end up with something that looks like this. And inside, if it is divisible by three, we'll end the current row and start a new row. And now we have this piece of code we just wrote inside of our template, which is going to end the last row every three columns and then start a new one. And now we can see our grid is working even if we have images of different heights. In case you're wondering, there's some other forloop variables we can use. We already saw. counter,. counter is zero will start counting at zero, and. first and. last will return true or false if this is the first iteration of the loop or the last iteration of the loop. There are also a lot more filters available to us. For example, for add, you can do | add and a number, and that'll return the value +2. You can add |lower after a string, and that will return that string in lowercase. And you can even chain multiple filters to get different results. There are many more Django template filters, which you can read about in the docs here.

Delving into Details

(Music) Welcome back. In this section, we'll be adding a detail page. Now that our homepage is displaying a grid of just images, we'd like to be able to click on one of those images and be brought to the detail page to show more details about our treasures. The URL for the detail page will be localhost/ id of the treasure we're displaying. For example, here we have localhost/ 15, which is the id of the golden monkey displayed. For the new detail page, we'll need to create this new URL in urls. py, create a detail view in views. py, and create a detail. html template. We'll start by adding our details URL. This URL will capture any number, save that number, and then send it to the detail view. This regular expression is a little more complex than the one we saw before, so

let's break it down. In our existing index URL, we're not checking for any path, which means it would just match localhost. But now we want to capture a number, and how we do that is with square brackets zero through nine plus, and that will match any number. So then we just add parentheses and that will capture that number, which means it will then be passed as a parameter to the detail view. For instance, localhost/0, that zero would be captured and then passed to the detail view as a parameter. So we said that this number would be captured by the URL and passed to the detail view. So when we create our detail view, we'll have two parameters. Request, and then whatever we name that number, and we're going to call it treasure id. Now we want to get the treasure that corresponds to that id. So we can do Treasure. objects. get and then check if that id is equal to treasure id. You may be wondering where this id field came from in the model. Because if we look at our model, there is no id field. So how did we look up a treasure by its id? By default, Django automatically provides us with an auto-incrementing primary key field. That means we can look up a specific treasure with this id field, and every treasure is guaranteed to have a unique id. Back in the detail view, we can pass the template the treasure we just looked up. Now that we have the URL and views set up, the last step is to put together our template. This is a new template called detail. html. And because we passed in the treasure object, we can access all of its fields like name, image URL, value and location. We haven't added any style yet just to see that it's working. To add some style to our page, we'll basically copy the same styles we added to the homepage but we don't need a forloop because we're only displaying one single treasure. I know we added a lot of HTML and CSS here but it's specific to the design we've created for our page and you don't have to memorize it or anything because each app you create will have its own HTML and CSS elements. So now we have all of the pieces to our detail view in place: the URL, the view and the detail template. But we're missing one part, we need a link from the homepage to get to the detail page of each treasure. We can do this by wrapping the entire panel in an anchor tag that links to / treasure. id. That way, if you click anywhere on the panel, you'll be taken to that treasure's detail page. Now we can see this in action by clicking on a treasure on the homepage and being taken to that treasure's detail page.

Inspecting Template Inheritance

(Music) Welcome back to level one, section three, where we'll be covering template inheritance. Each page on our site will have repeated code for at least the nav bar, the header and the footer. It would be great if we could share this repeated code across different templates. To do this, we can create a base template that will hold the repeated code for the nav bar and footer that other templates can then extend. In this base template, we'll start by loading static files. Then we'll have

all of the repeated code for the nav bar, and then we'll create this block that will assign a name to, we'll call ours block content. This is a special template tag that's a placeholder for all of the content from all of our other templates. So we're actually not going to fill this block in with any code. Then in index. html, we'll start by extending the base template and then we'll write all of the code that's specific to the index page. And that will be wrapped by this block tag that will tell Django to put all of this code where it belongs in the base template. So the final rendered index template will have all of the nav bar, then the copied in index code and then the footer combined into one template. So what's the actual code we would put in our base.. html template? We start with loading the static files, then we put in our head tags and we load our CSS files. We'll add our nav bar and we can even put in the main tags that will contain the block content from each template. And then finally, we'll add our footer. Then an index. html will extend base. And then inside of our block content, we'll add all of the code for displaying our treasures. The detail template will look very similar, except it will have code specific to the detail page for displaying a single treasure. If we look at this in action, nothing has changed. But behind the scenes, our code is more organized. So if we change something in the nav bar for instance, we only have to change it in one place, in the base.

Forms

Digging up Forms

Deep within the jungle lie treasures most unknown The track we'll uncover We'll need to use Django AJAX, forms and grids We'll cross the pyramids Dust off your whole Durango, We're digging into Django. - Welcome back to Digging into Django. This is level two where we'll be going over forms. Right now there's no way for users to add treasures in our app. In Try Django we were adding new treasures through the admin or the Django shell but we don't want regular users to do that. Instead we can create a form to show on the home page that will allow users to add new treasures. That way only staff or dev can interact with the model through the admin or shell. To use a Django form we'll create a new form class called TreasureForm which is very similar to when we created our treasure model. In forms. py we'll first import forms from Django and then we'll create our TreasureForm class that inherits from forms. form. We'll have similar char fields and decimal fields that will map to HTML input elements just like our model fields map to a database table. Just like we did with model fields we can set the max_length and the label to

display in the HTML. By setting these, Django takes some of the work out of creating form fields and validating form data. Before we get to displaying our form let's add a URL that will handle our form submission. We'll call this post_url so localhost/post_url will go to the view post_treasure which will handle our form submission. You may be wondering why our URL was called post_url. Well, there's two main types of requests. A GET and a POSt. By default, all of our URL requests are of type GET. For example, if we make a request to the server for a treasure with a specific treasure ID that's going to use a get request and return that treasure back to us. However, with our new treasure form when we press Submit that's going to send a treasure that's going to add an object to our database. So, since it's updating the server we'll want to use a POST request. That's why we named our URL post_url. Now in our views we want to start with creating our post_treasure view but before we do that we want to import the TreasureForm we just created. Then we'll create our post_treasure view that takes in a request, it then creates that TreasureForm from that request. post submission and then it checks if the form is valid, then if the form is valid, we'll go ahead and create our treasure by looking up the attributes in form. clean_data. Remember that Django takes care of cleaning the data and validating it for us. So, now we can save that treasure to the database by doing treasure. save. Then outside of the if statement we'll redirect to the home page. That way even if the form wasn't valid, we'll still redirect to the home page. We'll also need to update our index view so that it can create our new form and pass it to the index template for rendering. Now we've handled the form submission but we haven't actually displayed the form in the template. At the bottom of our index. html template we'll add our form since we want it to show up at the bottom. We'll add a form HTML element with the action going to our post_url and the method is POST. Then inside that we can simply display our form with curly bracket and the variable name form. Notice we also have to add this special Django template tag csrf_token. Django handles a lot of security tasks for us like this token which protects against cross-site request forgery attacks. This is just one of the ways that Django helps protect your apps with simple functionality. And now we can see in the HTML form that was automatically generated for us. We can also add some style to our form so it fits in better with our home page. We'll wrap the form in a panel, add a title and then we can also display the form with form. as_p which will format the form like an HTML paragraph with each field on a separate line. And if we look at the HTML source for this page that was generated we can see that all the labels and input fields are filled in for us. It even has the csrf_token which now has a value filled in and it even provides the max_length and type variables to validate the data for us. And now we can see our new form in action. We can enter data into these fields and submit and we see our new treasure displayed. You may be wondering if Django forms can render anything other than just simple text fields. Django form widgets can render more complex input

fields like password input, date pickers and file input and they're as easy as adding a widget to your form field. And we'll see more widgets like this later in the course.

Smarter Forms with the Meta Class

Welcome back to level two, section two where we'll look at creating our forms in a smarter way using meta classes. We'll end up with a form that has the exact same functionality as before but with a lot less code. If we compare to our original form code there will be a few differences. First, we'll need to import our treasure model since we'll be referencing it and our TreasureForm class will now inherit from forms. ModelForm which is the first step to linking a form with a model. Notice we don't have to create any of the fields as we did before. Instead we'll just use this class called meta. The name in this class will always be meta and we'll use it to link our new form to our already created model. Inside the class we'll specify that our model will be treasure and the fields will be a list of all of the model fields we want to show up on our form. Now that we've changed our form to be a model form, we'll go and update our view. We can replace all of the code we had before for manually creating a treasure and saving it with this one line, form. save(commit equals true) and in this single step we're reading all of the data from the form fields and saving it to the database. Even though we're doing save on the form this is actually creating a treasure object for us because the form is linked to the treasure model. Afterwards we'll still redirect to the home page. Now you can see that our form is working exactly as it was before but with a lot less form code.

Image Uploads

The Image Field

Deep within the jungle Lie treasures most unknown Track will reuncover We'll need to use Django AJAX, forms, and grids We'll cross the pyramids Dust off your whole Durango We're digging into Django. - Welcome back to Digging into Django. This is level three where we'll be covering adding the ability to upload images. Users can now add treasures through the form we created in level two but they still need to have an image URL of where their image is hosted and we like them to be able to upload an image directly into the app. To do this we'll want to use an ImageField and there's a few steps to getting that working. First, we'll add the ImageField to our model, then we'll need to install Pillow to use that ImageField. Then we'll migrate our changes to the database.

We'll update settings. py to add a media directory. And finally, add a special URL to urls. py to serve our media files locally. So, let's do each of these steps in order. First, in our treasure model we'll get rid of image URL and we'll add our new ImageField which has two required parameters. Upload_to is the location inside of the media directory where we'll add our images to. And default is a default image in case the user doesn't upload one. So, since we just changed the model you'd think we want to run makemigrations but if we try to do that we get this error that Pillow's not installed and we get some information on how to install it. We can install Pillow with pip install Pillow. This will download it and install it for us. Now that Pillow's installed we can run Python manage. py makemigrations which will add our ImageField. Then we can run Python manage. py migrate which will finish migrating our changes to the database. At the bottom of settings. py we'll add this media_root variable that will tell us the location of where to store all of these user uploaded files. So, all our images will go in that media directory and media_URL handles serving the media from media_root. Because we're running our app locally we need to add this special URL to our urls. py file. This will match any image in media/ and will serve that file locally from the media_root directory we just created. Now we finished all the steps to set up an ImageField but we still need to refactor a few things to replace our old image URL with our new ImageField in forms. py, views. py and both of our templates. In forms. py we'll just replace our image URL with our new image. And remember, since we declared that ImageField in our model, it will pull over all of the associated properties and the fact that it's an image field. So, now in our post_treasure view we'll also need to pass our request. files to our form. Finally, in our index template we can change the enctype to multipart/form-data which is required when submitting files through a form. And then in both our index and detail templates we'll need to change the old treasure. image_url to our new treasure. image and then add. url to get that images URL field. So, now our image field is ready to use and we can see that we can create a new treasure and upload our corresponding image and then we can see that new treasure displayed.

Users

The User Model

(Music) Deep within the jungle lie treasures most unknown Track will we uncover, we'll need to use Django AJAX, forms, and grids, we'll cross the pyramids Dust off your old Durango We're digging into Django. - Welcome back to Digging Into Django, this is level four where we'll be

covering users. Right now, we can upload a treasure to our app, but our treasures don't have any owners so we don't know whose treasures are whose. Well, we can solve that by adding a relationship to a user model from our treasure model, so that's what we'll be doing now. In databases, there are three different types of table relationships: one-to-one, one-to-many and many-to-many. We want a user to have many treasures associated with their account, so we'll use a one-to-many association. The way to relate two tables together is to mark a field in one as a ForeignKey in another. So in our treasure model, we'll have a user field that's a ForeignKey that belongs to the user model. But you may be thinking we don't have a user model. Django actually creates a user model as part of the auth module, which is what we use to log into the admin. So we'll be using that and we'll import that here. Then we can create a user field that's a ForeignKey to the user model like this. So now that we've made this change in our model, we need to migrate these changes to the database. We'll run python manage. py makemigrations. This will tell us that we're trying to create a non-nullable field user and ask us if we want to provide a one-off default now. We'll select one to provide a one-off default and then we'll enter one, which corresponds to the first user in our user table. That will then create our migration file. Now we can apply that migration file by running python manage. py migrate. Now we can create new users through the admin so we'll create a new user called Indiana and set up all his default information, including his password. For any new treasures we create, we'll want to assign a user to those treasures. But for our existing treasures, we can use the admin to go through and assign users to those treasures. Now if we go back and try to add a treasure through our form, we have this problem. We get an error. And that's because we have this non-nullable user field and we're not adding not through the form. So we'll want to add the user to the treasure in our view. Now if we go into our post treasure view, we'll want to change a few things. First, we can call form. save without committing it, and this will return a treasure, which we can capture in a treasure variable. Then we'll assign the user to the treasure as the request. user, since user is sent with the request. And then we can save that treasure, and now that treasure has a user associated with it. So now that we have this user field set up, we can display a profile page for each individual user that will show all of the treasures that belong to that specific user. To do that, we'll want to go through the steps we've seen before for creating a new page, like setting up the URL, the view and the template. For the URL, we want localhost/user/user name to go to a user name's profile. For the view, we'll want to look up all the treasures that belong to that user name. And in the template, we'll display all of those treasures. First, we'll create our URL with a regular expression that matches user/ and then a word of any length. \w+ will match the word of any length, and then adding parentheses around that will capture that value and then we can send it as a parameter to our view. In that view, we'll now have two parameters, the request and the user name. We can use get to look up

the user object in the user model by its user name and checking if it matches the passed in user name. The next step is to get all of the treasures that belong to that user. We can use a filter on Treasure. objects with that new user object to get all of that user's treasures. And finally, we'll pass that user name and treasures lists to our profile template for rendering. Then we'll create the profile template, which will look almost identical to our index template to display the user name as a title and then display all of the treasures that belong to that user in a for loop. We can see that the user profile page is working for our Indiana user, but we can only get to this page by manually typing in the URL. So we want a link to this page from the homepage. We'll need to add an anchor tag underneath our title, which will link to /user/treasure. user. username, and the text will just be the username. Then to link to our detail page now, we'll add an anchor tag inside of the panel body around the treasure photo so when you click on the treasure photo, you'll now go to the detail page. Finally, we can see our homepage linking to our new profile page.

User Authentication

(Music) Welcome back to level four, section two, where we'll cover user authentication. Now that we have users that can create treasures, we like to add the ability for only logged in users to be able to create treasures. So we like to add login and logout functionality to our app. We'll start by adding a login page, and the plan for doing that is the same as usual with adding a URL, view and template, but we'll want to add one more step for adding a login form. We'll start again with adding our URL that matches /login and goes to our view login view. We can't name our view login because there's a built-in login function we'll be using later. Now we'll want to create our login form. It will inherit from forms. from and will have two fields for the user name and password. The username will be a CharField with a label for username and a max length of 64. The password will also be a CharField but will use that password input widget that we talked about earlier. Now we'll go to creating our login view, but first we'll need to import our login form we just created and the built-in login, logout and authenticate functions. Because we want that option to either post or submit in login form data or simply display the login form, we'll check if the request was a POST and then authenticate from there. Otherwise, we'll display the login form. This is a little different than how we handled posting a new treasure form because in our index view, we were displaying the form; and then in our post treasure view, we were processing the form data in those separate views, and now we're doing that in just one view. First, we'll create our login form, check that the form is valid, get our username and password from the form and then authenticate with that username and password, which will return the authenticated user. We'll first want to check that the user is not none and the user is active, and then we'll login the

user with request and user using the built-in login function. And then we'll redirect to the homepage. We can also provide feedback without statements by printing out whether the account has been disabled or the user name and password were incorrect, and we'll just print those statements to the console for now. Finally, we'll create the login template by simply displaying the form inside of form tags, and we'll still need that csrf token we used before for posting. You can see the method is post and the action means that we're just submitting back to this same URL. Now you can see our login form working on our login page.

Logging Out

(Music) Welcome back to level four, section three. We're almost done with user authentication, but we'll also want to add the ability to log out and links for logging in and logging out. To add logout functionality, we'll just need a URL and view but not a template. The logout URL is pretty similar to the login URL except named logout. In our logout view, we'll use the built-in logout function that takes in the request and then we'll simply redirect back to the homepage. But right now, if we go to our homepage, we don't have any links set up to login or logout so we'll want to add those. Since we want those to show up on every page, the best place to add those is in our base. html template. And we'll add those right after we display the TreasureGram logo. So we only want to show a login link if we're not actually logged in. And then if we are logged in, we want to show a logout link and a link to our profile. And we can accomplish this with a big if else block. If the user is authenticated, we'll show the link to their profile and a link to logout. Otherwise, we'll just add a link to login. Before we were hardcoding any links in our templates, but there's a better way to do it. You can use a URL template tag, and you can do that by writing url, the name of the URL and any parameters you want to pass to that URL. So here we have the profile as the name of the URL and user. username that we want to pass there. So if my user name is Sarah, the URL will look like this. Then we can add the links to login and logout with the same URL tag, except they don't need any parameters. And we can also add a little style by adding our CSS classes. Now we can see our login and logout process in action.

Ajax

Creating a like Button

(Music) Deep within the jungle lie Treasures most unknown The track will we uncover We'll need to use Django AJAX, forms, and grids We will cross the pyramids Dust off your old Durango We're digging into Django. - Welcome back to Digging into Django. This is level five, section one, where we'll be going over AJAX. We'd like to add a Like button to keep track of the number of likes for each treasure, whether you're on the homepage or the detail page. We'll also want to use AJAX for the requests so that the page will display the new number of likes without having to refresh the entire page again. So it's probably pretty familiar to you by now, but we'll want to create a URL, view, and template. So the plan for creating this Like button will be to create a URL at localhost/like_treasure that will go to our like_treasure view which will then calculate the total number of likes and update the database. And then our index and detail templates will display the Like button and the updated number of likes. And then we'll add these two extra steps, add AJAX to process our request and we'll add a like field to the Treasure model so it can save a treasure's likes. Let's tackle our model first since we'll need to access the likes and the view and the templates. We'll add likes as an integer field, and we'll set its default to be zero. We can then migrate this change to the database by running python manage. py makemigrations followed by python manage. py migrate. Then in both of our templates at the bottom of the treasure panel, we'll add a Like button. This will use Bootstrap's button class as well as glyphicon to display a heart. And for showing the number of likes, we'll check if the likes are greater than zero and then display them, otherwise we won't show anything except the heart icon. Before we look at updating the view, let's look at this process without using AJAX or the process that we're used to. Let's look at the process for when a user requests the homepage. The URL dispatcher will then send that request to the index view which will look up all of the treasures in the database and then will render the template with all of those treasures and generate the corresponding HTML. Then any click to links within that page will go through the URL dispatcher again. So if we want to add AJAX to prevent the page from refreshing when we click the Like button, things are going to change a bit. Parts of this process will still be the same. We're still going to load the index. html file. But when the Like button is clicked, the JavaScript is going to intercept this click event. The JavaScript will then get the ID of the treasure that was clicked and pass it through the URL dispatcher to our new like_treasure view. Our like_treasure view will then look up the treasure by its ID and save the new number of likes. It will then take that number of likes and send that back to our JavaScript. Our JavaScript will then update the Like button with the new number of likes. We could write this AJAX in standard JavaScript, but we're going to use jQuery to help us because it has a lot of AJAX built-in functions. So we'll download the latest version of jQuery and put it in our main_app/static/js folder. And then we'll also create our main. js for writing our application-specific AJAX code. To include these new JavaScript files, we'll want to add them to

base. html, and then they'll be available to all the templates that inherit from base. In our main. js file, we'll start by adding a button click listener. And when that button is clicked, it will prevent the default action and will capture that button in a variable. We'll then add a jQuery AJAX function which takes in the parameters url, which will go to our like_treasure url. We'll use a GET request. And the data is any information we want to pass to the view. We want to get the treasure_id and we can get that by looking at the data-id attribute of the clicked button. You might have missed it, but we created this data-id attribute inside our button in the index template. So now we can see in our diagram that we just got the treasure_id and sent it to our url. Now we'll need to actually create that like_treasure url in the corresponding view. Our new url will be like_treasure, and we'll go to the like_treasure view. In our new like_treasure view, we'll start by getting our treasure_id from the request. Then we'll initialize a local likes variable set to zero. Then we'll check if our treasure_id was actually received from the GET request. And if it was, we'll look up the treasure that corresponds to that ID. So then if that treasure is not none, we'll look up its number of likes and increase them by one and save that into our local likes variable. We'll then update that treasure's number of likes and save it to the database. Finally, we'll return our likes as an HttpResponse since that's what our AJAX is expecting. If we look at our diagram again, we can see we now got the treasure by its ID in our view, and then saved its number of likes to the database, and then we returned that new number of likes back to our AJAX. Back in our AJAX we need to capture that returned number of likes, and we can do that in the success function. We'll then update our button element with the new number of likes. Notice that number of likes that got sent from the view is captured in this variable that we called response. So if we look at our AJAX diagram again, we can see we completed our final step of updating the Like button with the new number of likes. And now we can see our new fancy Like button in action. It instantaneously updates without refreshing the page. And if we click into the detail view, the number of likes is still correct.

Code School Digging into Django

(Music) Welcome back to level five, section two. Remember how in level two we said that a GET should be used when we want to retrieve data from the server, but a POST should be used when we want to send data to be saved to the database? Well, we'll want to change our AJAX to use a POST. We'll want to use a POST because we're updating the treasure with the new number of likes in the database. So there's just a few changes we need to make to change our GET to a POST. In our main. js file, we'll change the request type to a POST. In our view, we'll also change request. get to request. POST. But now if we try to run this, we see that we get an error because a

Django POST needs a CSRF token. But since we're doing the POST through AJAX, we can't just put the CSRF token in the form like we did before. So if you remember from level two, we had a form POST with a CSRF token added in the template. For AJAX, we can look up the CSRF token in the user session cookie, and then add that to the POST header. The code to do this is kind of crazy, but luckily Django provides this code for you on their docs website. So we're going to look at it so we can see how it's working but we're not going to go into great detail. The first function we find in the docs is this getCookie function that will take in a string and look up the corresponding cookie. And we can see we use this below by creating a CSRF token variable that will store the cookie returned by this function. So we want to add this CSRF token to every AJAX POST request. So we're going to use this ajaxSetup function that takes in a beforeSend function that will append the CSRF token to the RequestHeader. This beforeSend function will run with every AJAX request. Now we can see our Like button is working exactly the same as it did before, but we're using a POST request like we should be.

Course author

Sarah Holderness

Sarah just finished her PhD in Computer Science and in that process found two loves: inventing things and teaching. Working at Pluralsight allows her to do what she loves and work with some amazing...

Course info

Level	Intermediate
Rating	★★★★½ (28)
My rating	★★★★★
Duration	0h 54m
Released	7 Sep 2016

Share course