Concord

Release 0.1

Shauna Gordon-McKeon

Sep 27, 2020

MORE INFO

Concord is a Python (Django) library that can be used by developers to build sites with comprehensive, customizable, dynamic, and most importantly *user-determined* community governance. It can be added to existing websites with Django backends or used to create a new site from scratch.

Contents

CHAPTER

ONE

MOTIVATION

At its heart, governance revolves around a single question: who gets to do what? Take the United States Constitution. Article 1, Section 8, Clause 2 says Congress can "borrow Money on the credit of the United States". Congress is the who. Borrowing money is the what. The more famous First Amendment says Congress can't establish a religion. Congress, again, is the who, and establishing a religion is the what - something they don't get to do.

There's more to governance than simply laying out these rules: there's wrangling over definitions, questions of fair and unfair enforcement, new rules to be made and old rules to be changed, as time and circumstances and beliefs about right and wrong shift. But "who gets to do what" describes a lot of the most important questions we ask about governance.

We learn about governance from the society we're born into, and every culture has different models of governance it emphasizes. In the United States, the two most common models of governance are representative democracy and dictatorship. Although other governance models exist (for example, in jury selection, or in the oligarchical structure of many non-profits), representative democracy and dictatorship are most commonly found in our institutions and our culture. It may seem odd to claim we can commonly find dictatorships in the United States, but most for-profit corporations in the United States are run as dictatorships, and many informal projects run by their founders use a structure that the open source community calls "benevolent dictator for life".

Given the dictator-like governance structure of most corporations, it's no surprise that most digital platforms - designed and run by corporations - implement only dictatorships. Sure, a Facebook group or a subreddit may allow the owner to delegate many rights to individual users, but there is always one person who has ultimate control. While this structure reflects the values of the corporate platform, it does not reflect the values of American society, as they offer no ability to implement representative democracy. Nor does it give users access to the vast and fascinating variety of governance structures that have been tried and tested, or even just imagined, across the history of the world.

Concord seeks to enable communities to experiment with a variety of forms of self-governance, to find which structures best suit their needs. They may end up with a structure very like a dictatorship, but it will be their choice and require their consent. Or they may end up with a managed commons, or a sortition-based system like in ancient Athens, or a consensus model like those practiced by Quakers and radical activists, or, or, or...

CHAPTER

TWO

DESIGN OVERVIEW

The fundamental design units in Concord are actions and permissions. Actions contain information about who is taking the action, what they're taking the action on, and what, specifically, they want to do. These objects concretize the question highlighted above: *who* can do *what* to *what*? Permissions are the answer to that question.

Changes of state within a web app can be "wrapped" within an Action object. Action objects are then applied against zero, one, or many permissions. If any of the permissions are granted, the Action "passes" and is implemented. Users may be granted permissions individually or based on the "role" they hold, and conditions may be placed on permissions so that additional steps (approval, a vote, etc) must be taken after the permission passes but before the Action is implemented. The flexibility of this system can lead to quite complex configurations, and so we also offer templates, which allow users to implement a complex configuration with the click of a button.

2.1 Actions and Permissions

Action objects are Django models. The three most important fields on an Action object are "actor", "target", and "*change*". The actor is whoever is taking the action (always a user); the target is any Django model within the system that accepts actions (called a 'permissioned model'); and the change field details the specific changes that the user wants to make.

Once the change has been validated the Action is run through the 'permissions pipeline'. The pipeline checks to see if any "*Permissions*" have been set on the Action's target. These Permissions also have change fields which are nearly identical to those set on actions. We compare the change fields of the Action and the Permission and, if they match, we check to see if the Action actually passes the Permission.

We determine if an Action passes a Permission by checking its "roles" and "actors" fields. A permission may be granted to individuals ("actors") or to roles (simple names associated with a list of actors). If a user is either listed as an individual actor in the Permission, or if they are assigned a role listed in the Permission, they will pass the Permission.

2.2 Communities and Roles

All Actions in the Concord system take place within Communities. There can be a single large Community covering all actions on the site or, more commonly, users can choose to create their own communities on a platform.

All permissioned models are owned by a community. When we say that, for example, all users with role "administrators" have permission to make a certain type of change, we are referring to the role set on the owning community. (Our long term goals include allowing for federations of communities and sub-communities, at which point we will likely allow permissions to reference roles set on other communities.)

Communities can have an arbitrary number of roles and those roles can be included in an arbitrary number of permissions. There are three special types of roles, which all communities have: members, owners, and governors.

The member role is more or less self-explanatory: all members of the community are given the "member" role. This may grant them many permissions, or none, or something in between. But a user may not be granted any additional roles until they have the member role.

People with the "owner" role are the foundational source of authority within the system. They may do whatever they want with the community - for instance, kick out all the members or even delete it. Given this power, there should always be conditions set on their actions. We will talk about conditions in more detail soon, but for now, just imagine a community where all members are also owners. The condition set on them taking action might be, for instance, a supermajority vote.

But holding a vote every time someone wants to make a change in the system will quickly get very tedious. This is especially true when communities are young and don't have an established, stable system of roles and permissions yet. Communities may want to grant a broad discretion to certain trusted individuals. These are "governors". Governors may also have conditions set on their actions, but these conditions if they exist are almost always less onerous than those set on owners. However, if governors abuse their discretion, they can be removed by the owners.

With this added context, we can better understand the "*permissions pipeline*" that actions go through. When an action enters the pipeline, we first check whether or not the change is of a special type (eg "delete the community", "change owner of community", "change governor of community") that only the owners can take. If this is the case, we check only to see if the actor is an owner - if they are, the action passes; if not, the action is rejected. If the change is not of a special type, we proceed to checking whether the actor is a governor - if they are; the action passes; if not, the action continues through the pipeline. Finally, we look for specific permissions set on the target of the action. Once again, we check if an actor matches and passes those permissions. If they pass a permission, the action as a whole passes, otherwise it is rejected.

2.3 Conditions

If someone has a permission to do something, they will be immediately able to do it. But communities may want to grant permissions while still providing a check on their actions. For this, we use conditions. Conditions may be set on individual permissions or on the roles of "owner" and "governor" as a whole. In the later case, conditions would apply to every action a user took as an owner or as a governor.

There are currently two types of conditions available: 'Approval' and 'Voting'. When an approval condition is applied to a permission, someone trying to take an action that triggers that permission will require one person's approval. When a voting condition is applied to a permission, someone trying to take an action that triggers that permission will need to wait for people to vote on whether their action should pass.

Conditions always require configuration when they're set. That's because we always need to know who can take action on the condition. Conditions also usually have additional configuration. For instance, the voting condition lets you set a voting period, let voters abstain or not, and lets you pick whether a majority or only a plurality is needed to pass.

We have designed Concord so that it is relatively easy to plug in new condition types. We hope to add soon a Consensus condition as well as various other voting conditions like Ranked Choice, Quadratic, and Liquid.

On our roadmap is the incorporation of 'filter' conditions. As opposed to decision-making conditions, which require interactions from the community to resolve, 'filter' conditions would automatically apply existing information to determine whether the action can proceed, eg "this action may be taken if the user was created over one week ago" or "this action may be taken if the user has taken no other similar actions in the last twenty-four hours". As with decision-making conditions, they would be configured by the community and could be adjusted over time.

Learn more about "*how condition models work*" or learn *how to add a condition*.

2.4 Templates

The system described provides the building blocks for a large variety of governance systems. But it may be difficult for new users to build such systems from scratch, and even experienced governance-framers can benefit from access to the innovations of others. As such, we have a template system which allows users to apply pre-defined sets of permissions to their communities.

Templates have a variety of "scopes", which allow users to narrow their search within the template library. A template with "community" scope will likely make broad changes, for example by creating a role 'voting members' and making them owners, with a condition of a three-day-long majority vote. A template with a more narrow "membership" scope will only set permissions related to membership, for example by specifying that anyone may request to be a member of the community but they must get two approvals from role "membership admins". Each template has a "plain English" name and description, along with a detailed list of the actual changes a template will make when you apply it.

Currently, templates must be defined by developers behind the scenes, but on our roadmap is the creation of an interface for users to develop templates themselves.

Want to contribute to our templates library? Learn *how to add templates*.

CHAPTER

THREE

DEVELOPMENT GUIDE

3.1 Using and Extending Concord

Concord is build to be resuable and extensible. Currently, the three main ways to extend Concord are by adding conditions, adding permissioned models, and adding templates. These can be done as part of building a new site, or can be contributed back to the Concord project core.

When you add a new condition, it will be added to the list of conditions which may be set on permissions or on the "owner" and "governor" roles. Users will be able to configure and set it, and when an action triggers it a link to it will be created in the action history view. Adding a new condition requires: defining the condition model and the required methods (such as how the status of the condition is determined); creating change objects corresponding to actions taken on that condition, such as AddVote or Approve; creating a simple client to expose the change objects; and creating a default template for interacting with the condition on the front end. Depending on the complexity of the condition and the experience of the developer creating it, we estimate that adding new conditions takes a day or two of worker. To learn more, read the guide on *how to add a condition*.

Creating new permissioned models (also refered to as resources) is a similar process to adding conditions (after all, a condition is a type of permissioned model). Developers will create a new Django model which inherits from PermissionedModel and define their own custom fields on it. They must then add change objects and corresponding client methods to control when an instance's data can be changed. It is up to the developer how to add the permissioned model as a template - the default is "create a Vue component for this Django model" but in practice the template design will be driven by the data model. To learn more, read the guide on *how to add a permissioned model/resource*.

Currently we add templates by writing a template function in template_library.py. Adding templates can be very quick (under an hour) with the bulk of the work largely coming from adding tests to make sure the template is doing what you expect it to do. To learn more, read the guide on *how to add a template*.

3.2 Tools and Frameworks Used

Concord currently relies heavily on two languages and two frameworks. Concord's back end logic is implemented as a Django (Python) package, and made avaialble through AJAX views, which return data as JSON for the front end to use. Developers may implement a completely custom front end, but Concord provides some default templates using the Javascript framework Vue. We are working on making the templates as modular as possible, so developers can create their own front end but re-use, for instance, the interface for interacting with permissions and conditions, or the inteface for viewing action history.

3.2.1 Deep Dives

Change Objects

Imagine that your website has forums and the basic actions a user can take on your site include "add a post", "edit a post", "delete a post", "add a comment to a post", "edit a comment on a post" and "delete a comment from a post". (In reality there are a variety of other less obvious actions a site like this would have, but these will do for now.)

A developer working with Concord would create "change objects" (simple Python objects) for each of these actions. The change objects accept both required and optional parameters in their init methods, for example a "delete comment" change object would accept the primary key of the comment. In the validate method of the change object, the developer can add custom validation - for instance, in an "add comment" change they may want to automatically reject a comment that is too long. In the implement method, they define the steps to implement the change - for instance, when editing a post, the method would take the steps of looking up the post object from the database, changing the field, and saving it. The change's validate method is run just before an Action is created. If the change is valid, we create the action and run it through the permissions pipeline. If the permission passes, the implement method is called.

Change objects are part of both Actions and Permissions Items.

Permissions

Permission objects have a few different fields. First, they have a specific target they're set on - always a permissioned model. Also, as mentioned above, they have a "roles" field and an "actors" field, specifying who has the given permission. They also have a boolean (true or false) field which, when flipped, allows any user (including those who are not members of the community) to have the permission, overriding whatever roles and actors have been set.

Permission objects also have a change field and a configuration field. This optional configuration field can be applied to the change field to narrow the permission's scope. For instance, take the change "add people to role". We may want to grant someone the ability to add people to all roles, or we may want to specify that they can only add people to a given role. When creating or editing the permission, we add the configuration by specifying the role_name, that is, the name of the role we want to limit the user(s) permission to.

While developers can extend the permissions system to cover a variety of change types, there are some core types that are necessary for the basic structure of Concord to function, such as: add people to role, remove people from role, add role, remove role, add governor, remove governor, etc.

One final note: permissions objects are themselves a permissioned model. That is, permissions can be the target of actions and be subject to their own permissions. Communities content to leave the changing of permissions up to their owners and governors will not need to nest permissions (that is, set permissions on permissions) but the functionality can be useful in some cases.

Permissions Pipeline

Let's go through the steps of the permissions pipeline again in more depth, addressing some of the caveats and complexities.

We refer to the three elements of the permissions pipeline as the "foundational pipeline" (aka the owners pipeline), the "governing pipeline", and the "specific permission pipeline", and they are tried in that order.

To check if the action should enter the foundational pipeline, we look to see whether the change is of a special type, what we call a "foundational change", or when the target of the action has "foundational permissions enabled". When this simple boolean value is True, all changes of any type on the target must come from owners, although typically the value is set to False. Once an action enters the foundational pipeline, it passes or fails based on that pipeline - we do not continue on to the governing or specific permission pipeline regardless of the result.

If an action is not foundational, we proceed to the governing pipeline. Before checking if the owner is a governor, we look to see if the target has the "governing permission enabled". The governing permission is enabled by default, but it can be disabled as a way of limiting the discretion of the governors. If the governing permission is enabled, and the owner is a governor, they pass the permission and exit the pipeline. However, unlike with the foundational pipeline, if the actor is not a governor we do continue on to the last pipeline, the specific permissions pipeline.

The specific permission pipeline is the most complex of the pipelines. First, we look up all permissions set on the target and check to see whether the action passes them. If none do, we continue on to look at parent objects to see if any permissions are set on them which the action passes. By allowing permissions to be set on parent objects, we can broaden the scope of a permission. For instance, we may want to give anyone with role "editors" the ability to edit a given post. But we might also want to give them the ability to edit all posts in a given forum. We can check for permission to 'edit post' on the individual post, or on the forum as a whole.

Although typically we only evaluate zero or one specific permissions per action, an action can match multiple permissions. In that case, an action only has to pass one permission to be approved.

At the end of the permissions pipeline, the actions will have a status: approved, rejected, or waiting. "Waiting" is the status an action will have if they passed a permission or permissions (specific permission(s), governing, or foundational) but there was a condition set on it, and the condition is not yet resolved. If the status is "approved", the action will be implemented. If the status is "rejected", the action will be closed without implementing.

Condition Models

When conditions are set on a permission, the basic information about condition type and configuration are saved in the permission model. When an action passes that permission, we check and see if a condition is set. If a condition is set, we'll create a new condition model object using information from both the permission the condition was set on, and the action that triggered it. That condition must then be approved before the action can pass. Every time the condition is updated, we send a signal to the action, which attempts to pass the permissions pipeline again.

All conditions have a status: accepted, rejected, or waiting. Conditions with a status of "Accepted" or "Rejected" are considered resolved, while those that are "Waiting" are unresolved. How that status is determined is implemented by the individual condition. For instance, the VoteCondition takes into account the existing votes, whether the voting period has expired, and whether or not the condition is configured to require a majority or a plurality. More simply, the ApprovalCondition just checks to see whether anyone has approved or rejected it. Once a condition is resolved, it can no longer be updated.

If an action passes a permission but is rejected by the condition set on the permission, it is as if they were rejected by the permission itself. The action may pass through some other route, but not via that permission.

Conditions are themselves permissioned models. When setting an Approval Condition on a permission, the community may specify that only those with role "mods" can approve. When an action runs into a condition, an ApprovalCondition model is created corresponding to that action, along with a permission to approve which only those with role "mods" can take.

3.2.2 How to Add a Template

Concord's permissions system gives users very fine-grained control over permissions in thier communities, but it can be very difficult to build a desirable set of permissions up from scratch. Many, if not most, users will prefer to work from existing, familiar systems. Even users who are very comfortable adding and editing permissions, conditions, roles, etc will still benefit from having a library of templates to choose from.

To that end, Concord provides functionality for a "template library", that is a set of pre-existing structures that users can choose from and apply to their communities. The current template system is simplistic, and we hope to improve it in the future. In particular, we hope to add an interface so users can create their own templates via a web interface. For now, though, templates are added through code.

You can create new templates by adding them to any file called `template_library.py` that is in the top level of your Django app. The built-in templates from Concord are stored in `concord/actions/template_library.py`.

Templates are stored in a TemplateModel object and so to actually create new instances, you'll need to run `python manage.py update_templates`. If you've edited an existing template, you can recreate all of the template models by running `python manage.py update_templates --recreate`.

Example 1: Community Template

Let's add a template to the file. Our template will set up a leadership structure with a 'core team' that has a high level of trust. Because there's a high level of trust, the only condition set on owners will be a simple approval condition. All of our core team members will be governors as well.

To create a new template, start by importing `TemplateLibraryObject` from `concord/actions/template_library.py` and inheriting from that object:

```
class CommunityCoreTeamTemplate(TemplateLibraryObject):
    """Creates a template with core team as owners with an approval condition for␣
↪foundational actions.
    Core team members are also set as governor. Also incldues 'anyone can join'␣
↪permission."""
    name =
    description =
    scopes =
    supplied_fields =

    def get_action_list(self):

        client = self.get_client()

        return []
```

Let's start by filling out most of the class attributes. Name and description should be fairly straightforward. The name should be unique and do a decent job reflecting the template, though of course there are limits in how much nuance you can capture in a few words. With the description, do your best to explain what the template does in 'plain English'. Users considering applying the template will also see a description of the actions that the template takes, but the description can help them understand what they're reading.

We can also specify the scope. The scope determines where in the front-end the template is shown. Our template applies community-wide so it will appear in the governance section of the front-end. Community scopes are also made available when the user is creating their group.

We might fill out name, description and scope like this (we'll handled `supplied_fields` later):

```
class CommunityCoreTeamTemplate(TemplateLibraryObject):
    """Creates a template with core team as owners with an approval condition for␣
↪foundational actions.
    Core team members are also set as governor. Also incldues 'anyone can join'␣
↪permission."""
    name = "Core Team"
    scopes = ["community"]
    description = """This template is a good fit for small teams with high trust. It␣
↪creates a 'core team' role
                which is given both owner and governor powers. In order for core␣
↪team members to use their
                ownership authority one other core team member must approve. The␣
↪template also allows anyone
```

(continues on next page)

(continued from previous page)

```
                to join the community."""
    supplied_fields = {}

    def get_action_list(self):

        client = self.get_client()

        return []
```

Templates work by creating a series of "mock actions", returned by the method `get_action_list`. These are then used when a template is applied to generate real actions which are taken on the target community. To generate these actions,we use our client in "mock" mode, which is handled by the `get_client` method inherited from `TemplateLibraryObject`.

Our first step will be to add the 'core team' role:

```
def get_action_list(self):

    client = self.get_client()

    # Step 1: add 'core team'
    action_1 = client.Community.add_role(role_name="core team")
    action_1.target = "{{context.action.target}}"
```

When we make templates, we don't always know what the attributes of the action will be ahead of time. We can create placeholders in our mock actions via strings with the structure `"{{placeholder_content}}"`. The three types of placeholders reference the `context` (that is, the set of objects associated with the template, which always including the Apply Template action but often includes other objects), `supplied_fields` (data specified by the user when applying the template), and `previous` (a dictionary of the previous actions generated by the template and their results). Here, we're making the target of our action the target of the trigger action.

```
# Step 2: make initial people into 'core team'
action_2 = client.Community.add_people_to_role(
    role_name="core team", people_to_add="{{supplied_fields.initial_core_team_members}
↪}")
action_2.target = "{{context.action.target}}"
```

Next, we add people to the core team role. Whoever applies the template will be asked to specify these people. Notice that again, we need to specify that the target of this action should be the community that the ApplyTemplate action was taken on. Also note that we've changed the name of the mock_action being created, so we don't overwrite the previous one. At the end of this method, we'll be listing them all out in order, so we need to keep them separate.

The next step is to make the core team role into an ownership role:

```
# Step 3: make 'core team' role an ownership role
action_3 = client.Community.add_owner_role(owner_role="core team")
action_3.target = "{{context.action.target}}"
```

We also want to make it into a governorship role:

```
# Step 4: make 'core team' role an governorship role
action_4 = client.Community.add_governor_role(governor_role="core team")
action_4.target = "{{context.action.target}}"
```

Then we add a condition to the ownership role (though not the governing role):

```
# Step 5: add approval condition to ownership role
permission_data = [{"permission_type": Changes().Conditionals.Approve, "permission_
↪roles": ["core team"],
                    "permission_type": Changes().Conditionals.Reject, "permission_
↪roles": ["core team"]}]
action_5 = client.Community.add_leadership_condition(
    condition_type="approvalcondition", leadership_type="owner", permission_
↪data=permission_data)
action_5.target = "{{context.action.target}}"
```

Working with conditions takes a bit more configuration than other client calls. Here, we're controlling who has permission to update the condition by approving or rejecting the action that triggered it. Because conditions are themselves a type of template, we can configure them with placeholder strings as well, but we'll talk about this more in the next example. For now, the only placeholder we use is for the target, just like all the other actions in this template.

Finally, we'll let anyone join the group, but we don't give the members any special permissions by default:

```
# Step 6: add anyone can join permission
action_6 = client.PermissionResource.add_permission(
    permission_type=Changes().Communities.AddMembers, anyone=True, permission_
↪configuration={"self_only": True})
action_6.target = "{{context.action.target}}"
```

Finally, we collect all our actions and return them:

```
return [action_1, action_2, action_3, action_4, action_5, action_6]
```

One thing you may be wondering is why we had to specify the mock action target each time. After all, it was always `"{{context.action.target}}"`. And that's often (though not always) the case for template mock actions. To handle this, there's an optional attribute called `default_action_target`, set by default to `"{{context.action.target}}"`. If you don't set a mock action's target via `mock_action.target`, it will automatically be set to whatever the default is. If you overwrite the default to set it to None, you will have to make sure to set each mock action target individually, or an error will be raised.

Given that we've got `default_action_target` to handle setting all our actions' targets to the default, we can remove all those references from `get_action_list`.

The final step is to create our supplied fields.

```
supplied_fields = {
    "initial_core_team_members":
        ["ActorListField", {"label": "Who should the initial members of the core team␣
↪be?", "required": True}]
}
```

Supplied_fields is a dictionary with keys exactly the same as how they're referenced in our action placeholders. The value is a list, where the first item is always a string representing the type of field it is (ActorListField, RoleListField, CharField, IntegerField, or BooleanField), and the second is a dictionary with a variety of options. You should always provide a label for the field, which is what the user will see when applying your template. The other valid options vary depending on the type of field it is. For instance, integer fields accept a minimum and maximum argument, while others don't. All accept the 'required' argument, which we use here.

Putting everything together, we get the following object:

```
class CommunityCoreTeamTemplate(TemplateLibraryObject):
    """Creates a template with core team as owners with an approval condition for␣
↪foundational actions.
```

(continues on next page)

(continued from previous page)

```
    Core team members are also set as governor. Also incldues 'anyone can join'␣
↪permission."""
    name = "Core Team"
    scopes = ["community"]
    description = """This template is a good fit for small teams with high trust. It␣
↪creates a 'core team' role
                     which is given both owner and governor powers. In order for core␣
↪team members to use their
                     ownership authority one other core team member must approve. The␣
↪template also allows anyone
                     to join the community."""
    supplied_fields = {
        "initial_core_team_members":
            ["ActorListField", {"label": "Who should the initial members of the core␣
↪team be?", "required": True}]
    }

    def get_action_list(self):

        client = self.get_client()

        # Step 1: add 'core team'
        action_1 = client.Community.add_role(role_name="core team")

        # Step 2: make initial people into 'core team'
        action_2 = client.Community.add_people_to_role(
            role_name="core team", people_to_add="{{supplied_fields.initial_core_team_
↪members}}")

        # Step 3: make 'core team' role an ownership role
        action_3 = client.Community.add_owner_role(owner_role="core team")

        # Step 4: make 'core team' role an governorship role
        action_4 = client.Community.add_governor_role(governor_role="core team")

        # Step 5: add approval condition to ownership role
        permission_data = [{"permission_type": Changes().Conditionals.Approve,
↪"permission_roles": ["core team"],
                            "permission_type": Changes().Conditionals.Reject,
↪"permission_roles": ["core team"]}]
        action_5 = client.Community.add_leadership_condition(
            condition_type="approvalcondition", leadership_type="owner", permission_
↪data=permission_data)

        # Step 6: add anyone can join permission
        action_6 = client.PermissionResource.add_permission(
            permission_type=Changes().Communities.AddMembers, anyone=True, permission_
↪configuration={"self_only": True})

        return [action_1, action_2, action_3, action_4, action_5, action_6]
```

Here's what our new template looks like on the front end:

Example 2: Resource Template

Let's try another template, this one slightly more complex. This template will be for a resource object, simple list. You can learn more about simple lists *here* but the tl;dr is that they're ordered lists with configurable columns. The six types of state changes associated with simple lists are Add List, Edit List, Delete List, Add Row, Edit Row and Delete Row. Our template will set permissions on an existing list, so we'll ignore the Add List permission.

Let's start with our actions this time:

```
def get_action_list(self):

        client = self.get_client()

        # Step 1: give members permission to add row
        action_1 = client.PermissionResource.add_permission(
            permission_roles=['members'], permission_type=Changes().Resources.AddRow)

        # Step 2: give members permission to edit list
        action_2 = client.PermissionResource.add_permission(
            permission_roles=['members'], permission_type=Changes().Resources.
↪EditList)
```

The person who created the list likely wants to ability to review changes. This template will allow them to do so, by creating an approval condition and giving them permission to update the condition:

```
# Step 3: set approval condition on permission
permission_data = [{"permission_type": Changes().Conditionals.Approve,
                    "permission_actors": "{{nested:context.simplelist.creator||to_pk_
↪in_list}}"},
                   {"permission_type": Changes().Conditionals.Reject,
                    "permission_actors": "{{nested:context.simplelist.creator||to_pk_
↪in_list}}"}]
action_3 = client.PermissionResource.add_condition_to_permission(
    condition_type="approvalcondition", permission_data=permission_data
)
action_3.target = "{{previous.1.result}}"
```

Notice that we *are* explicitly setting the target of this action. That's because it's not the default target of `{{context.action.target}}` but instead the result of the previous action - that is, the permission we just created. To access this permission, we use the placeholder type `previous`. Previous is zero-indexed, so we're getting not the result of the first action, but the result of the second. We use `.result` to indicate that we want to get the result of the action. To get the action itself, we'd use `previous.1.action`.

When a condition is applied to a permission, it gets a lot of context from the permission object it's set on. We know the permission this condition is set on targets SimpleLst and so we know we'll have access to the target simplelist in the context. And we know that the attribute on simplelist we want to use is the creator.

The other thing to note is the `permission_actors`. We use the placeholder action to indiciate who should have permission to approve or reject - the creator of the simplelist - but we don't want to make that substitution when we apply the template. Instead, we want the placeholder to carry through to the condition itself. We do this by using the `nested:` syntax.

Finally, note the `||to_pk_in_list`. Sometimes we need to transform a piece of data to make it fit the field. Permission actors are stored as a list of pks, while accessing the attribute creator on simplelist will give us a User object. There are a small number of transformations available in the placeholder syntax, always called at the end of a placeholder string and indicated with `||`. `to_pk_in_list` assumes the value passed in has a pk, gets that pk, and puts it in a list, which is exactly what we want here.

Let's do the permissions & conditions for EditRow and DeleteRow. We can reuse the permission data for the two other conditions, because it's exactly the same for all three, but if we wanted to we could set different conditions.

```
# Step 4: give members permission to edit row
action_4 = client.PermissionResource.add_permission(
    permission_roles=['members'], permission_type=Changes().Resources.EditRow)

# Step 5: set approval condition on permission
action_5 = client.PermissionResource.add_condition_to_permission(
    condition_type="approvalcondition", permission_data=permission_data
)
action_5.target = "{{previous.3.result}}"

# Step 6: give members permission to edit row
action_6 = client.PermissionResource.add_permission(
    permission_roles=['members'], permission_type=Changes().Resources.DeleteRow)

# Step 7: set approval condition on permission
action_7 = client.PermissionResource.add_condition_to_permission(
    condition_type="approvalcondition", permission_data=permission_data
)
action_7.target = "{{previous.5.result}}"
```

And that's it. Here's the full Template object:

```
class SimpleListLimitedMemberTemplate(TemplateLibraryObject):
    """Creates permissions on a simplelist where members can add rows to the list␣
↪without condition but can only
    edit or delete rows, or edit the list itself, if the creator approves. Only the␣
↪creator can delete the list."""
    name = "Limited Member Permissions"
    scopes = ["simplelist"]
    supplied_fields = {}
    description = """This template allows members to add rows to the list without␣
↪needing approval. To edit or                        delete rows, or to edit the list␣
↪itself, the creator must approve. Only the creator may
                     delete the list."""

    def get_action_list(self):

        client = self.get_client()

        # Step 1: give members permission to add row
        action_1 = client.PermissionResource.add_permission(
            permission_roles=['members'], permission_type=Changes().Resources.AddRow)

        # Step 2: give members permission to edit list
        action_2 = client.PermissionResource.add_permission(
            permission_roles=['members'], permission_type=Changes().Resources.
↪EditList)

        # Step 3: set approval condition on permission
        permission_data = [{"permission_type": Changes().Conditionals.Approve,
                            "permission_actors": "{{nested:context.simplelist.
↪creator||to_pk_in_list}}"},
                           {"permission_type": Changes().Conditionals.Reject,
                            "permission_actors": "{{nested:context.simplelist.
↪creator||to_pk_in_list}}"}]
```

(continues on next page)

(continued from previous page)

```
        action_3 = client.PermissionResource.add_condition_to_permission(
            condition_type="approvalcondition", permission_data=permission_data
        )
        action_3.target = "{{previous.1.result}}"

        # Step 4: give members permission to edit row
        action_4 = client.PermissionResource.add_permission(
            permission_roles=['members'], permission_type=Changes().Resources.EditRow)

        # Step 5: set approval condition on permission
        action_5 = client.PermissionResource.add_condition_to_permission(
            condition_type="approvalcondition", permission_data=permission_data
        )
        action_5.target = "{{previous.3.result}}"

        # Step 6: give members permission to edit row
        action_6 = client.PermissionResource.add_permission(
            permission_roles=['members'], permission_type=Changes().Resources.
↪DeleteRow)

        # Step 7: set approval condition on permission
        action_7 = client.PermissionResource.add_condition_to_permission(
            condition_type="approvalcondition", permission_data=permission_data
        )
        action_7.target = "{{previous.5.result}}"

        return [action_1, action_2, action_3, action_4, action_5, action_6, action_7]
```

Note that we're not taking any user input here, so we have no supplied fields. (If we wanted to make a custom role the moderator of member actions on the list, we'd have to have a supplied field to determine what role to use.)

The scope of a template targetting a specific resource type should be the name of the model exactly in lower case.

Here's what our new template looks like on the front end. Note that the template doesn't appear as an option when creating the group, only when editing our list permissions:

3.2.3 How to Add a Condition

Conditions are placed on permissions and come into effect when a user wants to take an action. If a person has permission to take the action, either through a specific permission being set or because they're an owner or governor, and a condition is set, they must also pass the terms of that condition.

The output of a condition is always one of three things: "approved", "rejected" and "waiting". Additional data can be created by the condition, but that output is what the permissions system actually uses.

Right now, the only kind of condition that exists is a *decision* condition. Decision conditions require a decision - that is, at least one more action - by members of the community. The simplest decision condition is the Approval Condition, which requires a person to approve. Eventually, we will be adding automatic conditions, like "this person must have been in the group for two weeks" or "this person does not have the flag 'suspended'", and compound conditions, which allow us to specify a combination of conditions to be satisfied.

But that's getting ahead of ourselves. For now, let's stick to what we have.

Step 1: Planning Out Your Condition

It's always helpful to take some time to think through what you want the condition to do before you start coding.

Our condition is going to allow people to decide things by consensus. The options for people participating in the consensus condition will be "support", "support with reservations", "block" and "stand aside".

We'll also have two modes that the person setting the condition can choose from, "strict" and "loose". With strict consensus, all participants in the consensus process must actively choose one of the four options listed above for the condition to be resolved. With loose consensus, the only requirement is that no one block. But in that case, how do we know when a loose consensus process is finished? For that matter, how do we know if a "strict" consensus process is finished - what if, even though everyone's given an initial response, people want to keep discussing?

We'll have a special "resolve" action that can be taken on our consensus. When the resolve action is taken, no more responses can be added. Instead, we figure out the result with whatever responses are there at the moment of resolution.

If you've got a devious brain, you may have noticed some potential for abuse: what if someone prompts a loose consensus action and then immediate resolves it? No one will have blocked because no one will have had a chance to see it! So let's also add a minimum duration, before which the condition cannot be resolved.

Finally, let's require that in order for any consensus item to pass, at least one person actively supports it, either with reservations or without - we can't have all "stand asides" (or "no responses" and "stand asides", for loose consensus). m_duration field Step 2: Creating A Condition Model ^^

The heart of a condition is the Condition Model, where all the Condition's logic lives. New conditions are added in `concord.conditionals.models.py` and all condition model objects must descend from the `ConditionModel` defined in that module.

We'll start by specifying the fields we'll need for the condition to function:

```
class ConsensusCondition(ConditionModel):

    resolved = models.BooleanField(default=False)

    is_strict = models.BooleanField(default=False)
    responses = models.CharField(max_length=500, default="[]")

    minimum_duration = models.IntegerField(default=48)
    discussion_starts = models.DateTimeField(default=timezone.now)

    response_choices = ["support", "support with reservations", "stand aside", "block
↪"]
```

The `resolved` field stores whether or not the resolved action has been taken. `is_strict` stores the mode a particular consensus process is in. The `responses` field stores the responses of individual participants. The `minimum_duration` field stores the minimum length of the decision-making process while the `discussion_starts` stores when the discussion started, so the 'okay' point can be calculated from it using the duration. The `response_choices` field is not a model field, and technically could be specified elsewhere, but we'll keep it here where it's easy to find.

This seems good enough for now - we can always come back to add more fields later.

ConditionModel has a variety of abstract methods on it that *must* be implemented by any condition model descending from it. Let's start with `condition_status`. This is the method which returns "approved", "rejected" or "waiting".

```
def condition_status(self):
    """This method returns one of status 'approved', 'rejected', or 'waiting', after␣
↪checking the condition
    for its unqiue status logic."""
```

`condition_status` should never take in any parameters. The status should be inferrable from information contained on the condition itself. Let's try to think through the logic of our condition status:

- if the condition is not resolved, we should return "waiting"
- if the condition is resolved, and the mode is strict, and people haven't participated, we should "reject"
- if the condition is resolved, and the mode is strict, and everyone's participated, and there's a block or no supporters, we should "reject"
- if the condition is resolved, and the mode is strict, and everyone has participated, and there's no block and some supporters, we should "approve"
- if the condition is resolved, and the mode is loose, and there's a block or no supporters, we should "reject"
- if the condition is resolved, and the mode is loose, and there's no blocks and some supporters, we should "approve"

We can instantiate this logic in our condition status method:

```
def condition_status(self):

    if not self.resolved:
        return "waiting"
    if self.is_strict:
        if self.full_participation():
            if self.has_blocks() or not self.has_support():
                return "rejected"
            return "approved"
        return "rejected"
    if self.has_blocks() or not self.has_support():
        return "rejected"
    return "approved"
```

When building this method we want to take special care that we're always returning one of those three terms. The most common error here is to accidentally return None by making a mistake with the code. Here, though, we can see that there's no way to go through this code without returning one of our strings - every if/else ends with a return statement.

Let's actually split up this method into two, so we can, separately, check what the current result is:

```
def condition_status(self):

    if not self.resolved:
        return "waiting"
    return self.current_result()

def current_result(self):
    if self.is_strict:
        if self.full_participation():
            if self.has_blocks() or not self.has_support():
                return "rejected"
            return "approved"
        rreturn "rejected"
    if self.has_blocks() or not self.has_support():
        return "rejected"
    return "approved"
```

It's the exact same logic, just separated into two methods. In addition to relying on our existing fields (`resolved` and `is_strict`), this method relies on three helper methods, `full_participation`, `has_blocks` and `has_support`. Let's fill those out.

To work on these methods, we need a better sense of what that `responses` field looks like. Let's imagine it's a dictionary with participants' unique IDs as keys and their response as a value. If the value is null, that means they haven't responded yet.

Let's start by creating a helper method which generates this dictionary given a list of users, which will be supplied when creating the condition.

```
def create_response_dictionary(self, participant_pk_list):
    response_dict = {pk: "no response for pk in participant_pk_list}
    self.responses = json.dumps(response_dict)
```

Because we're storing the data as JSON, we'll need another helper method just to access the data:

```
def get_responses(self):
    return json.loads(self.responses)
```

Now we're ready to build our `full_participation` and `has_blocks` methods:

```
def full_participation(self):
    for user, response in self.get_responses().items():
        if response == "no response":
            return False
    return True

def has_blocks(self):
    for user, response in self.get_responses().items():
        if response == "block":
            return True
    return False

def has_support(self):
    for user, response in self.get_responses().items():
        if response in ["support", "support with reservations"]:
            return True
    return False
```

Let's tackle that `resolve` field next. We need a helper method that determines whether the condition can be resolved. We'll use this later to determine whether a person's "resolve" action is valid.

```
def time_until_duration_passed(self):
    seconds_passed = (timezone.now() - self.discussion_starts).total_seconds()
    hours_passed = seconds_passed / 360
    return self.minimum_duration - hours_passed

def ready_to_resolve(self):
    if self.time_until_duration_passed() <= 0:
        return True
    return False
```

We moved the `time_until_duration_passed` calculations into a separate method because we will likely want to access it on the front end, to show users how much time remains until the condition can be resolved.

We will be coming back and adding more to our model, but for now this is enough to be getting on with.

Step 3: Writing State Changes for Your Condition

The next step is to write state changes which control how the condition can be updated. All state changes should be placed in a file named `state_changes.py` and should descend from `BaseStateChange` which can be imported from `concord.actions.state_changes`.

Let's create a stub for our first state change, which will control how people add responses to the consensus condition:

```
class RespondStateChange(BaseStateChange):
    """State change for responding to a consensus condition"""
    description = ""
    section = ""
    verb_name = ""
    input_fields = []

    def __init__(self, response):
        ...

    @classmethod
    def get_allowable_targets(cls):
        ...

    def description_present_tense(self):
        ...

    def description_past_tense(self):
        ...

    def validate(self, actor, target):
        ...

    def implement(self, actor, target):
        ...
```

The set of attributes above are there to allow the state change to be used and displayed in various ways by the system, and can occasionally seem a little redundant. `description` is a short, simple description of what the state change does, in this case "Respond" will do just fine. `preposition` helps us correctly use the state change in an English sentence. The default preposition if none is specified is "to" which works for this state change, so we can remove that attribute. Next is `section` - this helps the front end group permission options when offering them to the user. We'll put it as "Consensus". `verb_name` is an all-lower-case term again used for providing human-readable descriptions of what's happening, so "respond" works here.

Finally, `input_fields`, which is a bit more complex. It helps us provide metadata for the parameters supplied to init, in this case "response". To do this, we import InputField, which is just a named tuple with four fields: name, type, required, and validate. Name should exactly correspond to the input parameter's name; type should be one of a dozen or so options for field types, including standard Django field types like BooleanField and CharField as well as Concord-specific field like ActorListField or RoleListFild; required indicates whether the field is required; and validate indicates whether the field should be checked when the change object is being validated.

Putting it all together, we fill out the attributes & init like this:

```
class RespondStateChange(BaseStateChange):
    """State change for responding to a consensus condition"""
    description = "Respond"
    preposition = ""
    section = "Consensus"
    verb_name = "respond"
```

(continues on next page)

(continued from previous page)

```
    input_fields = [InputField(name="response", type="CharField", required=True,␣
↪validate=False)]

    def __init__(self, response):
        self.response = response
```

The description_past_tense and description_present_tense are two additional methods helping us display English langauge descriptions of the actions. They can reference data supplied in __init__ so can be more precise. get_allowable_targets lists the permissioned models that the state change can be applied to. In this case, the only valid option is the ConsensusCondition.

```
@classmethod
def get_allowable_targets(cls):
    return [ConsensusCondition]

def description_present_tense(self):
    return f"respond with {self.response}"

def description_past_tense(self):
    return f"responded with {self.response}"
```

Next let's fill out our validation method. We start by calling the super() method which checks that the target of the action is one of the allowable targets, and also validates any input_fields that have validate=True. Then we provide validation specific to this action:

```
def validate(self, actor, target):
    """Checks that the actor is a participant."""
    if not super().validate(actor=actor, target=target):
        return False

    if self.response not in target.response_choices:
        self.set_validation_error(
            f"Response must be one of {', '.join(target.response_choices)}, not {self.
↪response}")
        return False

    return True
```

Finally, we tell the state change how to actually implement the action in the database:

```
def implement(self, actor, target):
    target.add_response(actor, self.response)
    target.save()
    return self.response
```

Again, we need a helper method:

```
def add_response(self, actor, new_response):
    responses = self.get_responses()
    for user, response in responses.items():
        if user == actor.pk:
            responses[user] = new_response
    self.responses = json.dumps(responses)
```

Note that we never save the condition model from the condition model. In fact, we can't - the system will raise an error if we try. Instead, we always save from the implement method of a state change.

Putting it all together, our state change looks like this:

```
class RespondStateChange(BaseStateChange):
    """State change for responding to a consensus condition"""
    description = "Respond"
    preposition = ""
    section = "Consensus"
    verb_name = "respond"
    input_fields = [InputField(name="response", type="CharField", required=True,␣
↪validate=False)]

    def __init__(self, response):
        self.response = response

    @classmethod
    def get_allowable_targets(cls):
        return [ConsensusCondition]

    def description_present_tense(self):
        return f"respond with {self.response}"

    def description_past_tense(self):
        return f"responded with {self.response}"

    def validate(self, actor, target):
        """Checks that the actor is a participant."""
        if not super().validate(actor=actor, target=target):
            return False

        if self.response not in target.response_choices:
            self.set_validation_error(
                f"Response must be one of {', '.join(target.response_choices)}, not
↪{self.response}")
            return False

        return True

    def implement(self, actor, target):
        target.add_response(actor, self.response)
        target.save()
        return self.response
```

The other change a user will want to make to a consensus condition is to resolve it. That state change will look like this:

```
class ResolveConsensusStateChange(BaseStateChange):
    """State change for resolving a consensus condition."""
    description = "Resolve"
    preposition = ""
    section = "Consensus"
    verb_name = "resolve"

    @classmethod
    def get_allowable_targets(cls):
        return [ConsensusCondition]

    def description_present_tense(self):
        return f"resolve"
```

(continues on next page)

(continued from previous page)

```
    def description_past_tense(self):
        return f"resolved"

    def validate(self, actor, target):
        """Checks that the actor is a participant."""
        if not super().validate(actor=actor, target=target):
            return False

        if not target.ready_to_resovlve():
            self.set_validation_error("The minimum duration of discussion has not yet passed.")
            return False

        return True

    def implement(self, actor, target):
        target.resolved = True
        target.save()
        return target
```

Step 4: Condition Creation

Some condition types don't require any special data on creation, but in our case, we want to specify a set of participants in the discussion. (We might also want to give people the ability to add new participants later, in which case we'd need a third state change - AddParticipant - but we'll leave that as an exercise for the reader.)

Instances of Conditions are created by code in `SetConditionOnActionStateChange` in the `conditionals.state_changes.py`. That code calls a method `initialize_condition` on that condition model that we can customize for our model. That method takes in parameters `target`, `condition_data`, and `permission_data`.

What we're looking for here is `permission_data`, which determines who can take action on our consensus condition. We want to find the permission associated with the RespondConsensusStateChange, and use the actors and roles specified there to populate our list of participants.

The condition may also be set not on a specific permission but on a leadership type. The leadership type will be one of two options: owner or governor. We should only ever be grabbing our participants from the permission *or* owners *or* governors.

As we create our participants we store them as a set (a data type with no duplicates) so each participant has one response, even if they qualify through multiple roles.

```
def initialize_condition(self, target, condition_data, permission_data, leadership_type):
    """Called when creating the condition, and passed condition_data and permission_data."""

    client = Client(target=target.target.get_owner())
    participants = set([])

    for permission in permission_data:
        if permission["permission_type"] == Changes().Conditionals.RespondConsensus:
            if permission["permission_roles"]:
                for role in permission["permission_roles"]:
                    for user in client.Community.get_users_given_role(role_name=role):
```

(continues on next page)

(continued from previous page)

```
                        participants.add(user)
            if permission["permission_actors"]:
                for actor in permission["permission_actors"]:
                    participants.add(int(actor))

    if leadership_type == "owner":
        for action in client.get_users_with_ownership_privileges():
            participants.add(int(actor))

    if leadership_type == "governor":
        for action in client.get_users_with_governorship_privileges():
            participants.add(int(actor))

    self.create_response_dictionary(participant_pk_list=list(participants))
```

Step 5: Interacting Via Client & Views

We need to create mechanisms for interacting with our consensus condition. We'll start by creating a client for people to call:

```
class ConsensusConditionClient(BaseClient):
    """The target of the ConsensusConditionClient must always be a ConsensusCondition␣
↪instance."""

    # Read only

    def get_current_results(self) -> Dict:
        """Gets current results of vote condition."""
        return self.target.get_responses()

    # State changes

    def respond(self, *, response: str) -> Tuple[int, Any]:
        """Add response to consensus condition."""
        change = sc.RespondStateChange(response=response)
        return self.create_and_take_action(change)

    def resolve(self) -> Tuple[int, Any]:
        """Resolve consensus condition."""
        change = sc.ResolveConsensusStateChange()
        return self.create_and_take_action(change)
```

We'll also add some views that call the client, so that we can interact with our condition via an API:

```
@login_required
def update_consensus_condition(request):

    request_data = json.loads(request.body.decode('utf-8'))
    condition_pk = request_data.get("condition_pk", None)
    action_to_take = request_data.get("action_to_take", None)
    response = request_data.get("response", None)

    consensusClient = Client(actor=request.user).Conditional.\
        get_condition_as_client(condition_type="ConsensusCondition", pk=condition_pk)
```

(continues on next page)

(continued from previous page)

```
    if action_to_take == "respond":
        action, result = consensusClient.respond(response=response)
    elif action_to_take == "resolve":
        action, result = consensusClient.resolve()

    return JsonResponse(get_action_dict(action))
```

And of course we need to add a reference in urls.py so it actually works:

```
path('api/update_consensus_condition/', views.update_consensus_condition, name=
↪'update_consensus_condition'),
```

Finally, we're going to add a signal for our condition, so that when it updates, we check and see if the action it's set on now passes:

```
for conditionModel in [ApprovalCondition, VoteCondition, ConsensusCondition]:
    post_save.connect(retry_action, sender=conditionModel)
```

Step 6: Default Front-End Implementation

The next thing we want to do is build a front-end implementation of this condition so it can be used on actual websites. Projects may end up override some or all of this implementation, but we want to give them a plug-and-play solution to use by default, and the process of doing this will also explain a few additional fields and methods we'll be adding to our Consensus Condition model.

The default system is built using the Vue framework. Most condition interfaces are a single Vue component on a single page, which are including in the detail view for the action that triggers the condition. That view has a place for discussion, which is where we assume the discussion necessary to reach consensus will take place.

So we don't need to build an interface for discussion, just a way for users to see the current status of the condition and make any changes they want to. We'll start by building the html part of our component:

```
<script type="text/x-template" id="consensus_condition_template">

    <span>

        <!-- Information about the discussion. -->

        <h5 class="my-2">Discussion Status</h5>

        <span v-if="is_resolved">The condition was resolved with resolution [[␣
↪condition_resolution_status]].
            <span v-if="response_selected">Your response was <b>[[response_selected]]
↪</b>.</span>
        </span>
        <span v-else>
            <span v-if="can_be_resolved">The minimum duration of [[ minimum_duration␣
↪]] has passed. If the discussion
                was resolved right now, the result would be: [[ current_result ]].
                <b-button v-if="can_resolve" class="btn-sm" variant="outline-secondary
↪" @click="resolve_condition()">
                    Resolve this discussion?</b-button>
            </span>
            <span v-else>The discussion cannot be resolved until the minimum duration␣
↪of [[ minimum_duration]] has passed.
```

(continues on next page)

(continued from previous page)

```
                This will happen in [[ time_remaining ]].
            </span>
        </span>

        <b-container class="bv-example-row border border-info my-2 p-2" id="consensus_
↪responses">
            <b-row><b-col class="text-center my-2">Current Responses</b-col></b-row>
            <b-row class="font-weight-bold">
                <b-col>Support</b-col><b-col>Support With Reservations</b-col><b-col>
↪Stand Aside</b-col><b-col>Block</b-col>
                <b-col>No Response</b-col>
            </b-row>
            <b-row>
                <b-col>[[get_names(response_data.support)]]</b-col>
                <b-col>[[get_names(response_data.support_with_reservations)]]</b-col>
                <b-col>[[get_names(response_data.stand_aside)]]</b-col>
                <b-col>[[get_names(response_data.block)]]</b-col>
                <b-col>[[get_names(response_data.no_response)]]</b-col>
            </b-row>
        </b-container>

        <!-- Interface for changes -->

        <div v-if="!is_resolved" class="my-3">
            <span v-if="can_respond">
                <b-form-group label="Your Response">
                    <b-form-radio-group id="user_response_radio_buttons" v-model=
↪"response_selected" :options="response_options"
                        button-variant="outline-info" buttons name="user_response_
↪radio_buttons"></b-form-radio-group>
                </b-form-group>
                <b-button class="btn-sm" @click="submit_response()">Submit</b-button>
            </span>
            <span v-else>You are not a participant in this consensus discussion.</
↪span>
        </div>

        <span v-if="error_message" class="text-danger">[[ error_message ]]</span>

    </span>
</script>
```

There's a bunch of different logic here, and we're using a lot of variables that we still need to define on our component. But for now, notice how the template falls into three main sections:

The top-most section provides information about the overall status of the resolution. We show the user different things based on whether the condition is resolved, able to be resolved, or not able to be resolved. If it can be resolved and the user has permission to resolve it, we give them the option to do so, along with the helpful information of what the result will be if the condition resolves right now.

The next section displays the list of current responses. Our responses are stored as a list of pks, so we call the `get_names` method to look up their username and turn them into a comma-separated list.

Finally, in the bottom section, if the user has permission to respond they are given the option to do so. Their current response (which defaults to 'no response' is pre-selected for them).

Now let's go ahead and make our component. We'll start from the template that all components have, with only the name of the component changed:

```
consensusConditionComponent = Vue.component('consensus-condition-ui', {
    delimiters: ['[[', ']]'],
    template: '#consensus_condition_template',
    props: ['condition_type', 'condition_pk', 'action_details'],
    store,
    data: function() {
        return {
            error_message: null,
            permission_details: null,
            condition_details: null,
        }
    },
    computed: {
    },
    created () {
        this.get_conditional_data()
    },
    methods: {
        ...Vuex.mapActions(['addOrUpdateAction']),
        get_axios() {
            axios.defaults.xsrfCookieName = 'csrftoken';
            axios.defaults.xsrfHeaderName = 'X-CSRFTOKEN';
            axios.defaults.headers = { "headers": { 'Content-Type': "application/json
↪" } }
            return axios
        },
        get_conditional_data() {
            axios = this.get_axios()
            url = "{% url 'get_conditional_data' %}"
            params = { condition_pk: this.condition_pk, condition_type: this.
↪condition_type }
            return axios.post(url, params).then(response => {
                this.permission_details = response.data.permission_details
                this.condition_details = response.data.condition_details
                for (field in this.condition_details.fields) {
                    name = this.condition_details.fields[field]["field_name"]
                    value = this.condition_details.fields[field]["field_value"]
                    Vue.set(this, name, value)
                }
                this.set_user_response()
            }).catch(error => {  console.log(error)  })
        },
        update_action(new_action_pk) {
            // update action this was a condition on
            this.addOrUpdateAction({ action_pk: this.action_details["action_pk"] })
            // also call vuex to record this as an action (need to do this for all␣
↪actions)
            this.addOrUpdateAction({ action_pk: new_action_pk })
        },

    }
})
```

Most of the above needs to be wrapped into a mixin so you don't need to define it yourself, sorry. But, quickly: `get_condition_data` gets data on the condition from the backend, and adds all the fields on the condition to the component for easy access. `update_action` handles making sure data about the actions we're taking (or influencing, through the condition) makes it back to the action vuex store so it can be referenced elsewhere on the site. The rest is configuration for axios, which we use to talk to the backend.

Now let's create the variables and methods for all the things we listed in our template. The data section is very straightforward, everything is populated elsewhere in the component so we can just set everything to null:

```
data: function() {
    return {
        error_message: null,
        permission_details: null,
        condition_details: null,
        // select data
        response_selected: null,
        // fields that will be automatically filled by getConditionData
        minimum_duration: null,
        time_remaining: null,
        can_be_resolved: null,
        responses: null,
        response_options: null,
        current_result: null
    }
},
```

The computed section has a bit more going on:

```
computed: {
    ...Vuex.mapGetters(['getUserName']),
    can_respond: function() {
        if (this.permission_details) {
            return this.permission_details["concord.conditionals.state_changes.
↪RespondConsensusStateChange"][0]
        }
    },
    can_resolve: function() {
        if (this.permission_details) {
            return this.permission_details["concord.conditionals.state_changes.
↪ResolveConsensusStateChange"][0]
        }
    },
    is_resolved: function() {
        if (this.condition_details) {
            if (["approved", "rejected", "implemented"].includes(this.condition_
↪details.status)) {
                return true
            } else { return false }
        }
    },
    response_data: function() {
        response_dict = {}
        if (this.response_options) {
            this.response_options.forEach(response_option => response_dict[response_
↪option.replace(/\s/g, "_")] = [])
            for (user in this.responses) {
                response_dict[this.responses[user].replace(/\s/g, "_")].push(user)
            }
        }
        return response_dict
    },
    condition_resolution_status: function() { if (this.condition_details) { return␣
↪this.condition_details.status }}
},
```

`can_respond` and `can resolve` look up whether the user has the corresponding permissions associated with the condition. This information is automatically supplied by the back end, we just need to know which permission we're looking for. The status attribute on `condition_details` is also automatically supplied. The only complex thing happening here is in `response_data`, where we're reformatting from the dictionary of user pk keys and response values to make 'collections' of values for us to display. When we do this, we're turing "No Response" into no_response so we can access it in our template.

Let's move on to the methods. We've got two little helper methods here:

```
get_names(pk_list) {
    if (pk_list) {
        name_list = []
        pk_list.forEach(pk => name_list.push(this.getUserName(parseInt(pk))))
        return name_list.join(", ")
    } else { return "" }

},
set_user_response() {
    user_pk = parseInt("{{request.user.pk}}")
    for (user in this.responses) {
        if (user == user_pk) { this.response_selected = this.responses[user] }
    }
},
```

`get_names` is looking up user names using the `getUserName` method defined in the Governance Vuex include and imported in the computed section like so:

```
computed: {
    ...Vuex.mapGetters(['getUserName']),
```

`set_user_response` gets the logged in user and looks for their response in the response dictionary. We use this primarily to pre-select the right radio button.

The two final methods are the calls to the backend. Let's look at `submit_response` first:

```
submit_response() {
    if (!this.response_selected) { this.error_message = "Please select a response" }
    if (this.response_selected == this.user_response) { this.error_message = "Your␣
↪response has not changed"; return }
    url = "{% url 'update_consensus_condition' %}"
    params = { condition_pk: this.condition_pk, action_to_take: "respond", response:␣
↪this.response_selected }
    axios.post(url, params).then(response => {
        if (["implemented", "waiting"].indexOf(response.data.action_status) > -1) {
            this.update_action(response.data.action_pk)
            this.get_conditional_data().catch(error => { this.error_message = error })
        } else {
            this.error_message = response.data.action_log
        }
    }).catch(error => {  console.log("Error updating condition: ", error); this.error_
↪message = error })
},
```

We do a little bit of client-side validation, checking that a response has been selected and it's different from what their current response in the back end. Then we submit that data to our backend and do some more error handling on the response. Note that on success we update all the actions and refresh the condition data from the backend.

`resolve_condition` looks very similar:

```
resolve_condition() {
    url = "{% url 'update_consensus_condition' %}"
    params = { condition_pk: this.condition_pk, action_to_take: "resolve" }
    axios.post(url, params).then(response => {
        if (["implemented", "waiting"].indexOf(response.data.action_status) > -1) {
            this.update_action(response.data.action_pk)
            this.get_conditional_data().catch(error => { this.error_message = error })
        } else {
            this.error_message = response.data.action_log
        }
    }).catch(error => {  console.log("Error updating condition: ", error); this.error_
↪message = error })
}
```

The last thing we need to do is make sure our new component is hooked up. We need to add the file itself to the list of includes imported in `html_templates_to_include.html` with the line: `{% include 'groups/actions/consensus_condition_component.html' %}`

We also need to add a reference in the `action_detail` template:

```
<b-card border-variant="secondary" class="my-3" v-if="condition_type">

        This action has a condition on it.  To pass the condition, [[ condition_pass␣
↪]].

        <approve-condition-ui v-if="condition_type=='ApprovalCondition'" :condition_
↪pk=condition_pk
            :condition_type=condition_type :action_details=action></approve-condition-
↪ui>

        <vote-condition-ui v-if="condition_type=='VoteCondition'" :condition_
↪pk=condition_pk
            :condition_type=condition_type :action_details=action></vote-condition-ui>

        <consensus-condition-ui v-if="condition_type=='ConsensusCondition'"␣
↪:condition_pk=condition_pk
            :condition_type=condition_type :action_details=action></consensus-
↪condition-ui>

</b-card>
```

Finishing Up in the Back End

This is all the code we need on the front end, but if we try to run it like this it will break, because we still need to do some work to supply some of these fields from the back end for the front end to use. So let's go back to our `models.py` and add a few more methods to our condition.

```
def display_fields(self):
    """Gets condition fields in form dict format, for displaying in the condition␣
↪component."""
    return [
        {"field_name": "minimum_duration", "field_value": self.duration_display(),
↪"hidden": False},
        {"field_name": "time_remaining", "field_value": self.time_remaining_display(),
↪ "hidden": False},
        {"field_name": "responses", "field_value": self.get_responses(), "hidden":␣
↪False},
```

(continues on next page)

(continued from previous page)

```
        {"field_name": "response_options", "field_value": self.response_choices,
↪"hidden": False},
        {"field_name": "can_be_resolved", "field_value": self.ready_to_resolve(),
↪"hidden": False},
        {"field_name": "current_result", "field_value": self.current_result(), "hidden
↪": False}
    ]
```

The field names all correspond to things etiher used directly in our component template or used in one of the component methods. Whatever field_name you define here is how you'll access it in the component. There's a few new helper methods here, `duration_display` and `time_remaining_display`, in addition to the methods we've already defined (`get_responses`, `ready_to_resoleve` and `current_result`).

```
def time_remaining_display(self):
    time_remaining = self.time_until_duration_passed()
    units = utils.parse_duration_into_units(time_remaining)
    return utils.display_duration_units(**units)

def duration_display(self):
    units = utils.parse_duration_into_units(self.minimum_duration)
    return utils.display_duration_units(**units)
```

We've got two utility function, the first of which takes a duration of time in hours and turns it into a dictionary of weeks, days, hours and minutes, and the second of which displays that as a string. This makes it easy for us to pass human-readable durations to the front end.

Two other methods we need to add are `display_status` and `description_for_passing_condition`. Display status is used in our template, but `description_for_passing_condition` is additionally used when displaying conditions that have been set on permissions.

```
def display_status(self):
    """Gets 'plain English' display of status."""
    consensus_type = "strict" if self.is_strict else "loose"
    if self.resolved:
        return f"The discussion has ended with result {self.condition_status} under
↪{consensus_type} consensus"
    return f"The discussion is ongoing with {self.time_remaining_display()}. If the␣
↪discussion ended now, " + \
           f"the result would be: {self.current_result()}"

def description_for_passing_condition(self, fill_dict=None):
    """Gets plain English description of what must be done to pass the condition."""
    return utils.description_for_passing_consensus_condition(self, fill_dict)
```

The descriptions for passing conditions are stored in the utils file, since they can get lengthy, although this one isn't too bad:

```
def description_for_passing_consensus_condition(condition, fill_dict=None):
    """Generate a 'plain English' description for passing the consensus condtion."""

    participate_actors = fill_dict.get("participate_actors", []) if fill_dict else␣
↪None
    participate_roles = fill_dict.get("participate_roles", []) if fill_dict else None

    if not fill_dict or (not participate_roles and not participate_actors):
        consensus_type = "strict" if condition.is_strict else "loose"
```

(continues on next page)

(continued from previous page)

```
        return f"a group of people must agree to it through {consensus_type} consensus
↪"

    participate_str = roles_and_actors({"roles": participate_roles, "actors":␣
↪participate_actors})

    if condition.is_strict:
        return f"{participate_str} must agree to it with everyone participating and␣
↪no one blocking"
    else:
        return f"{participate_str} must agree to it with no one blocking"
```

There's one other place our condition shows up on the front end, besides the template we created and besides the display of existing conditions, and that's when a person is *creating* or *editing* a condition. We need to let the front end know what parts of the condition model are available to configure. To do that, we set a method called `get_configuration_fields`, which are used to populate the condition creation/editing form:

```
@classmethod
def configurable_fields(cls):
    """Gets fields on condition which may be configured by user."""
    return {
        "is_strict": {
            "display": "Use strict consensus mode? (Defaults to loose.)", "can_depend
↪": False,
            **cls.get_form_dict_for_field(cls._meta.get_field("is_strict"))
        },
        "minimum_duration": {
            "display": "What is the minimum amount of time for discussion?", "can_
↪depend": False,
            **cls.get_form_dict_for_field(cls._meta.get_field("minimum_duration"))
        },
        "participant_roles": {
            "display": "Roles who can participate in the discussion", "type":
↪"RoleListField",
            "can_depend": True, "required": False, "value": None, "field_name":
↪"participant_roles",
            "full_name": Changes().Conditionals.RespondConsensus
        },
        "participant_actors": {
            "display": "People who can participate in the discussion", "type":
↪"ActorListField",
            "can_depend": True, "required": False, "value": None, "field_name":
↪"participant_actors",
            "full_name": Changes().Conditionals.RespondConsensus
        },
        "resolver_roles": {
            "display": "Roles who can end discussion", "type": "RoleListField",
            "can_depend": True, "required": False, "value": None, "field_name":
↪"resolver_roles",
            "full_name": Changes().Conditionals.ResolveConsensus
        },
        "resolver_actors": {
            "display": "People who can end discussion", "type": "ActorListField",
↪"can_depend": True,
            "required": False, "value": None, "field_name": "resolver_actors",
            "full_name": Changes().Conditionals.ResolveConsensus
```

(continues on next page)

(continued from previous page)

```
        }
    }
```

And that's it!

When it's all done, the condition should work like this:

3.2.4 How to Add a Resource

"Resources" is our catch-all term for objects which descend from PermissionedModel, excluding a few core objects like permissions, conditions and communities. Comments are a resource, for instance, as are forums and posts.

In order to show you how to add your own resources, we'll walk you through how we added the List resource. The steps are:

1. Create a new Django model inheriting from PermissionedModel, with any fields you like. Make sure to implement any required methods.
2. Think through however you'd like people to interact with the resource, and then write state changes for each action type.
3. Create a client for your resource following the pattern of existing clients, adding methods for each of the state changes as well as methods for getting information about your resource.
4. Create the Django views (& linked urls) which call that method and return the results as Javascript.
5. Create one or more Vue components (and corresponding Vuex storage and Vue-router references) for users to interact with.

This sounds like a lot, but it's relatively straightforward. Let's dive in.

Create a New Model

We begin by creating a Django model for our new resource, Lists. This model can go in any `models.py` as long as it's an installed app, and Concord's system will automatically detect it.

Our lists will have names and descriptions, but we also want to store list item data. We'll do so in a `rows` field. We use Django's TextField for this as, by default, the TextField's maximum length is None, and we may want to fit a *lot* of rows in our table.

```
from concord.actions.models import PermissionedModel

class SimpleList(PermissionedModel):

    name = models.CharField(max_length=200)
    description = models.CharField(max_length=200)
    rows = models.TextField(default=dict)
```

We can't save a Python dictionary straight to the database, though. We need to serialize it and deserialize it using Json. We'll also create a few helper methods for adding, editing and removing rows. We'll give our add_row method an optional index argument. If an index is supplied, we'll add the row at that location, otherwise we'll just append it to the end. Conversely, the index will be required when editing and deleting rows.

```
import json
from concord.actions.models import PermissionedModel

class SimpleList(PermissionedModel):

    name = models.CharField(max_length=200)
    description = models.CharField(max_length=200)
    rows = models.TextField(default=list)

    def get_rows(self):
        return json.loads(self.rows)

    def add_row(self, row, index=None):
        rows = self.get_rows()
        if index or index == 0:
            rows.insert(index, row)
        else:
            rows.append(row)
        self.rows = json.dumps(rows)

    def edit_row(self, row, index):
        rows = self.get_rows()
        rows[index] = row
        self.rows = json.dumps(rows)

    def delete_row(self, index):
        rows = self.get_rows()
        rows.pop(index)
        self.rows = json.dumps(rows)
```

Note that we're not saving our changes here. A PermissionedModel can only be saved by a call within a State Change's implement method.

If you look at the PermissionedModel definition in *concord.actions.models*, you'll see a few fields and a number of methods defined. These are inherited automatically by our model - we don't need to do anything to make them work, with one exception. We'll also inherit a few fields that way: the generic foreign key to the model's `owner`, and the booleans `foundational_permission_enabled` and `governing_permission_enabled`.

One method you may actually want to overwrite is `get_nested_objects`. Nested objects are anything our object might be placed 'within', and are referenced when looking up permissions that might control access to a model instance. For example, a Post is nested within a Forum and a Community, so *get_nested_objects* would include Forum and the community that owns it. This allows users to create a permission on the Forum applying to all posts within the Forum. Let's update our model to overwrite this method, and return the Community that owns the List.

```
import json
from concord.actions.models import PermissionedModel
from concord.communities.models import Community

class SimpleList(PermissionedModel):

    name = models.CharField(max_length=200)
    description = models.CharField(max_length=200)
    rows = models.TextField(list)

def get_rows(self):
    return json.loads(self.rows)
```

(continues on next page)

(continued from previous page)

```
def add_row(self, row):
    rows = self.get_rows()
    rows.append(row)
    self.rows = json.dumps(rows)

def delete_row(self, index):
    rows = self.get_rows()
    rows.pop(index)
    self.rows = json.dumps(rows)

def get_nested_objects(self):
    return [self.get_owner()]
```

This is enough detail for now. We'll come back later to add a bit more functionality to our SimpleList model.

Once you finish your model, you can make and run migrations.

Write State Changes

To create State Changes, think about the types of changes users might want to make to your model. For our SimpleList, the main thing they'll want to do is add and delete rows. They may also want to edit a row. They may want to edit the name or description of the list, or delete the list entirely. And of course to do any of this they'll probably want to create a list in the first place.

This gives us six types of actions:

- add list
- edit list
- delete list
- add row
- edit row
- delete row

This gives us six State Changes to write. Let's start with "add list". We create our first State Change in a file called `state_changes.py`, by inheriting from `BaseStateChange`. Again, as long as the file has the right name and is within an installed app, the system should find and use it automatically.

There are a number of methods we'll have to write for the State Change, shown here as stubs.

```
from concord.actions.state_changes import BaseStateChange

class AddListStateChange(BaseStateChange):
    description = ""
    input_fields = []

    def __init__(self):
        ...

    def get_settable_classes(self):
        ...

    def description_present_tense(self):
        ...
```

(continues on next page)

(continued from previous page)

```
    def description_past_tense(self):
        ...

    def validate(self, actor, target):
        ...

    def implement(self, actor, target):
        ...
```

Let's start with `__init__`. When we initialize our State Change, we need to pass into it any information about the change we want to make. In this case, that's the name and description of the list we want to create. We'll make the name required and the description optional:

```
def __init__(self, name, description=None):
    self.name = name
    self.description = description if description else ""
```

We also need to provide some metadata for our input fields. Primarily, this is used on the front end when users are specifying that some element of the condition depends on the action that triggers the condition. By providing metadata for the state change, we let the user know which elements of the action might be a good match for the condition.

To do this, we import InputField, which is just a named tuple with four fields: name, type, required, and validate. Name should exactly correspond to the input parameter's name; type should be one of a dozen or so options for field types, including standard Django field types like BooleanField and CharField as well as Concord-specific field like ActorListField or RoleListFild; required indicates whether the field is required; and validate indicates whether the field should be checked when the change object is being validated.

A change object's default validate method, defined on the abstract parent class, will look for any fields listed in `input_fields`. For each field, if validate is True, it will check whether the value supplied for it is valid, using Django's inbuilt `clean` method. In some cases, the data we're providing is not meant to be applied directly to the target (for instance, if we are adding a permission to a target, the supplied fields need to be validated against the Permission model, not the target model). In that case, we'd set `input_target` to the relevant model, but we don't need to do that here.

We have two input fields, name and description, both of which are CharFields. Only name is required, but both can be validated against the target.

```
class AddListStateChange(BaseStateChange):
    description = ""
    input_fields = [InputField(name="name", type="CharField", required=True,␣
↪validate=True),
                    InputField(name="description", type="CharField", required=False,␣
↪validate=True)]

    def __init__(self, name, description=None):
        self.name = name
        self.description = description if description else ""
```

The next thing we're going to do is validate our input in the `validate` method. This method returns True or False and is called when starting to process an Action. If this method returns False, the process is aborted and an error message returned to the user. Otherwise, processing continues.

Basic validation is taken care of automatically in the validation method defined on BaseStateChange. That method checks whether the target of the action is one of the State Change's allowable targets, and whether the data supplied for individual fields can actually be saved to their corresponding model fields. If you want to add additional validation, you'll make a super call within the method, like so:

```
def validate(self, actor, target):
    if not super().validate(actor=actor, target=target):
        return False
    # Additional validation goes here
```

Because we don't need to do any extra validation, we can ommit the implementation of `validate` entirely and just use the parent class's method. However we do want to make sure that validate method has the data it needs. So we need to add two additional things - one class method (`get_allowable_targets`) one attribute (`input_fields`).

`get_allowable_targets` indicates which models a change may be applied to. By default, all permissioned models in the entire system are allowable, but we want to override that to indicate only communities may be targets:

```
@classmethod
def get_allowable_targets(cls):
    return cls.get_community_models()
```

The last major method is `implement`. This is where we actually save changes to the database. It's called after the action is validated and after it's passed the permissions pipeline. Because this state change adds a list, we'll need to create an instance of our SimpleList model. We feed in the name and description we recieved in `__init__` along with specifying that the owner of the SimpleList should be the owner of the group or resource we're adding it to. Note also that we return the created object to the caller.

```
from concord.resources.models import SimpleList

def implement(self, actor, target):
    return SimpleList.objects.create(name=self.name, description=self.description,
        owner=target.get_owner(), creator=actor)
```

Finally, we have three simpler methods.

In `get_settable_classes`, we indicate which models a permission may be set on. This overlaps with, but is not identical to, allowable targets. Take for example an Edit List State Change. The target of "edit list" has to be limited to lists, but we might still want to set permissions at the level of a community, so we can say a user can edit all lists in a given community. By default `get_settable_classes` returns whatever we specified in `get_allowable_targets`. This works just fine for this particular State Change, so we don't need to implement the method.

The last two methods create human-readable descriptions of the state changes, in past and present tense. They pair with the `description` attribute set on the class itself to help our users understand what the state change does. The `description` attribute contains no instance-specific data, while `description_present_tense` and `description_past_tense` do incorporate it:

```
def description_present_tense(self):
    return f"add list with name {self.name}"

def description_past_tense(self):
    return f"added list with name {self.name}"
```

Putting this all together, we get the code for our Add List state change:

```
from concord.resources.models import SimpleList

class AddListStateChange(BaseStateChange):
    """State Change to create a list in a community (or other target)."""
    description = "Add list"
    input_fields = [InputField(name="name", type="CharField", required=True,␣
↪validate=True),
```

(continues on next page)

(continued from previous page)

```
                    InputField(name="description", type="CharField", required=False,␣
↪validate=True)]

    def __init__(self, name, description=None):
        self.name = name
        self.description = description if description else ""

    @classmethod
    def get_allowable_targets(cls):
        return cls.get_community_models()

    def description_present_tense(self):
        return f"add list with name {self.name}"

    def description_past_tense(self):
        return f"added list with name {self.name}"

    def implement(self, actor, target):
        return SimpleList.objects.create(name=self.name, description=self.description,
            owner=target.get_owner(), creator=actor)
```

Let's take a look at another state change, Edit List:

```
class EditListStateChange(BaseStateChange):
    """State Change to edit an existing list."""
    description = "Edit list"
input_fields = [InputField(name="name", type="CharField", required=False,␣
↪validate=True),
                InputField(name="description", type="CharField", required=False,␣
↪validate=True)]

    def __init__(self, name=None, description=None):
        self.name = name
        self.description = description if description else ""

    def get_allowable_targets(self):
        return [SimpleList]

    def get_settable_classes(self):
        return self.get_community_models() + [SimpleList]

    def description_present_tense(self):
        return f"edit list with new name {self.name} and new description {self.
↪description}"

    def description_past_tense(self):
        return f"edited list with new name {self.name} and new description {self.
↪description}"

    def validate(self, actor, target):
        if not super().validate(actor=actor, target=target):
            return False
        if not self.name and not self.description:
            self.set_validation_error(message="Must supply new name or description␣
↪when editing List.")
            return False
        return True
```

(continues on next page)

(continued from previous page)

```
    def implement(self, actor, target):
        target.name = self.name if self.name else target.name
        target.description = self.description if self.description else target.
↪description
        target.save()
        return target
```

Note that name is now optional too, as the state change allows us to edit the name or the description, or both. In `validate`, we'll add some custom validation to make sure the user is supplying at least one of those two fields.

Our allowable target is now only the SimpleList model, but our settable classes include any community model the list might belong to, as well as the list model itself.

Finally, the implement method looks up the List we want to change and edits the appropriate fields. If the user didn't specify a new value for a field, we just use the old value. Once again, we return the list instance from the implement method.

Let's do one more state change, Add Row.

```
class AddRowStateChange(BaseStateChange):
    """State Change to add a row to a list."""
    description = "Add row to list"
    input_fields = [InputField(name="row_content", type="CharField", required=True,␣
↪validate=False),
                    InputField(name="index", type="IntegerField", required=False,␣
↪validate=False)]

    def __init__(self, row_content, index=None):
        self.row_content = row_content
        self.index = index

    @classmethod
    def get_allowable_targets(cls):
        return [SimpleList]

    def get_settable_classes(self):
        return self.get_community_models() + [SimpleList]

    def description_present_tense(self):
        return f"add row with content {self.row_content}"

    def description_past_tense(self):
        return f"added row with content {self.row_content}"

    def validate(self, actor, target):
        if not super().validate(actor=actor, target=target):
            return False
        if not isinstance(self.row_content, str):
            self.set_validation_error(message="Row content must be a string.")
            return False
        if self.index and not isinstance(self.index, int):
            self.set_validation_error(message="Index must be an integer.")
            return False
        return True

    def implement(self, actor, target):
```

(continues on next page)

(continued from previous page)

```
        target.add_row(self.row_content, self.index)
        target.save()
        return target
```

Here we make use of the `add_row` method we wrote for our model. Note that we do not need to validate whether the index is valid as the `list.insert()` call we make in that method does not raise index errors. Instead, it interprets any negative integer indexes as "put this at the front of the list" and any too-high positive value indexes as "put this in the back of the list". That behavior works fine here, so we don't need to add any extra validation for it.

Go ahead and implement Delete List, Edit Row, and Delete Row yourself, and we'll be ready to move on to the next step.

Add a Client

We use clients to interact with state changes. This helps maintain a layer of abstraction, and makes it easier for users to utilize state changes without having to remember the name of each one. When creating our client,we inherit from BaseClient, which provides many of the basic and helper methods that clients use. All we need to do is create the methods specific to our model. Let's create the client with stubs again:

```
from concord.actions.client import BaseClient

class ListClient(BaseClient):
    """Client for interacting with Lists."""

    # Read methods

    def get_list(self, pk):
        ...

    def get_all_lists(self):
        ...

    def get_all_lists_given_owner(self, owner):
        ...

    # State changes

    def add_list(self, name, description=None):
        ...

    def edit_list(self, name=None, description=None):
        ...

    def delete_list(self):
        ...

    def add_row(self, row_content, index=None):
        ...

    def edit_row(self, row_content, index):
        ...

    def delete_row(self, index):
        ...
```

The read methods are straightforward - we merely use the Django models to get the appropriate data:

```
def get_list(self, pk):
    return SimpleList.objects.get(pk=pk)

def get_all_lists(self):
    return SimpleList.objects.all()

def get_all_lists_given_owner(self, owner):
    content_type = ContentType.objects.get_for_model(owner)
    return SimpleList.objects.filter(
        owner_content_type=content_type,owner_object_id=owner.id)
```

Client methods which handle state changes follow a very specific format. First, you want to instantiate a State Change object using the parameters passed in, and then you want to call BaseClient's method `create_and_take_action`. Typically, we import all state changes as `sc` for easy reference. This gives us a client that looks like this:

```
from django.contrib.contenttypes.models import ContentType
from concord.resources.models import SimpleList
from concord.resources import state_changes as sc
from concord.actions.client import BaseClient

class ListClient(BaseClient):
    """Client for interacting with Lists."""

    # Read methods

    def get_all_lists(self):
        return SimpleList.objects.all()

    def get_all_lists_given_owner(self, owner):
        content_type = ContentType.objects.get_for_model(owner)
        return SimpleList.objects.filter(
            owner_content_type=content_type,owner_object_id=owner.id)

    # State changes

    def add_list(self, name, description=None):
        change = sc.AddListStateChange(name=name, description=description)
        return self.create_and_take_action(change)

    def edit_list(self, name=None, description=None):
        change = sc.EditListStateChange(name=name, description=description)
        return self.create_and_take_action(change)

    def delete_list(self):
        change = sc.DeleteListStateChange()
        return self.create_and_take_action(change)

    def add_row(self, row_content, index=None):
        change = sc.AddRowStateChange(row_content=row_content, index=index)
        return self.create_and_take_action(change)

    def edit_row(self, row_content, index):
        change = sc.EditRowStateChange(row_content=row_content, index=index)
        return self.create_and_take_action(change)

    def delete_row(self, index):
        change = sc.DeleteRowStateChange(index=index)
```

(continues on next page)

(continued from previous page)

```
        return self.create_and_take_action(change)
```

Add Urls & Views

Our front end will not be directly interacting with our client, so the next step is to build urls & views that correspond to our methods. We can often create one url-view pair for each method. Our urls will look something like this.

```
# list views
path('api/<int:target>/get_lists/', views.get_lists, name='get_lists'),
path('api/<int:target>/add_list/', views.add_list, name='add_list'),
path('api/<int:target>/edit_list/', views.edit_list, name='edit_list'),
path('api/<int:target>/delete_list/', views.delete_list, name='delete_list'),
path('api/<int:target>/add_row/', views.add_row, name='add_row'),
path('api/<int:target>/edit_row/', views.edit_row, name='edit_row'),
path('api/<int:target>/delete_row/', views.delete_row, name='delete_row'),
```

Then, for each view listed in the urls, we need to create the method. Within the method, we'll call our client and return the response as JSON. Here are our stubs. Note the decorators which make it so only logged in users can access the methods, and reformats request data to be available as parameters.

```
@login_required
def get_lists(request, target):
    ...

@login_required
@reformat_input_data
def add_list(request, target, name, description=None):
    ...

@login_required
@reformat_input_data
def edit_list(request, target, list_pk, name=None, description=None):
    ...

@login_required
@reformat_input_data
def delete_list(request, target, list_pk):
    ...

@login_required
@reformat_input_data
def add_row(request, target, list_pk, row_content, index=None):
    ...

@login_required
@reformat_input_data
def edit_row(request, target, list_pk, row_content, index):
    ...

@login_required
@reformat_input_data
def delete_row(request, target, list_pk, index):
    ...
```

Let's fill out a couple of these stubs. We'll start with the get method. The target being passed in is the group in which a list might be contained, so we can use that target for our client (once we get the actual object).

```
@login_required
def get_lists(request, target):
    client = Client(actor=request.user)
    target = client.Community.get_community(community_pk=target)
    client.update_target_on_all(target=target)

    lists = client.List.get_all_lists_given_owner(owner=target)
    serialized_lists = serialize_listsfor_vue(lists)

    return JsonResponse({"lists": serialized_lists})
```

This method uses the helper method `serialize_lists_for_vue`, which puts the Python list object into a JSON format. This will be helpful any time we need to return list data. Here are the two helper methods:

```
def serialize_list_for_vue(simple_list):
    return {'pk': simple_list.pk, 'name': simple_list.name, 'description': simple_
↪list.description}


def serialize_lists_for_vue(simple_lists):
    serialized_lists = []
    for simple_list in simple_lists:
        serialized_lists.append(serialize_list_for_vue(simple_list))
    return serialized_lists
```

The next view is slightly more complex - add list. Again, we can re-use the target passed in as the target of our client, as we are adding lists to our community by default.

```
@login_required
@reformat_input_data
def add_list(request, target, name, description=None):

    client = Client(actor=request.user)
    target = client.Community.get_community(community_pk=target)
    client.update_target_on_all(target=target)

    action, result = client.List.add_list(name=name, description=description)

    action_dict = get_action_dict(action)
    if action.status == "implemented":
        action_dict["list_data"] = serialize_list_for_vue(result)
    return JsonResponse(action_dict)
```

Client calls that successfully change state always return two variables - the action that was created through this process and the result of the `action.change.implement` call, which is almost always the thing being created or edited.

`get_action_dict` is a helper method that serializes action data for vue to make use of. This is very important since our front end exposes the action history to the user. If the action's status is implemented and we have a new list, we serialize that list and pass it back to vue so it can update what the user sees.

Finally, let's look at edit list:

```
@login_required
@reformat_input_data
def edit_list(request, target, list_pk, name=None, description=None):

    client = Client(actor=request.user)
    target = client.List.get_list(pk=list_pk)
```

(continues on next page)

(continued from previous page)

```
    client.List.set_target(target=target)

    action, result = client.List.edit_list(name=name, description=description)

    action_dict = get_action_dict(action)
    if action.status == "implemented":
        action_dict["list_data"] = serialize_list_for_vue(result)
    return JsonResponse(action_dict)
```

Our edit list view is very similar to our add list view, but note that instead of using the community as a target we get the pk for the list that we're editing from our request data. We use the `get_list` helper method to get the list object and we set it as our target.

Go ahead and fill out the code for the other four views.

Hooking Up Vue

Creating your front-end in Vue can be a very complex process or a relatively simple one, depending on how users will interact with your model. For a simple use case like our SimpleList, we can divide the process into three steps:

1. Create a Vuex store corresponding to your object data which gets and sends data to the urls/views you just created. Then, hook it up to the application-wide store.
2. Create a Vue component or two corresponding to your model. This component should get and send data to the vuex store. Hook up the various components via Vue Router.
3. Add some final infrastructure to set up permissions and action history for your object.

Before we begin, let's create a templates folder for our front end. We'll call the folder `simplelists`.

Create a Vuex store

A vuex store is a Javascript module which handles the data for a component or components. Because our front end is currently a mix of Django templates and Vue/Vuex, we're going to stick our vuex store in a Javascript html block within a Django template. The naming structure for this store should be something like "datastructure + vuex + include" ('include' is a Django term for a piece of template that gets included in larger templates. So let's create a file *simplelist_vuex_include.html* in our simplelists folder and give it the following contents:

```
<script type="application/javascript">

const SimplelistVuexModule = {
    state: {
    },
    getters: {
    },
    mutations: {
    },
    actions: {
    }
}

</script>
```

You then want to go to wherever you're including your vuex stores (in the default system, that's `group_vuex_include.html`) and add following line:

`{% include 'groups/simplelists/simplelist_vuex_include.html' %}`.

You'll also need to scroll down to the Vuex Store and add a reference to our new component:

```
const store = new Vuex.Store({
    modules: {
        concord_actions: ActionsVuexModule,  // use 'concord_actions' so as not to␣
↪conflict with vuex actions
        forums: ForumsVuexModule,
        permissions: PermissionsVuexModule,
        governance: GovernanceVuexModule,
        comments: CommentVuexModule,
        templates: TemplateVuexModule,
        simplelists: SimplelistVuexModule
    },
```

Once that's set up, we'll set up our module's store. Vuex works by storing data in `state`. The only way to change state is through `mutations`. Getters are helper methods for accessing state without changing it, and actions are helper methods for accessing state while changing it (although you don't *have* to change state within a vuex action).

Let's start by defining our state. For now, it will be very simple - we're just going to store the list data which we serialized in our view. I like to put a comment in the state to remind me of the data structure.

```
state: {
    lists: []           // {'name': x, 'description': y, 'rows': ['str', 'str']}
},
```

We'll skip the getters for now and move down to the mutations. Mutations should be as atomic as possible, and are typically written in all caps. We'll end up creating approximately one mutation for each state change. We can use `ADD_OR_UPDATE_LIST` for both adding and editing lists, so we need five mutations.

```
ADD_OR_UPDATE_LIST (state, data) {
},
DELETE_LIST (state, data) {
},
ADD_ROW (state, data) {
},
EDIT_ROW (state, data) {
},
DELETE_ROW (state, data) {
}
```

When writing mutations, pay close attention to the types of changes Vue recognizes. Generally speaking, we use `Vue.set()` or `push` when adding data and `splice` when removing or inserting data. Check out the Vue documentation to learn more.

Let's go ahead and write one mutation:

```
ADD_OR_UPDATE_LIST (state, data) {
    for (index in state.lists) {
        if (state.lists[index].pk == data.list_data.pk) {
            state.lists.splice(index, 1, data.list_data)
            return
        }
    }
    state.lists.push(data.list_data)
}
```

The two parameters passed in, state and data, refer to the Vuex state we just defined and the data passed in from the

caller. In this case, we're passing in a serialized list as list_data, which includes the pk of the list. We iterate through all the lists currently in the store and, if we find one with our list's pk, we replace that item and return. If we don't find a match, we push our new list to the end of the lists array.

We do something similar for add row:

```
ADD_ROW (state, data) {
    list_to_edit = state.lists.find(list => list.pk == data.list_pk)
    if (data.index) {
        list_to_edit.rows.splice(data.index, 0, data.row_content)
    } else {
        list_to_edit.rows.push(data.row_content)
    }
},
```

Once you've added your mutations, it's time to write the Vuex actions that will utilize them. There are two types of Vuex actions in our project, both of which involve querying a Django view and then updating the store via a mutation if the query resolves successfully. Let's first write an action that gets all existing lists:

```
getLists({ commit, state, dispatch}, payload) {
    url = "{% url 'get_lists' target=object.pk %}"
    params = {}
    implementationCallback = (response) => {
        for (list in response.data.lists) {
            commit('ADD_OR_UPDATE_LIST', { list_data : response.data.lists[list] })
        }
    }
    return dispatch('getAPIcall', { url: url, params: params, implementationCallback:␣
↪implementationCallback})
},
```

We've wrapped most of the details of the API call into a method called `getAPIcall`. To use it in an Action, you call it while giving it a url, parameters, and a function to be called if the API call is successful. Here, we use Django's url template tag to get the appropriate url for the view we want to query, 'get lists'. We don't need to send any params, because the get_lists view doesn't require any - it just returns all the lists. Finally, in our implementation callback method we call the `ADD_OR_UPDATE_LIST` mutation for all of the lists returned to us.

In addition to the "getAPIcall" Action, other type of Vuex Action we use is an "actionAPIcall". It's pretty similar, but the key difference is that this API call, if successful, will create a Concord Action on the backend (not to be confused with a Vuex Action). We want to get the data about the action from our backend and display it to the user. That's what the `get_action_dict` calls in our views were helping us do. By using `actionAPIcall` instead, all of that should be handled for us:

```
addList({ commit, state, dispatch}, payload) {
    url = "{% url 'add_list' target=object.pk %}"
    params = { name: payload.name, description: payload.description }
    implementationCallback = (response) => {
        commit('ADD_OR_UPDATE_LIST', { list_data : response.data.list_data })
    }
    return dispatch('actionAPIcall', { url: url, params: params,␣
↪implementationCallback: implementationCallback})
},
```

Again, we feed url and parameters into our call, along with a set of mutations to be implemented if the call is successful.

The final thing to look at is the getters. This are simply helper methods that components can use to access state more easily. For instance, if we want to get a single list merely by passing in the list_pk, we can write the getter:

```
getters: {
    getListData: (state, getters) => (list_pk) => {
        return state.lists.find(list => list.pk == list_pk)
    }
},
```

Once we've finished writing our Vuex store, including mutation-action pairs corresponding to our view-url pairs (which correspond to our state-change-client pairs), we're finally ready to make our Vue componet.

Writing the Vue Component

There's actually four components we'll be making here: the component which lists all of our lists, the component which shows the details of a single list, and two form components - one for adding/editing list details, and one for adding/editing rows.

For each component, we'll take approximately these steps:

1. Create the file and the script structure, including the bare bones of the Vue component. Include it in other templates so it can be accessed.
2. Get the existing state from your store and write html to display it in your component. If the state needs to be fetched from the backend, get it in the `create` method.
3. In the methods section, create action calls to alter state like, for example, "delete list".
4. Add buttons which link to other components. Then wrap those buttons in a router-link and add the route to the router.
5. When the main functionality is complete, add in router-links to the item's permissions view and action history view.
6. Add in permissions checks and set up ability to hide things the user doesn't have permission for.

(Note: the docs below don't walk through those steps. We hope to come back and update these docs to do so, but for now we're just showing you all the code in each component at once.)

Let's start with the list of lists:

```
<script type="text/x-template" id="simplelist_list_component_template">

    <span>

        <h4 class="text-secondary pb-3">{{ object.name }}'s Lists

        <router-link :to="{ name: 'add-new-list'}">
            <b-button variant="outline-secondary"
                class="btn-sm ml-3" id="new_list_button">+ add new</b-button>
        </router-link>
        </h4>

        <router-link v-for="{ pk, name, description } in lists" v-bind:key=pk
                :to="{ name: 'list-detail', params: { list_id: pk } }">
        <b-card v-bind:key=pk class="bg-light text-info border-secondary mb-3">
            <b-card-title>[[ name ]]<span class="text-dark ml-2"><small>
                [[ rows.length ]] items</small></span></b-card-title>
            <p class="mb-1 text-secondary list-description"> [[ description ]] </p>
        </b-card>
        </router-link>
```

(continues on next page)

(continued from previous page)

```
        <span v-if="Object.keys(lists).length === 0">You do not have any lists yet.</
↪span>

    </span>

</script>


<script type="application/javascript">

    // Main vue instance for forums app
    var simplelistListComponent = Vue.component('simplelist-list-component', {
        delimiters: ['[[', ']]'],
        template: '#simplelist_list_component_template',
        store,
        data: function() {
            return {
            }
        },
        computed: {
            ...Vuex.mapState({
                lists: state => state.simplelists.lists,
                user_permissions: state => state.permissions.current_user_permissions
            }),
        },
        created () {
            this.getLists()
            this.checkPermissions({ permissions: { add_list: null }})
                .catch(error => {  this.error_message = error; console.log(error) })
        },
        methods: {
            ...Vuex.mapActions(['checkPermissions', 'getLists'])
        }
    })

</script>
```

There are a few things to note here. First, the overall format: all components consist of two sets of script tags: the first, with type "text/x-template" and a unique id, which contains the html for our component, and the second with type "application/javascript" which contains the component itself. The template attribute on our component is an exact match for the unique ID of the first script.

To access our Vuex data we import `store`. We also have to map any state, actions, or getters we defined on our store (we never map mutations, because mutations are only accessed via actions). For our list of lists, all we need to access are the `getLists` action and the lists state. Using those, we iterate through our existing lists and display their basic data.

We use vue router to link to other components. This allows us to create unique-urls for different views, which will allow our users to navigate to a particular part of the app by entering a url in the browser. Without this, there would be no way for users to link within the app. Here, we include links to the list component, which shows a single list in detail, as well as to the form for adding/editing lists. We'll talk about those in just a moment, and we'll talk about how to hook up all your vue-router paths near the end.

You can ignore the rest for now, including the `checkPermissions` action. That's the helper method which hides actions a user does not have permission to take, and we'll talk more about it later.

Let's take a look at our add/edit list component:

```
<script type="text/x-template" id="list_form_component_template">

    <b-modal id="list_modal" :title="title_string" size="md" :visible=true hide-
↪footer @hide="$router.go(-1)">

        <b-form-group id="name_form_group" label="List name:" label-for="list_name">
            <b-form-input id="list_name" name="list_name" v-model="name" required␣
↪placeholder="Please pick a name">
            </b-form-input>
        </b-form-group>

        <b-form-group id="description_form_group" label="List description:" label-for=
↪"list_description">
            <b-form-textarea id="list_description" name="list_description" v-model=
↪"description" placeholder="Add a description">
            </b-form-textarea>
        </b-form-group>

        <b-button v-if=!list_id variant="outline-secondary" class="btn-sm" id="add_
↪list_button"
            @click="add_list">submit</b-button>
        <b-button v-if=list_id variant="outline-secondary" class="btn-sm" id="edit_
↪list_save_button"
            @click="edit_list">submit</b-button>

        <error-component :message=error_message></error-component>

    </b-modal>

</script>

<script type="application/javascript">

    var listFormComponent = Vue.component('list-form-component', {
        delimiters: ['[[', ']]'],
        template: '#list_form_component_template',
        props: ['list_id'],
        store,
        data: function() {
            return {
                name: null,
                description: null,
                error_message: null
            }
        },
        created () {
            if (this.list_id && !this.lists_loaded) { this.getLists() }
        },
        computed: {
            ...Vuex.mapState({ lists: state => state.simplelists.lists }),
            ...Vuex.mapGetters(['getListData']),
            title_string: function() {
                if (this.list_id) {  return "Edit list '" + this.name + "'" }
                else { return "Add a new list" }
            },
            lists_loaded: function() {
                if (this.lists.length == 0) { return false }
```

(continues on next page)

(continued from previous page)

```
                else { this.populate_list_data(); return true }
            }
        },
        methods: {
            ...Vuex.mapActions(['getLists', 'addList', 'editList']),
            populate_list_data() {
                if (this.list_id) {
                    list = this.getListData(this.list_id)
                    this.name = list.name
                    this.description = list.description
                }
            },
            add_list() {
                this.addList({ name: this.name, description: this.description })
                .catch(error => {  console.log(error), this.error_message = error })
            },
            edit_list() {
                this.editList({ list_pk: parseInt(this.list_id), name: this.name,␣
↪description: this.description })
                .catch(error => {  this.error_message = error })
            }
        }
    })

</script>
```

You can see how similar it is in overall structure. The add/edit form is the component that uses our Addlist and EditList actions/mutations. Also note how, when the modal is hidden, we go back one route. This brings us back to the 'list of lists' page, if we've called it from there, or from the 'list detail' page, if we've called it from there.

Note that this component gets a prop, `list_id`. Ths is passed in via the router. If `list_id` is passed in, it checks whether or not the lists have been fetched from the backend and, if they have not, fetches them so we can supply the data for an existing list. This is helpful when someone is coming to this view directly via the URL, as opposed to having clicked the edit list button. When someone is coming from the edit list button the lists will already be loaded, so we don't have to fetch them again.

Next let's look at the list detail component, which is the most complex component we'll make. I'll show this in two parts, first the html section and then the javascript section (to get the appropriate highlighting):

```
<!-- Goes within xtemplate script -->

<span>

    <h3 class="mt-3">[[ list_name ]]</h3>
    <p>[[ list_description ]]</p>

    <router-link v-if="user_permissions.edit_list"
            :to="{ name: 'edit-list-info', params: { list_id: list_id } }">
        <b-button variant="outline-secondary" class="btn-sm" id="edit_list_button">
            edit list info</b-button>
    </router-link>

    <b-button v-if="user_permissions.delete_list" variant="outline-secondary" class=
↪"btn-sm"
        id="delete_list_button" @click="delete_list(list_id)">delete list</b-button>

    <router-link v-if="user_permissions.add_row"
```

(continues on next page)

(continued from previous page)

```
                :to="{ name: 'add-list-row', params: { list_id: list_id, mode: 'create
↪' } }">
            <b-button variant="outline-info" class="btn-sm" id="add_row_button">
                add a row</b-button>
    </router-link>


    <error-component :message="error_message"></error-component>

    <hr >

    <b-table striped hover :items="list_data" :fields="list_fields">

            <template v-slot:cell(change)="data">

                <router-link v-if="user_permissions.edit_row" :to="{ name: 'edit-list-
↪row',
                    params: { list_id: list_id, mode: 'edit', row_index: data.item.
↪index } }">
                    <b-button variant="outline-secondary" class="btn-sm">edit</b-
↪button>
                </router-link>

                <b-button v-if="user_permissions.delete_row" variant="outline-
↪secondary"
                    class="btn-sm" @click="delete_row(data.item)">
                delete</b-button>

            </template>

    </b-table>

    <span v-if="Object.keys(rows).length === 0">There are no items yet in this list.</
↪span>

</span>
```

```
// Goes within Javascript script

listComponent = Vue.component('list-component', {
    delimiters: ['[[', ']]'],
    template: '#simplelist_component_template',
    props: ['list_id'],
    store,
    mixins: [utilityMixin],
    data: function() {
            return {
                error_message: null,
                list_fields: [
                    { key: 'index', sortable: true },
                    { key: 'content', sortable: true },
                    'change'
                ]
            }
        },
    created (){
```

(continues on next page)

(continued from previous page)

```
        if (!this.lists_loaded) {  this.getLists()  }
        alt_target = "simplelist_" + this.list_id
        this.checkPermissions({
            permissions:
                { edit_list: {alt_target : alt_target},
                delete_list: {alt_target : alt_target},
                add_row: {alt_target : alt_target},
                edit_row: {alt_target : alt_target},
                delete_row: {alt_target : alt_target}}
        }).catch(error => {  this.error_message = error; console.log(error) })
    },
    computed: {
        ...Vuex.mapState({
            lists: state => state.simplelists.lists,
            user_permissions: state => state.permissions.current_user_permissions
        }),
        ...Vuex.mapGetters(['getListData', 'getUserName']),
        lists_loaded: function() { if (this.lists.length == 0) { return false } else
↪{ return true }},
        rows: function() {
            if (this.lists_loaded) { return this.getListData(this.list_id).rows }␣
↪else { return [] }},
        list_name: function() {
            if (this.lists_loaded) { return this.getListData(this.list_id).name }␣
↪else { return "" }},
        list_description: function() {
            if (this.lists_loaded) { return this.getListData(this.list_id).
↪description }
            else { return }},
        list_data: function() {
            list_data = []
            index = 0
            for (row in this.rows) {
                list_data.push({content: this.rows[row], index: index })
                index++
            }
            return list_data
        }
    },
    methods: {
        ...Vuex.mapActions(['checkPermissions', 'getLists', 'deleteList', 'deleteRow
↪']),
        display_date(date) { return Date(date) },
        delete_list(list_id) {
            this.deleteList({list_pk: list_id})
            .then(response => { this.$router.push({name: 'home'}) })
            .catch(error => {  this.error_message = error })
        },
        delete_row(item) {
            this.deleteRow({list_pk: this.list_id, index:item.index})
            .catch(error => {  this.error_message = error })
        }
    }
})
```

There's a lot happening in this component. We're accessing two additional actions - deleteList and deleteRow. We're also reformatting our list information into `list_data` which is supplied to the table component (an inbuilt function

of Bootstrap Vue). We're linking our to our add/edit list form, this time via the edit route.

We're also linking to our final component, the row edit form:

```
<script type="text/x-template" id="row_form_component_template">

    <b-modal id="list_modal" :title="title_string" size="md" :visible=true hide-
↪footer @hide="$router.go(-1)">

        <b-form-group v-if="mode=='create'" id="index_form_group">
            Index to add row at: <b> [[ index ]] </b>
            <b-form-input id="index" v-model="index" type="range" min="0" :max="row_
↪length">
            </b-form-input>
        </b-form-group>
        <span class="mb-3" v-else>You are editing the item at index: [[ index ]]</
↪span>

        <b-form-group id="content_form_group" label="Row contents" label-for="row_
↪contents">
            <b-form-textarea id="row_contents" name="row_contents" v-model="content"
                placeholder="Add your contents">
            </b-form-textarea>
        </b-form-group>

        <b-button v-if="mode=='create'" variant="outline-secondary" class="btn-sm" id=
↪"add_row_button"
            @click="add_row">submit</b-button>
        <b-button v-if="mode=='edit'" variant="outline-secondary" class="btn-sm" id=
↪"edit_row_save_button"
            @click="edit_row">submit</b-button>

        <error-component :message=error_message></error-component>

    </b-modal>

</script>

<script type="application/javascript">

    var listRowFormComponent = Vue.component('list-row-form-component', {
        delimiters: ['[[', ']]'],
        template: '#row_form_component_template',
        props: ['list_id', 'row_index', 'mode'],
        store,
        data: function() {
            return {
                index: null,
                content: null,
                row_length: 0,
                error_message: null
            }
        },
        created () {
            if (this.lists_loaded) { this.get_initial_data() }
            else { this.getLists().then(response => { this.get_initial_data() }) }
        },
        computed: {
```

(continues on next page)

(continued from previous page)

```
            ...Vuex.mapState({ lists: state => state.simplelists.lists }),
            ...Vuex.mapGetters(['getListData']),
            lists_loaded: function() { if (this.lists.length == 0) { return false }␣
↪else { return true }},
            title_string: function() {
                if (this.mode == 'edit') {  return "Edit row '" + this.row_index + "'
↪" }
                else { return "Add a new row" }
            }
        },
        methods: {
            ...Vuex.mapActions(['getLists', 'addRow', 'editRow']),
            get_initial_data() {
                list = this.getListData(this.list_id)
                if (this.mode == 'edit') {
                    this.index = this.row_index
                    this.content = list.rows[this.row_index]
                    this.row_length = list.rows.length
                } else {
                    this.index = 0
                    this.row_length = list.rows.length
                }
            },
            add_row() {
                this.addRow({list_pk:this.list_id, index:parseInt(this.index), row_
↪content: this.content})
                .then(response => {
                    this.$router.push({name: 'list-detail', params: {list_id: this.
↪list_id}})
                }).catch(error => {  this.error_message = error })
            },
            edit_row() {
                this.editRow({list_pk:this.list_id, index:parseInt(this.index), row_
↪content: this.content})
                .then(response => {
                    this.$router.push({name: 'list-detail', params: {list_id: this.
↪list_id}})
                }).catch(error => {  this.error_message = error })
            }
        }
    })

</script>
```

This is a lot (and I hope to revisit these docs to break them down a little better) but hopefully provides a decent roadmap for adding the front-end for your resource.

To hook up all the components in Vue-router, we specify a name, path, and components for every route. For now, this information is stored in the `group_detail.html` template.

```
{
    name: 'add-new-list',
    path: '/lists/new',
    components: {
        sidebar: groupConfigComponent,
        main: resourcesComponent,
        modal: listFormComponent
```

(continues on next page)

(continued from previous page)

```
    }
},
{
    name: 'edit-list-info',
    path: '/lists/edit/:list_id',
    props: { sidebar: false, main: false, modal: true },
    components: {
        sidebar: groupConfigComponent,
        main: resourcesComponent,
        modal: listFormComponent
    }
},
{
    name: 'list-detail',
    path: '/lists/detail/:list_id',
    props: { sidebar: false, main: true},
    components: {
        sidebar: groupConfigComponent,
        main: listComponent
    }
},
{
    name: 'edit-list-row',
    path: '/lists/detail/:list_id/rows/:mode/:row_index',
    props: { sidebar: false, main: true, modal: true},
    components: {
        sidebar: groupConfigComponent,
        main: listComponent,
        modal: listRowFormComponent
    }
},
{
    name: 'add-list-row',
    path: '/lists/detail/:list_id/rows/:mode',
    props: { sidebar: false, main: true, modal: true},
    components: {
        sidebar: groupConfigComponent,
        main: listComponent,
        modal: listRowFormComponent
    }
},
```

Final Infrastructure

There's a few last pieces to add.

First, let's talk about those check permissions calls. They're get API calls which check whether the user can take the specified action on the given target. (By default, the target is the group/community - the alt_target parameter allows us to feed in a model + pk to specify a different target). We'll also need to edit the `check_individual_permissions` function in `views.py` to handle these queries:

```
# lists
if permission_name == "add_list":
    return client.PermissionResource.has_permission(client.List, "add_list", {"name":
↪"ABC"})
```

(continues on next page)

(continued from previous page)

```
if permission_name == "edit_list":
    return client.PermissionResource.has_permission(client.List, "edit_list", {"name
↪": "DEF"})
if permission_name == "delete_list":
    return client.PermissionResource.has_permission(client.List, "delete_list", {})
if permission_name == "add_row":
    return client.PermissionResource.has_permission(client.List, "add_row", {"row_
↪content": "ABC"})
if permission_name == "edit_row":
    return client.PermissionResource.has_permission(client.List, "edit_row", {"row_
↪content": "ABC", "index":0})
if permission_name == "delete_row":
    return client.PermissionResource.has_permission(client.List, "delete_row", {"index
↪": 0})
```

Very rarely, check_permissions will require an additional parameter to be passed in, but none of ours require it.

The next thing we want to do is create and access a view for seeing which permissions are set on our list. This is as simple as adding a router link in our list detail component:

```
<router-link :to="{ name: 'list-permissions',
    params: {list_id: list_id, item_id: list_id, item_model: 'simplelist', item_name:␣
↪list_name }}">
        <b-button variant="outline-secondary" id="list_permissions" class="btn-sm"">
            list permissions</b-button>
</router-link>
```

And we'll do the same thing for the action history of our list:

```
<router-link :to="{ name: 'list-action-history',
    params: {list_id: list_id, item_id: list_id, item_model: 'simplelist', item_name:␣
↪list_name }}">
        <b-button variant="outline-secondary" class="btn-sm"" id="list_history_button
↪">
            list history</b-button>
</router-link>
```

The last thing we need to do is make sure our vue-router has a route for these.

```
{
    name: 'list-action-history',
    path: '/lists/detail/:list_id/history/:item_id/:item_model/:item_name',
    props: { sidebar: false, main: true, modal: true},
    components: {
        sidebar: groupConfigComponent,
        main: listComponent,
        modal: actionHistoryComponent
    }
},
{
    name: 'list-permissions',
    path: '/lists/detail/:list_id/permissions/:item_id/:item_model/:item_name',
    props: { sidebar: false, main: true, modal: true },
    components: {
        sidebar: groupConfigComponent,
        main: listComponent,
        modal: permissionedItemComponent
```

(continues on next page)

(continued from previous page)

```
    }
},
```

Success!

At the end of this process, your front end should look something like this:

Enhancements

Let's go back and add a little more functionality to our SimpleLists. Instead of having lists with only one column named "content", we're going to give our users the ability to specify an arbitrary number of columns with whatever names they like.

To keep things manageable for us, we'll specify that the column names and contents will always be strings. We'll let users specify whether a column is required or not, though, and supply a default when they're adding a new required field to an existing list. When users create or edit a list, they can change the configuration of the table, and when they go to add or edit rows they'll be prompted to fill out the column fields.

Step 1: Models

We'll start by updating our models. First we'll add a new field and two methods to get and set data on it.

```
class SimpleList(PermissionedModel):
    """Model to store simple lists with arbitrary fields."""

    name = models.CharField(max_length=200)
    description = models.CharField(max_length=200)
    rows = models.TextField(list)
    row_configuration = models.TextField(list)

    def get_row_configuration(self):
        """Gets row configuation json and loads to Python dict."""
        if self.row_configuration:
            return json.loads(self.row_configuration)
        return {}

    def set_row_configuration(self, row_configuration):
        """Given a row configuration with format, saves to DB."""
        self.row_configuration = json.dumps(row_configuration)
```

When setting our configuration, we should check that the row configuration supplied is a valid configuration. We'll create a helper method to do this. This method checks for a variety of ways that the configuration might be invalid, and hopefully covers most cases.

```
def validate_configuration(self, row_configuration):
    """Checks that a given configuration is valid.  Should have format:
    { field_name : { 'required': True, 'default_value': 'default'}}
    If required is not supplied, defaults to False.  If default_value is not supplied,
↪ defaults to None."""
```

(continues on next page)

(continued from previous page)

```
    if not isinstance(row_configuration, dict):
        raise ValidationError(f"List configuration must be a dict, not {type(row_
↪configuration)}")
    if len(row_configuration.items()) < 1:
        raise ValidationError("Must supply at least one column to configuration.")
    field_name_list = []
    for field_name, params in row_configuration.items():
        if field_name in field_name_list:
            raise ValidationError(f"Field names must be unique. Multiple instances of␣
↪field {field_name}")
        field_name_list.append(field_name)
        params["required"] = params["required"] if "required" in params else False
        params["default_value"] = params["default_value"] if "default_value" in␣
↪params else None
        if not isinstance(params["required"], bool):
            raise ValidationError(f"Required parameter for {field_name} must be True␣
↪or False, " +
                                  f"not {type(params['required'])}")
        if params["default_value"] and not isinstance(params["default_value"], str):
            raise ValidationError(f"default_value parameter for {field_name} must be␣
↪str, not " +
                                  f"{type(params['default_value'])}")
        if set(params.keys()) - set(["required", "default_value"]):
            unexpected_keys = list(set(params.keys()) - set(["required", "default_
↪value"]))
            raise ValidationError(f"unexpected keys {unexpected_keys} in row␣
↪configuration")
```

Now we can update our `set_row_configuration` method to use the helper:

```
def set_row_configuration(self, row_configuration):
    """Given a row configuration with format, validated and saves to DB."""
    self.validate_configuration(row_configuration)
    self.row_configuration = json.dumps(row_configuration)
```

We also need a way to check that new row data matches our configuration. Let's start by creating a helper method which checks a given row against the current configuration:

```
def check_row_against_configuration(self, row):
    """Given a row, check that it's valid for the row configuration."""
    config = self.get_row_configuration()
    for field_name, params in config.items():
        if params["required"]:
            if field_name not in row or row[field_name] in ["", None]:
                if not params["default_value"]:
                    raise ValidationError(f"Field {field_name} is required with no␣
↪default_value, " +
                                          "so must be supplied")
    for field_name, params in row.items():
        if field_name not in config:
            field_names = ", ".join([field_name for field_name, params in config.
↪items()])
            raise ValidationError(f"Field {field_name} is not a valid field, must be␣
↪one of {field_names}")
```

For now, we just check that all the fields in the row exist in the configuration, and that all required fields are there (or that there's a default value for them).

We also want to create a method that supplies any rows with missing values in a required field with default values:

```
def handle_missing_fields_and_values(self, row):
    """Given a row, check that it's valid for the row configuration."""
    config = self.get_row_configuration()
    for field_name, params in config.items():
        if field_name not in row:
            row[field_name] = ""
        if params["required"] and not row[field_name]:
            row[field_name] = params["default_value"]
    return row
```

Now that we've made these two methods, we can add them to add_row and edit_row, for example:

```
def edit_row(self, row, index):
    """Edit a row in the list."""
    self.check_row_against_configuration(row)
    row = self.handle_missing_fields_and_values(row)
    rows = self.get_rows()
    rows[index] = row
    self.rows = json.dumps(rows)
```

We also need to handle when a user wants to change the list configuration and there are already rows in the database. We'll create a method which can be called by edit_list:

```
def adjust_rows_to_new_configuration(self, configuration):
    """Given a new row configuration, goes through existing rows and adjusts them␣
↪them."""
    required_fields = [field_name for field_name, params in configuration.items() if␣
↪params["required"] is True]
    adjusted_rows = []
    for row in self.get_rows():
        new_row = {}
        for row_field_name, row_field_value in row.items():
            if row_field_name in configuration:  # leaves behind fields not in new␣
↪config
                new_row.update({row_field_name: row_field_value})
        for field in required_fields:
            if field not in row:
                new_row[field] = None
            if field in row and row[field]:
                new_row[field] = row[field]
            else:
                default_value = configuration[field].get("default_value", None)
                if default_value:
                    new_row[field] = default_value
                else:
                    raise ValidationError(f"Need default value for required field
↪{field}")
        adjusted_rows.append(new_row)
    self.rows = json.dumps(adjusted_rows)
```

Let's add a reference to this too when setting a new configuration:

```
def set_row_configuration(self, row_configuration):
    """Given a row configuration with format, validated and saves to DB."""
    self.validate_configuration(row_configuration)
    self.adjust_rows_to_new_configuration(row_configuration)
```

(continues on next page)

(continued from previous page)

```
    self.row_configuration = json.dumps(row_configuration)
```

Step 2: State Changes

This should be all we need to change on our model. Let's go update our state changes. We'll need to update AddList, EditList, AddRow and EditRow.

We'll start with AddList. We'll need to add it as an input parameter to our `__init__`, and add a corresponding entry to our `input_fields`. We'll also add it when we're creating our SimpleList object in `implement`. The most important change we'll make is overridding the parent `validate` method to add our call to `validate_configuration`.

```
class AddListStateChange(BaseStateChange):
    """State Change to create a list in a community (or other target)."""
    description = "Add list"
    input_fields = [InputField(name="name", type="CharField", required=True,␣
↪validate=True),
                    InputField(name="configuration", type="DictField", required=True,␣
↪validate=False),
                    InputField(name="description", type="CharField", required=False,␣
↪validate=True)]
    input_target = SimpleList

    def __init__(self, name, configuration, description=None):
        self.name = name
        self.configuration = configuration
        self.description = description if description else ""

    @classmethod
    def get_allowable_targets(cls):
        return cls.get_community_models()

    def description_present_tense(self):
        return f"add list with name {self.name}"

    def description_past_tense(self):
        return f"added list with name {self.name}"

    def validate(self, actor, target):
        if not super().validate(actor=actor, target=target):
            return False
        try:
            SimpleList().validate_configuration(self.configuration)
            return True
        except ValidationError as error:
            self.set_validation_error(message=error.message)
            return False

    def implement(self, actor, target):
        simple_list = SimpleList(name=self.name, description=self.description,␣
↪owner=target.get_owner())
        simple_list.set_row_configuration(self.configuration)
        simple_list.save()
        return simple_list
```

The Edit List state change is similar - we add the new configuation parameter to `__init__` and call our

`validate_configuration` method in `validate`. In addition to checking that the configuration is valid, we'll also need to check that the existing rows can be updated to the new configuration withour raising an error. Our Edit List state change's validate method now looks like this:

```
def validate(self, actor, target):
    if not super().validate(actor=actor, target=target):
        return False
    if not self.name and not self.description and not self.configuration:
        self.set_validation_error(message="Must supply new name, description, or␣
↪configuration when editing List.")
        return False
    if self.configuration:
        try:
            target.validate_configuration(self.configuration)
            target.adjust_rows_to_new_configuration(self.configuration)
            return True
        except ValidationError as error:
            self.set_validation_error(message=error.message)
            return False
    return True
```

In the implement method, we can call `set_row_configuration`, knowing it will validate the configuration and the adjusted rows again, as there is a chance the target may have changed since the action was first validated:

```
def implement(self, actor, target):
    target.name = self.name if self.name else target.name
    target.description = self.description if self.description else target.description
    if self.configuration:
        target.set_row_configuration(self.configuration)
    target.save()
    return target
```

There's no changes to the Delete List state change (or the Delete Row state change, for that matter) so lets move on to the Add Row state change. We can keep the `row_content` variable but instead of assuming it's a simple string, we're going to assume it's a dictionary of fields and values. When we validate the state change, we'll call our `check_row_against_configuration` to make sure the row_content is formatted acceptably:

```
def validate(self, actor, target):
    if not super().validate(actor=actor, target=target):
        return False
    try:
        target.check_row_against_configuration(self.row_content)
    except ValidationError as error:
        self.set_validation_error(message=error.message)
        return False
    if self.index and not isinstance(self.index, int):
        self.set_validation_error(message="Index must be an integer.")
        return False
    return True
```

We'll make the same update to the validate method of the edit row state change as well. And that's it! We shouldn't need to make any more adjustments to our state changes.

Step 3: Client

The next step is to update our client, which should be very straightforward. The only methods we need to adjust are `add_list` and `edit_list`:

```
def add_list(self, name, configuration, description=None):
    change = sc.AddListStateChange(name=name, configuration=configuration,␣
↪description=description)
    return self.create_and_take_action(change)

def edit_list(self, name=None, configuration=None, description=None):
    change = sc.EditListStateChange(name=name, configuration=configuration,␣
↪description=description)
    return self.create_and_take_action(change)
```

Before we move on to adding our views, you should migrate your database. Because you're changing an existing model, you'll need to provide a default. The following JSON string should be a good default but feel free to adjust to your needs: `'{"content": {"required": true}}'` The rows will also be out of format, you'll want to run a command like the following in your shell:

```
for simple_list in SimpleList.objects.all():
    new_rows = [{'content': row_content } for row_content in simple_list.get_rows()]
    simple_list.rows = json.dumps(new_rows)
    simple_list.save(override_check=True)
```

Step 4: Views

Now, on to our views! We'll need to accept the configuration parameter parameter from the front end in our `add_list` and `edit_list` views, for example:

```
@login_required
@reformat_input_data
def add_list(request, target, name, configuration, description=None):

    client = Client(actor=request.user)
    target = client.Community.get_community(community_pk=target)
    client.List.set_target(target=target)

    action, result = client.List.add_list(name=name, configuration=configuration,
        description=description)

    action_dict = get_action_dict(action)
    if action.status == "implemented":
        action_dict["list_data"] = serialize_list_for_vue(result)
    return JsonResponse(action_dict)
```

We'll also want to include configuration data when serializing our lists:

```
def serialize_list_for_vue(simple_list):
    return {'pk': simple_list.pk, 'name': simple_list.name, 'description': simple_
↪list.description,
            'configuration': simple_list.get_row_configuration(), 'rows': simple_list.
↪get_rows()}
```

Step 5: Vue

The last thing we need to do is update our components. We'll need to update our list detail view so it displays our columns correctly, our form field so that we can specify the configuration there, and the row form so that it provides the user with our configured fields as fields to fill out.

List Detail Component

First let's update our detail view. We'll start by making a quick access method for the configuration:

```
configuration: function() {
    if (this.lists_loaded) { return this.getListData(this.list_id).configuration }␣
↪else { return {} }},
```

We'll also turn our list_fields (aka columns used by our table) into a computed variable:

```
list_fields: function() {
    list_fields = [{ key: 'index', sortable: true }]
    if (this.lists_loaded) {
        for (field in this.configuration) {
            list_fields.push({key: field, sortable: true})
        }
    }
    list_fields.push('change')
    return list_fields
},
```

Finally, we'll reformat the row data to match the new system as well:

```
list_data: function() {
    list_data = []
    index = 0
    for (row in this.rows) {
        row_dict = {}
        for (field in this.rows[row]) {
            row_dict[field] = this.rows[row][field]
        }
        row_dict["index"] = index
        list_data.push(row_dict)
        index++
    }
    return list_data
}
```

Add/Edit List Component

This should handle our display, and your table should like it did before, as we haven't actually added any new columns. Let's add the ability to do that now, in the list form.

We'll start by creating the html which will display our columns. This has two sections - a simple table which displays the current list of columns, including whether they're required or have a default value, and a short form which allows us to add new columns. We'll have two buttons, one which adds a new column and one which deletes an existing column.

```
<span id="list_columns" class="my-4">

    <b-table-lite striped :items="columns" caption-top>
        <template v-slot:table-caption>Existing columns</template>
        <template v-slot:cell(delete)="data"><b-button @click='delete_column(data.
↪item)'>
            delete</b-button></template>
    </b-table-lite>

    <error-component :message=column_error_message :dismissable=true></error-
↪component>

    <p class="my-2">Add a new column:</p>

    <b-form inline>
        <b-form-input id="column_name" v-model="column_name" placeholder="column name
↪" required
            class="mr-2"></b-form-input>
        <b-form-input id="column_default" v-model="column_default" placeholder=
↪"default value"
            class="mr-2"></b-form-input>
        <b-form-checkbox id="column_required" v-model="column_required" class="mr-2">
            required</b-form-checkbox>
        <b-button variant="info" class="btn-sm" id="add_new_col_button"
            @click="add_column()">add</b-button>
    </b-form>

</span>
```

The format in which our configuration is stored is not quite right to display on the table, so let's create two helper methods to reformat back and forth, which we'll add to the methods section of our component:

```
reformat_columns(input) {
    columns = []
    for (column in input) {
        required = input[column].required ? input[column].required : false
        default_value = input[column].default_value ? input[column].default_value : ""
        columns.push({name: column, required: required, default_value: default_value,␣
↪delete: null})
    }
    return columns
},
reformat_columns_for_backend(columns) {
    configuration = {}
    for (index in columns) {
        column = columns[index]
        configuration[column.name]= { required: column.required, default_value:␣
↪column.default_value }
    }
    console.log(configuration)
    return configuration
},
```

We'll add calls to these reformatters when loading up the form. Note our default column structure for a new list:

```
populate_columns() {
    if (this.list_id) {
```

(continues on next page)

(continued from previous page)

```
        if (this.lists_loaded) {
            list = this.getListData(this.list_id)
            this.name = list.name
            this.description = list.description
            this.columns = this.reformat_columns(list.configuration)
        }
    } else {
        this.columns = this.reformat_columns({"content": {}})
    }
},
```

Next, let's fill out those add_column and delete_column methods, adding some basic validation:

```
add_column() {
    if (!this.column_name) {
        this.column_error_message = "New column must have a name"
        return
    }
    existing_column = this.columns.find(column => column.name == this.column_name)
    if (existing_column) {
        this.column_error_message = "Columns must have unique names"
        return
    }
    if (this.list_id && this.column_required && this.column_default == "") {
        this.column_error_message = "If column is required, must supply default value"
        return
    }
    this.columns.push({name: this.column_name, required: this.column_required,
        default_value: this.column_default, delete: null})
},
delete_column(column_to_delete) {
    if (this.columns.length == 1) {
        this.column_error_message = "Must have at least one column"
    } else {
        index = 0
        for (column in this.columns) {
            if (this.columns[column].name == column_to_delete.name) {
                this.columns.splice(index, 1)
            }
            index++
        }
    }
}
```

And, lastly, we insert the new configuration into our `add_list` and `edit_list` actions, making sure to reformat as we do so:

```
add_list() {
    this.addList({ name: this.name, description: this.description,
        configuration: this.reformat_columns_for_backend(this.columns)})
    .catch(error => {  console.log(error), this.error_message = error })
},
edit_list() {
    this.editList({ list_pk: parseInt(this.list_id), name: this.name, description:␣
↪this.description,
        configuration: this.reformat_columns_for_backend(this.columns) })
```

(continues on next page)

(continued from previous page)

```
    .catch(error => {  this.error_message = error })
},
```

Add/Edit Row Component

The final change we need to make is in the row form. When we're adding and editing rows now, we should be given the new columns as options. When a column is required, the form should indicate that, and when a default value is provided, it should let the user know that, if they don't provide a value, that's what will be used.

We add the following html to our row form template:

```
<span class="my-3">
    <b-form-group v-for="column in columns" :label=column.name :label-for=column.name␣
↪v-bind:key=column.name>
        <b-form-input :id=column.name v-model=column.current_value></b-form-input>
        <b-form-text>
            <span v-if=is_column_required(column)>You must fill out a value for this␣
↪column</span>
            <span v-if=column.default_value>The default value for this column is
                [[ column.default_value ]].</span>
        </b-form-text>
    </b-form-group>
</span>
```

This is fairly straightforward. For each column, we create an input field with the column name as the label, the current value of that field, and some helper text regarding whether the field's required and what, if any, default value it has. Here's the Javascript to go along with it, in our component:

```
get_initial_data() {
    list = this.getListData(this.list_id)
    this.row_length = list.rows.length
    if (this.mode == 'edit') { this.index = this.row_index } else { this.index = 0 }
    row_data = (this.mode == 'edit') ? list.rows[this.index] : null
    this.columns = this.initialize_columns(list.configuration, row_data)
},
initialize_columns(configuration, row_data){
    columns = []
    for (col_name in configuration){
        params = configuration[col_name]
        current_value = row_data ? row_data[col_name] : ""
        columns.push({name: col_name, required: params.required,
            default_value: params.default_value, current_value: current_value })
    }
    return columns
},
```

Here, we restructure the configuration fields to be a list of maps, with name, required, default_value and current_value as the keys. If we're in editing mode, row_data is passed in to the method and can be used to populate the current value.

We make two more changes to our component's methods:

```
format_row_content() {
    column_data = {}
    for (index in this.columns) {
```

(continues on next page)

(continued from previous page)

```
        column = this.columns[index]
        column_data[column.name] = document.getElementById(column.name).value
    }
    return column_data
},
is_column_required(column){
    if (column.required && column.default_value == "") { return true } else { return␣
↪false }
}
```

`format_row_content()` is called in the `add_row` and `edit_row` method to get the final version of the row content parameter which will be passed back all the way to our models. Finally, `is_column_required` is a simple helper method to help remind the user that a field is required.

When you're all done, the interface should now look something like this:

3.2.5 Example Implementation

It may be helpful to look at a concrete implementation. Shown below is just one possible implementation of Concord, of the many that the library has been designed to encompass.

This walkthrough demonstrates Concord as implemented on our prototype website, Kybern.org. If you would like access to Kybern.org as a beta user, please contact us. Here we cover a simple series of actions demonstrating some of the concepts discussed in the design section.

1: A user creates a new group. By default, she is added as both owner and governor:

2: The user, using her governing permission, adds new members to the group.

3: The user, using her governing permission, creates new roles and adds those new members to roles.

4: The user assigns one of the roles, voting members, to be the new owner. She adds a voting condition.

5: The user gives the other role, general members, permission to change the description of the group. She adds an approval condition from a voting member.

6: A general member tries to change the name of the group. There's a condition on their action. We go to the action history and see that it's waiting.

7: A voting member goes and has the ability. They approve. We check and see that the action is implemented and the name is different now.

3.2.6 Autodocumentation of the Actions module

Models

Django models for Actions and Permissioned Models.

class concord.actions.models.**Action**(**args*, ***kwargs*)
Represents an action between an actor and a target.

All changes of state that go through the permissions system must do so via an Action instance.

Action instances must include information about the actor taking the action and the target of the action, among other information.

exception DoesNotExist

exception MultipleObjectsReturned

get_description()
Gets description of the action by reference to *change_types* set via change field, including the target.

get_targetless_description()
Gets description of the action by reference to *change_types* set via change field, without the target.

implement_action()
Perform an action defined by the Change object. Carries out its custom implementation using the actor and target.

save(**args*, ***kwargs*)
If action is live (is_draft is False) check that target and actor are set.

property status
Gets status of Action from Resolution field.

take_action()
Runs the action through the permissions pipeline. If waiting on a condition, triggers that condition. If approved, implements action.

Returns itself and, optionally, the result of implementing the action.

class concord.actions.models.**PermissionedModel**(**args*, ***kwargs*)
An abstract base class that represents permissions.

PermissionedModel is an abstract base class from which all models using the permissions system should inherit.

The *PermissionedModel* contains information about owners and their related permissions.

get_actions()
Provides a helper method for getting actions.

Returns all actions targeting the instance of the subclass which has inherited from *PermissionedModel*.

get_content_type()
Gets content type of the model.

The content type is helpful since *PermissionedModels* may be subclassed into a variety of other models.

get_name()
Gets name of Model. By default, gets string representation.

get_nested_objects()
Gets objects that the model is nested within.

Nested objects are often things like the owner of instance or, for example, a forum that a post is posted within.

Called by the permissions pipeline in *permissions.py*.

get_owner()
Gets owner of the permissioned model.

All permissioned models have an owner.

get_serialized_field_data()
Returns data that has been been serialized.

By default, the readable attributes of a permissioned model are all fields specified on the model. However, we cannot simply use *self._meta.get_fields()* since the field name is sometimes different than the attribute name, for instance with related fields that are called X but show up as X_set on the model.

classmethod get_settable_state_changes()
Returns a list of state_changes that can be set via permissions targeting this model.

This may include some permissions where the *targets* are other than this model. For instance, if this object owns another object, we may have set permissions for actions targeting the owned object.

get_unique_id()
Gets a unique ID for the model, consisting of the content type and pk.

save(**args*, *override_check=False*, ***kwargs*)
Save permissions.

There are two things happening here:

1: Subtypes of *BaseCommunity* are the *only* children of *PermissionedModel* that should be allowed to have a null owner. We check that here and raise an error if a non-community model has null values for owner fields.

2: A permissioned model's save method can *only* be invoked by a descendant of *BaseStateChange*, on update (create is fine). For now, we inspect who is calling us, but there may be a better long-term solution.

class concord.actions.models.**TemplateModel**(**args*, ***kwargs*)
The template model allows users to apply sets of actions to their communities.

exception DoesNotExist

exception MultipleObjectsReturned

get_scopes()
Gets list of scopes the template model applies to.

get_supplied_fields()
Get supplied fields as dictionary.

get_supplied_form_fields()
Loads template supplied fields and gets their forms, using field_helper. Supplied fields typically have format like:

"field_x": ["RoleListField", None] "field_y": ["IntegerField", { "maximum": 2 }]

get_template_breakdown(*trigger_action=None*, *supplied_field_data=None*)
Gets a breakdown of actions contained in the template, including data from trigger action and supplied field data if passed in.

property has_foundational_actions
Returns True if any of the actions in the action_list are foundational changes.

set_scopes(*scopes*)
Saves a list of scopes to the template model.

State Changes

Defines state changes for concord.actions.models, as well as the BaseStateChange object from which all state change objects inherit.

class concord.actions.state_changes.**ApplyTemplateStateChange**(*template_model_pk*, *supplied_fields=None*, *is_foundational=False*, *original_creator_only=False*)

State change object for applying a template.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

classmethod get_configured_field_text(*configuration*)
Gets additional text for permissions item instance descriptions from configured fields.

implement(*actor*, *target*, *action=None*)
Implements the given template, relies on logic in apply_template.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.actions.state_changes.**BaseStateChange**
The BaseStateChange object is the object which all other state change objects inherit from. It has a variety of methods which must be implemented by those that inherit it.

all_context_instances(*action*)
Given the specific action that contains this change object, returns a dictionary with relevant model_instances. Used primarily by templates.

We always return the action, the owning group by its model name, and the action target by its model name, with the state change able to specify additional objects.

classmethod can_set_on_model(*model_name*)
Tests whether a given model, passed in as a string, is in allowable target.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_all_possible_targets()
Helper method, gets all permissioned models in system that are not abstract.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

get_change_data()
Given the python Change object, generates a json list of field names and values. Does not include instantiated fields.

classmethod get_change_field_options()
Gets a list of required parameters passed in to init, used by templates. Does not include optional parameters, as this may break the template referencing them if they're not there.

classmethod get_change_type()
Gets the full type of the change object in format 'concord.package.state_changes.SpecificStateChange'

classmethod get_community_models()
Helper method which lets us use alternative community models as targets for community actions.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

classmethod get_configurable_form_fields()
Gets the configurable fields of a change object as form fields.

classmethod get_configured_field_text(*configuration*)
Gets additional text for permissions item instance descriptions from configured fields.

get_context_instances(*action*)
Method to be optionally overridden by State Changes, adding context instances.

classmethod get_context_keys()
Gets action as key by default, plus any context keys specified. If no context keys are specified and allowable_targets incldues only one model, grabs that model name as a valid context key.

classmethod get_preposition()
By default, we make changes "to" things but change types can override this default preposition with "for", "with", etc.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

is_conditionally_foundational(*action*)
Some state changes are only foundational in certain conditions. Those state changes override this method to apply logic and determine whether a specific instance is foundational or not.

set_validation_error(*message*)
Helper method so all state changes don't have to import ValidationError

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.actions.state_changes.**ChangeOwnerStateChange**(*new_owner_content_type*, *new_owner_id*)
State change for changing which community owns the object. Not to be confused with state changes which change who the owners are within a community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

get_new_owner()
Helper method to get model instance of new owner from params.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.actions.state_changes.**DisableFoundationalPermissionStateChange**
State change object for disabling the foundational permission of a permissioned model.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.actions.state_changes.**DisableGoverningPermissionStateChange**
State change object for disabling the governing permission of a permissioned model.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.actions.state_changes.**EnableFoundationalPermissionStateChange**
State change object for enabling the foundational permission of a permissioned model.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.actions.state_changes.**EnableGoverningPermissionStateChange**
State change object for enabling the governing permission of a permissioned model.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.actions.state_changes.**InputField**(*name*, *type*, *required*, *validate*)

property name
Alias for field number 0

property required
Alias for field number 2

property type
Alias for field number 1

property validate
Alias for field number 3

class concord.actions.state_changes.**ViewStateChange**(*fields_to_include=None*)
ViewStateChange is a state change which doesn't actually change state. Instead, it returns the specified fields. It exists so we can wrap view permissions in the same model as all the other permissions.

check_configuration(*action*, *permission*)
All configurations must pass for the configuration check to pass.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

implement(*actor*, *target*)
Gets data from specified fields, or from all fields, and returns as dictionary.

validate(*actor*, *target*)
Checks if any specified fields are not on the target and, if there are any, returns False.

Client

Client for making changes to models in Action.models, along with the BaseClient which other packages inherit from.

class concord.actions.client.**ActionClient**(*actor=None*, *target=None*)
The ActionClient provides access to Action and ActionContainer models.

Args:

actor: User Model The User who the client is acting on behalf of. Optional, but required for many Client methods.

target: PermissionedModel Model The target that the change will be implemented on. Optional, but required for many Client methods.

get_action_given_pk(*pk*)
Takes a pk (int) and returns the Action associated with it.

get_action_history_given_actor(*actor=None*) → django.db.models.query.QuerySet
Gets the action history of an actor. Accepts an User model passed in or, if no actor is passed in, uses the actor currently set on the client.

get_action_history_given_target(*target=None*) → django.db.models.query.QuerySet
Gets the action history of a target. Accepts a target model passed in or, if no target is passed in, uses the target currently set on the client.

get_foundational_actions_given_target(*target=None*) → django.db.models.query.QuerySet
Gets the action history of a target, filtered to include only foundational changes.

get_governing_actions_given_target(*target=None*) → django.db.models.query.QuerySet
Gets the action history of a target, filtered to only include actions resolved via the governing permission.

get_owning_actions_given_target(*target=None*) → django.db.models.query.QuerySet
Gets the action history of a target, filtered to only include actions resolved through foundational permission. Similar to filtering foundational_actions, but includes non-foundational actions taken on targets with the foundational permission enabled.

retake_action(*action=None*, *pk=None*)
Helper method to take an action (or, usually, retry taking an action) from the client.

class concord.actions.client.**BaseClient**(*actor=None*, *target=None*)
Contains behavior needed for all clients.

Args:

actor: User Model The User who the client is acting on behalf of. Optional, but required for many Client methods.

target: PermissionedModel Model The target that the change will be implemented on. Optional, but required for many Client methods.

change_is_valid(*change*)
Returns True if the change passed in is valid, given the Client's actor and target, and False if it is not.

change_owner_of_target(*new_owner*) → Tuple[int, Any]
Changes the owner of the Client's target.

Args:

new_owner: descendant of base Community Model The new owner the target will be transferred to.

create_action(*change*)
Create an Action object using the change object passed in as well as the actor and target already set on the Client. Called by clients when making changes to state.

If the mode set on the client is "Mock", creates a mock action intead and returns it. Mocks are mostly used by Templates.

create_and_take_action(*change*, *proposed=False*)
Creates an action and takes it.

disable_foundational_permission() → Tuple[int, Any]
Disables the foundational permission on a target. Foundational permission is typically disabled.

disable_governing_permission() → Tuple[int, Any]
Disables the governing permission on a target. This prevents governors from taking actions on the target unless they're granted specific permissions. Governing permission is typically enabled.

enable_foundational_permission() → Tuple[int, Any]
Enables the foundational permission on a target. This overrides all specific permissions and governing permissions and requires changes to the target to be made by owners. Foundational permission is typically disabled.

enable_governing_permission() → Tuple[int, Any]
Enables the governing permission on a target. This allows anyone who is a governor to take any non-foundational action on the target. Governing permission is typically enabled.

get_object_given_model_and_pk(*model*, *pk*)
Given a model string and a pk, returns the instance. Only works on Permissioned models.

get_target()
Gets the target of the client.

get_target_data(*fields_to_include=None*)
Gets information about the target after passing request through permissions pipeline. Supply fields_to_include, a list of field names as strings, to limit the data requested, otherwise returns all fields.

optionally_overwrite_target(*target*)
Helper method that takes in a target, which may be None, and overwrites the existing target only if not None.

refresh_target()
Re-populates model from database.

set_actor(*actor*)
Sets actor.

set_target(*target=None*, *target_pk=None*, *target_ct=None*)
Sets target of the client. Accepts either a target model or the target's pk and ct and fetches, in which case it fetches the model from the Database. Target must be a permissioned model.

take_action(*action*, *proposed=False*)
If the action is a mock, invalid, or proposed, return without taking it, otherwise take the action.

validate_actor()
Helper method to check whether or not we've got an actor and whether they're a user.

validate_target()
Helper method to check whether or not we've got a target and that it's a permissioned model.

class concord.actions.client.**TemplateClient**(*actor=None*, *target=None*)
The TemplateClient provides access to the TemplateModel model.

Args:

actor: User Model The User who the client is acting on behalf of. Optional, but required for many Client methods.

target: PermissionedModel Model The target that the change will be implemented on. Optional, but required for many Client methods.

apply_template(*template_model_pk*, *supplied_fields=None*)
Applies a template to the target. If any of the actions in the template is a foundational change, changes the state change object's attr to foundational so it goes through the foundational pipeline.

get_template(*pk*)
Gets template with supplied pk or returns None.

get_templates()
Gets all templates in database.

get_templates_for_scope(*scope*)
Gets template in the given scope.

Permissions Pipeline

This module implements the logic of the permission system.

The *has_permission* function is called by external callers, while the rest of the functions are used by has_permission.

concord.actions.permissions.**check_conditional**(*action*, *community_or_permission*, *leadership_type=None*)

Checks to see if a condition item has been created for the action.

Args:

action: Action Model The Action which is being passed through the permissions pipeline.

community_or_permission: Model Either community model or permission model.

leadership_type: str Either "owner" or "governor". Required if community_or_permission is community. ignored if community_or_permission is permission.

Returns a dict containing condition information if condition_item exists or, if condition item does not exist, the same dict structure populated by Nones.

concord.actions.permissions.**check_specific_permission**(*permission*, *action*)

Helper method called by specific permissions pipeline. Given a permission, checks to see if its active, if it has the right configuration, that the actor satisfies the permission, and whether the action has a condition that passes, in that order.

Returns: True or False indicating whether permission passes Str or the role that the actor matched to, if it exists, or None Dict of the condition data from the condition set on the permission, or None

concord.actions.permissions.**foundational_permission_pipeline**(*action*)

Handles logic for foundational actions.

When an action is passed through the foundational pipeline, it is not passed through the governing or specific permission pipeline. So, if we don't have the authority, we reject the action.

concord.actions.permissions.**governing_permission_pipeline**(*action*)

Checks whether the actor behind the action has governing permissions and if so, passes.

concord.actions.permissions.**has_permission**(*action*)

has_permission directs the flow of logic in the permissions pipeline. It returns information about whether the action has permsision to take the action and if there are any conditions that need to be triggered. It does not change the database and it does not alter the action object other than updating its resolution field and, optionally, adding a conditions_list attribute with source_ids for uncreated conditions.

If the foundational permission is enabled or the change type is a foundational change (like change_owner), we go into the foundational permission pipeline and no other pipeline.

If the governing permission is enabled, we try that pipeline. If the action is approved by the governing pipeline and we finish with the permission pipeline, otherwise we move on to the last option, the specific permission pipeline.

concord.actions.permissions.**specific_permission_pipeline**(*action*)

Checks the target for specific permissions matching the change type and configuration of the action.

If matching permissions are found, we check to see if the actor satisfies the permission. If the actor does satisfy, we look for conditions. If any permissions have no condition, the action is approved. If there are conditions, they are saved to the conditions list.

If after this the action is not approved, we check to see whether the target is nested on another object which may have permissions set on it. For instance, the target may be a post in a forum, where the forum has an 'edit post' permission that applies across all posts. We go through the same process for each of the nested permissions.

At the end of all this, if any of these permissions pass, the action is approved. If any are waiting, the action is set to waiting. If none are approved or waiting, the action is rejected.

Utilities

Utility methods/classes for actions package. Contains some Concord-wide utility methods/classes as well.

class concord.actions.utils.**Attributes**
Hack to allow nested attributes on Changes.

class concord.actions.utils.**Changes**
Helper object which lets developers easily access change types.

class concord.actions.utils.**Client**(*actor=None*, *target=None*, *limit_to=None*)
Helper object which lets developers easily access all clients at once.

If supplied with actor and/or target, will instantiate clients with that actor and target.

limit_to is a list of client names, if supplied actors and targets will only be supplied to the specified clients.

property Community
Projects that use Concord may create a new model and client, descending from the Community model and CommunityClient. To handle this scenario, we look for Clients with an attribute community_model and, if something other than the CommunityClient exists, we use that. Users can override this behavior by explicitly setting community_client_override to whatever client they want to use.

get_clients()
Gets a list of client objects set as attributes on Client().

update_actor_on_all(*actor*)
Update actor for all clients.

update_target_on_all(*target*)
Update target for all clients.

class concord.actions.utils.**MockAction**(*change*, *actor*, *target*, *resolution=None*, *unique_id=None*)
Mock Actions are used in place of the Action django model in templates. They are easier to serialize, lack db-dependent fields like created_at, and crucially allow us to replace certain fields or subfields with references to either the trigger action, or action results from previous actions in an action container.

create_action_object(*container_pk*, *save=True*)
Creates an action object given the data set on MockAction plus the container_pk passed in.

property status
Gets status of Action from Resolution field.

concord.actions.utils.**check_permissions_for_action_group**(*list_of_actions*)
Takes in a list of MockActions, generated by clients in mock mode, and runs them through permissions pipeline.

concord.actions.utils.**get_all_apps**(*return_as='app_configs'*)
Get all apps that are part of Concord and the app that is using it. Returns as list of app_configs by default, but can also be returned as app name string by passing 'strings' to return_as.

concord.actions.utils.**get_all_clients**()
Gets all clients descended from Base Client in Concord and the app using it.

concord.actions.utils.**get_all_community_models**()
Gets all non-abstract permissioned models with attr is_community equal to True.

concord.actions.utils.**get_all_conditions**()
Gets all possible condition models in Concord and the app using it.

concord.actions.utils.**get_all_dependent_fields**()
Goes through all PermissionedModels, plus Action, and gets a list of fields. TODO: also get their type, for use on front-end validation?

concord.actions.utils.**get_all_foundational_state_changes**()
Gets all state changes in Concord and app using it that are foundational.

concord.actions.utils.**get_all_permissioned_models**()
Gets all non-abstract permissioned models in the system.

concord.actions.utils.**get_all_state_changes**()
Gets all possible state changes in Concord and the app using it.

concord.actions.utils.**get_all_templates**()
Get all classes with TemplateLibraryObject as parent defined in template_library files, either in Concord or app using Concord.

concord.actions.utils.**get_state_change_object**(*state_change_name*)
Given a full name string, gets the state change object.

concord.actions.utils.**get_state_changes_for_app**(*app_name*)
Given an app name, gets state_changes as list of state change objects.

concord.actions.utils.**get_state_changes_settable_on_model**(*model_class*)
Gets all state changes a given model can be set on. If state_changes is not passed in, checks against all possible state_changes.

concord.actions.utils.**process_field_type**(*field*)
Helper method to inspect field and return appropriate type.

concord.actions.utils.**replace_fields**(*, *action_to_change*, *mock_action*, *context*)
Takes in the action to change and the mock_action, and looks for field on the mock_action which indicate that fields on the action to change need to be replaced. For the change field, and the change field only, also look for fields to replace within.

concord.actions.utils.**replacer**(*key*, *value*, *context*)
Given the value provided by mock_action, looks for fields that need replacing by finding strings with the right format, those that begin and end with {{ }}. Uses information in context object to replace those fields. In the special case of finding something referencing nested_trigger_action (always(?) in the context of a condition being set) it replaces nested_trigger_action with trigger_action.

3.2.7 Autodocumentation of the Communities module

Models

Models for Community package.

class concord.communities.models.**BaseCommunityModel**(**args*, ***kwargs*)
The base community model is the abstract type for all communities. Much of its logic is contained in customfields.RoleField and customfields.RoleHandler.

get_condition(*leadership_type*)
Gets the condition set on the leadership type specified.

get_condition_data(*leadership_type*, *info='all'*)
Uses the change data saved in the mock actions to instantiate the condition and permissions that will be created and get their info, to be used in forms

get_name()
Get name of community.

get_owner()
Communities own themselves by default, although subtypes may differ.

has_condition(*leadership_type*)
Returns True if community has a condition set on owner or on governor, depending on the leadership_type passed in.

has_governor_condition()
Returns True if community has a governor condition, False if not.

has_owner_condition()
Returns True if community has an owner condition, False if not.

class concord.communities.models.**Community**(**args*, ***kwargs*)
A community is, at heart, a collection of users. Communities govern resources that determine how these users interact, either moderating discussion spaces, like a community forum, setting restrictions on membership lists, or by setting access rules for resources owned by the community, such as saying only admins may edit data added to a dataset.

exception DoesNotExist

exception MultipleObjectsReturned

class concord.communities.models.**DefaultCommunity**(**args*, ***kwargs*)
Every user has a default community of which they are the BDFL. (They can theoretically give someone else power over their default community, but we should probably prevent that on the backend.)

We're almost always accessing this through the related_name. We have the user, and we want to know what community to stick our object in.

exception DoesNotExist

exception MultipleObjectsReturned

concord.communities.models.**create_default_community**(*sender*, *instance*, *created*, ***kwargs*)
Creates default community for a user when a new user is created.

State Changes

Community state changes.

class concord.communities.state_changes.**AddGovernorRoleStateChange**(*role_name*)
State change to add governor role to Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.communities.state_changes.**AddGovernorStateChange**(*governor_pk*)
State change to add governor to Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.communities.state_changes.**AddLeadershipConditionStateChange**(*, *condition_type*, *condition_data*, *permission_data*, *leadership_type*)
State change to add leadership condition to Community.

apply_actions_to_conditions(*action_list*, *target*)
Apply actions to condtions.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

generate_mock_actions(*actor*, *target*)
Helper method with template generation logic, since we're using it in both validate and implement.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

get_template_description(*mock_action_list*)
Get 'plain English' description of template.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.communities.state_changes.**AddMembersStateChange**(*member_pk_list*, *self_only=False*)
State change to add members to Community.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

classmethod get_configured_field_text(*configuration*)
Gets additional text for permissions item instance descriptions from configured fields.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.communities.state_changes.**AddOwnerRoleStateChange**(*role_name*)
State change to add owner role to Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.communities.state_changes.**AddOwnerStateChange**(*owner_pk*)
State change to add owner to Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.communities.state_changes.**AddPeopleToRoleStateChange**(*role_name*, *people_to_add*)
State change to add people to role in Community.

check_configuration(*action*, *permission*)
All configurations must pass for the configuration check to pass.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

classmethod get_uninstantiated_description(***configuration_kwargs*)
Takes in an arbitrary number of configuration kwargs and uses them to create a description. Does not reference fields passed on init.

implement(*actor*, *target*)
Method that carries out the change of state.

is_conditionally_foundational(*action*)
If role_name is owner or governor role, should should be treated as a conditional change.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.communities.state_changes.**AddRoleStateChange**(*role_name*)
State change to add role to Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.communities.state_changes.**ChangeNameStateChange**(*name*)
State change to change name of Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.communities.state_changes.**RemoveGovernorRoleStateChange**(*role_name*)
State change to remove governor role from Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.communities.state_changes.**RemoveGovernorStateChange**(*governor_pk*)
State change to remove governor from Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.communities.state_changes.**RemoveLeadershipConditionStateChange**(*, *leadership_type*)
State change to remove leadership condition from Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.communities.state_changes.**RemoveMembersStateChange**(*member_pk_list*, *self_only=False*)
State change to remove members from Community.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
If any of the members to be removed are an owner or governor (either directly, or through being in an owner or governor role) the action is not valid.

class concord.communities.state_changes.**RemoveOwnerRoleStateChange**(*role_name*)
State change to remove owner role from Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
If removing the owner role would leave the group with no owners, the action is invalid.

class concord.communities.state_changes.**RemoveOwnerStateChange**(*owner_pk*)
State change remove owner from Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
If removing the owner would leave the group with no owners, the action is invalid.

class concord.communities.state_changes.**RemovePeopleFromRoleStateChange**(*role_name*, *people_to_remove*)
State change to remove people from role in Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

is_conditionally_foundational(*action*)
If role_name is owner or governor role, should should be treated as a conditional change.

validate(*actor*, *target*)
When removing people from a role, we must check that doing so does not leave us without any owners.

class concord.communities.state_changes.**RemoveRoleStateChange**(*role_name*)
State change to remove role from Community.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

role_in_permissions(*permission*, *actor*)
Checks for role in permission and returns True if it exists. Checks in permissions which are nested on this permission as well.

validate(*actor*, *target*)
A role cannot be deleted without removing it from the permissions it's referenced in, and without removing it from owner and governor roles if it is there.

Client

Client for Community models.

class concord.communities.client.**CommunityClient**(*actor=None*, *target=None*)
The target of a community client, if a target is required, is always a community model. As with all Concord clients, a target must be set for all methods not explicitly grouped as target-less methods.

add_governor(*, *governor_pk: int*) → Tuple[int, Any]
Add governor to community.

add_governor_role(*, *governor_role: str*) → Tuple[int, Any]
Add governor role tocommunity.

add_leadership_condition(*, *condition_type*, *leadership_type*, *condition_data=None*, *permission_data=None*)
Add condition to leadership type (owners or governors).

add_members(*member_pk_list: list*) → Tuple[int, Any]
Add members to community.

add_owner(*, *owner_pk: int*) → Tuple[int, Any]
Add owner tocommunity.

add_owner_role(*, *owner_role: str*) → Tuple[int, Any]
Add owner role to community.

add_people_to_role(*, *role_name: str*, *people_to_add: list*) → Tuple[int, Any]
Add people to role in community.

add_role(*, *role_name: str*) → Tuple[int, Any]
Add role to community.

change_name(*, *new_name: str*) → Tuple[int, Any]
Change name of community.

community_model
alias of *concord.communities.models.Community*

create_community(*, *name: str*) → concord.communities.models.Community
Creates a Community (or class descended from Community model) with actor as creator. Creates some additional structures by default but this can be overriden with bare=True.

get_communities()
Gets all communities with the model type specified on the client.

get_communities_for_user(*user_pk*, *split=False*)
Given a supplied user_pk, gets all communities the associated user is a part of. If arg 'split' is true, separates communities the user is a leader of from those they're not a leader of.

get_community(*, *community_name: str = None*, *community_pk: str = None*) → concord.communities.models.Community
Takes either community name or PK and returns Community object. If both are supplied, returns Community object corresponding to community_pk.

get_condition_data(*leadership_type*, *info='all'*) → dict
Gets condition data for conditions set on owners and governors.

get_custom_roles()
Gets all custom roles set on the community.

get_governance_info_as_text()
Gets governance info about the community as text.

get_governorship_info(*shorten_roles=False*) → dict
Get the governors of the community.

get_members() → list
Gets all members of the community as list of user instances.

get_owner(*, *owned_object: django.db.models.base.Model*) → concord.communities.models.Community
Gets the owner of the owned object, which should always be a community.

get_ownership_info(*shorten_roles=False*) → dict
Get the owners of the community.

get_role_names()
Get just the role names set on a community (no info on who has the roles).

get_roles()
Get all roles set on the community.

get_users_given_role(*, *role_name: str*)
Given the role name, get the users who have that role.

has_foundational_authority(*, *actor*) → bool
Returns True if actor has foundational authority, otherwise False.

has_governing_authority(*, *actor*) → bool
Returns True if actor has governing authority, otherwise False.

has_role_in_community(*, *role: str*, *actor_pk: int*) → bool
Returns True if actor has specific role in community. otherwise False.

remove_governor(*, *governor_pk: int*) → Tuple[int, Any]
Remove governor from community.

remove_governor_role(*, *governor_role: str*) → Tuple[int, Any]
Remove governor role from community.

remove_leadership_condition(*, *leadership_type*)
Remove condition from leadership type (owners or governors).

remove_members(*member_pk_list: list*) → Tuple[int, Any]
Remove members from community.

remove_owner(*, *owner_pk: int*) → Tuple[int, Any]
Remove owner from community.

remove_owner_role(*, *owner_role: str*) → Tuple[int, Any]
Remove owner role from community.

remove_people_from_role(*, *role_name: str*, *people_to_remove: list*) → Tuple[int, Any]
Remove people from role in community.

remove_role(*, *role_name: str*) → Tuple[int, Any]
Remove role from community.

set_target(*target*)
Sets target of the client. Accepts either a target model or the target's pk and ct and fetches, in which case it fetches the model from the Database. Target must be a permissioned model.

set_target_community(*, *community_name: str = None*, *community_pk: str = None*)
Sets target community given a name or pk. If the user already has the community object, it can be set directly using the parent method set_target.

update_governors(*, *new_governor_data*)
Takes in a list of governors, adds those that are missing and removes those that are no longer there.

update_owners(*, *new_owner_data*)
Takes in a list of owners, adds those that are missing and removes those that are no longer there.

update_role_membership(*, *role_data*)
Takes in a list of roles with members, adds any missing members and adds any which are missing from community.

update_roles(*, *role_data*)
Takes in a list of roles and adds any which are missing from community.

Custom Fields

Defines customfields used in Community.

class concord.communities.customfields.**RoleField**(**args*, ***kwargs*)
This custom field allows us to access the methods and validation of the RoleHandler object.

db_type(*connection*)
Return the database column data type for this field, for the provided connection.

deconstruct()
Return enough information to recreate the field as a 4-tuple:

- The name of the field on the model, if contribute_to_class() has been run.
- The import path of the field, including the class:e.g. django.db.models.IntegerField This should be the most portable version, so less specific may be better.
- A list of positional arguments.
- A dict of keyword arguments.

Note that the positional or keyword arguments must contain values of the following types (including inner values of collection types):

- None, bool, str, int, float, complex, set, frozenset, list, tuple, dict
- UUID
- datetime.datetime (naive), datetime.date
- top-level classes, top-level functions - will be referenced by their full import path
- Storage instances - these have their own deconstruct() method

This is because the values here must be serialized into a text format (possibly new Python code, possibly JSON) and these are the only types with encoding handlers defined.

There's no need to return the exact way the field was instantiated this time, just ensure that the resulting field is the same - prefer keyword arguments over positional ones, and omit parameters with their default values.

get_prep_value(*value*)
Perform preliminary non-db specific value checks and conversions.

to_python(*value*)
Convert the input value into the expected Python data type, raising django.core.exceptions.ValidationError if the data can't be converted. Return the converted value. Subclasses should override this.

class concord.communities.customfields.**RoleHandler**(*, *members=None*, *owners=None*, *governors=None*, *custom_roles=None*)
Every community has a list of roles, which are set community-wide. People with the relevant permissions can add and remove roles, and add or remove people from roles.

There are three protected names which cannot be used: owners, governors, and members.

A person cannot be added as owner, governor or custom role unless they're already a member.

add_governor(*pk*)
Add governor given pk.

add_governor_role(*role_name*)
Add role as governor role given role_name.

add_member(*pk*)
Adds a member given member pk.

add_members(*pk_list*)
Adds a list of members given a list of pks.

add_owner(*pk*)
Add owner as governor given pk.

add_owner_role(*role_name*)
Add role as owner role given role_name.

add_people_to_role(*role_name*, *people_to_add*)
Add people to custom role. Protected roles are handled separately.

add_role(*role_name*)
Checks if role passed in already exists and, if not, adds to the list of roles. Roles that differ only by capitalization are not allowed.

get_custom_role_names()
Gets the names of custom roles only, but not protected roles.

get_custom_roles()
Gets custom roles only, but not protected roles.

get_governors(*actors_only=False*)
Gets all governors.

get_members()
Gets all members.

get_owners(*actors_only=False*)
Gets all owners.

get_role_names()
Gets names of all roles, including protected roles.

get_roles()
Gets all roles, including protected roles.

get_roles_given_user(*pk*)
Gets all roles in the group, given a user's pk.

Note that this doesn't catch when a user is owner/governor through custom roles.

get_users_given_role(*role_name*)
Gets all users with a given role.

has_governors()
Returns true if any governor is set on community, otherwise false.

has_specific_role(*role_name*, *pk*)
Checks whether a given user, specified by pk, has a role on the community, specified by role_name.

initialize_with_creator(*creator*)
To be valid, the RoleHandler must have at least one member and one owner. The most common use case is that the creator of a community is the only member and only owner upon initialization, which this help method allows us to accomplish. We also add the creator to the governors as well. Most commonly used after RoleHandler is initialized with no data passed in.

is_governor(*pk*)
Checks if user is an governor. Not a pure boolean since it's helpful to know which (if any) role matched for permission pipeline logging.

is_member(*pk*)
Returns True if pk passed in is member, False if not.

is_owner(*pk*)
Checks if user is an owner. Not a pure boolean since it's helpful to know which (if any) role matched for permission pipeline logging.

is_role(*role_name*)
Returns True if role_name passed in is a role on the community, False if not.

overwrite_roles(*role_dict*)
This method overwites existing roles and should only be used by the system, not the user.

remove_governor(*pk*)
Remove governor given pk.

remove_governor_role(*role_name*)
Remove role as governor role given role_name.

remove_member(*pk*)
Remove member given pk.

remove_members(*pk_list*)
Remove members given list of pks.

remove_owner(*pk*)
Remove owner given pk.

remove_owner_role(*role_name*)
Add role as owner role given role_name.

remove_people_from_role(*role_name*, *people_to_remove*)
Remove people from custom role. Protected roles are handled separately.

remove_role(*role_name*)
Removes the role passed in unless it is a protected role.

validate_custom_roles(*custom_roles*)
Custom roles dict should look like:

{ 'knights': [1,2,3], 'rooks': [2,3,4] }

All items in each list must correspond to members pks.

validate_governors(*governors*)
Governors dict should look like:

{ 'actors': [1,2,3], 'roles': ['knights', 'bishops', 'rooks'] }

All actors must be members and all roles must be a protected role or custom role.

validate_members(*members*)
Members should be a list of integers. For now, we do not check that these are real users.

validate_owners(*owners*)
Owners dict should look like:

{ 'actors': [1,2,3], 'roles': ['knights', 'bishops', 'rooks'] }

All actors must be members and all roles must be a protected role or custom role.

validate_role_handler()
Validates role handler by calling individual validate methods and verifying there's at least one owner.

concord.communities.customfields.**parse_role_handler_data**(*role_handler_data*)
Parse role handler data when de-serializing.

Utilities

3.2.8 Autodocumentation of the Conditionals module

Models

Condition models.

class concord.conditionals.models.**ApprovalCondition**(**args*, ***kwargs*)
Approval Condition class.

exception DoesNotExist

exception MultipleObjectsReturned

approve()
Approve a condition.

condition_status()
This method returns one of status 'approved', 'rejected', or 'waiting', after checking the condition for its unqiue status logic.

classmethod configurable_fields()
All conditions must supply their own version of the configurable_fields method, which should return a dict with field names as keys and field objects as values.

description_for_passing_condition(*fill_dict=None*)
This method returns a verbose, human-readable description of what will fulfill this condition. It optionally accepts permission data from the configured condition_template to be more precise about who can do what.

display_fields()
This method returns a list of fields and their values which can be shown to the user. Some overlap with get_configurable_fields_with_data since in many cases we're just showing the configured fields, but some data may be specific to the condition instance. Note that we do not, for now, return permission data.

display_status()
This method returns a more verbose, human-readable description of the condition status, after checking the condition for its unique status logic.

reject()
Reject a condition.

class concord.conditionals.models.**ConditionModel**(**args*, ***kwargs*)
Attributes:

action: integer representing the pk of the action that triggered the creation of this condition source_id : consists of a type and a pk, separated by a _, for example "**perm_**" + str(permission_pk) or

"**owner_**" + str(community_pk)

abstract condition_status()
This method returns one of status 'approved', 'rejected', or 'waiting', after checking the condition for its unqiue status logic.

abstract classmethod configurable_fields()
All conditions must supply their own version of the configurable_fields method, which should return a dict with field names as keys and field objects as values.

abstract description_for_passing_condition(*fill_dict*)
This method returns a verbose, human-readable description of what will fulfill this condition. It optionally accepts permission data from the configured condition_template to be more precise about who can do what.

abstract display_fields()
This method returns a list of fields and their values which can be shown to the user. Some overlap with get_configurable_fields_with_data since in many cases we're just showing the configured fields, but some data may be specific to the condition instance. Note that we do not, for now, return permission data.

abstract display_status()
This method returns a more verbose, human-readable description of the condition status, after checking the condition for its unique status logic.

get_action()
Get action associated with condition instance.

classmethod get_configurable_field_names()
Return field names as list.

classmethod get_configurable_fields()
Returns field values as list instead of dict

get_configurable_fields_with_data(*permission_data=None*)
Returns form_dict with condition data set as value.

get_display_string()
Get display text describing condition..

classmethod get_form_dict_for_field(*field*)
Get dictionary with form data for supplied field.

get_model_name()
Get name of condition model.

get_name()
Get name of condition.

initialize_condition(**args*, ***kwargs*)
Called when creating the condition, and passed condition_data and permission data.

user_condition_status(*user*)
User condition status is a shortcut which helps us determine if a user can take an action on a condition without actually creating an action. This is useful in determining what to show the user on the front-end. We assume that by the time user_condition_status is called the user has passed the permissions system, and so this status is to check instance-specific issues, like preventing a person who has already voted from voting again.

class concord.conditionals.models.**ConsensusCondition**(**args*, ***kwargs*)
Consensus Condition class.

exception DoesNotExist

exception MultipleObjectsReturned

condition_status()
This method returns one of status 'approved', 'rejected', or 'waiting', after checking the condition for its unqiue status logic.

classmethod configurable_fields()
Gets fields on condition which may be configured by user.

description_for_passing_condition(*fill_dict=None*)
Gets plain English description of what must be done to pass the condition.

display_fields()
Gets condition fields in form dict format, for displaying in the condition component.

display_status()
Gets 'plain English' display of status.

initialize_condition(*target*, *condition_data*, *permission_data*, *leadership_type*)
Called when creating the condition, and passed condition_data and permission data.

class concord.conditionals.models.**VoteCondition**(**args*, ***kwargs*)
Vote Condition class.

exception DoesNotExist

exception MultipleObjectsReturned

add_vote(*vote*)
Increments vote depending on vote type.

add_vote_record(*actor*)
Adds vote to record.

condition_status()
Gets status of condition.

classmethod configurable_fields()
Gets fields on condition which may be configured by user.

current_results()
Get the current results of the vote.

current_standing()
If voting ended right now, returns what the status would be.

describe_voting_period()
Gets human readable description of voting period.

description_for_passing_condition(*fill_dict=None*)
Gets plain English description of what must be done to pass the condition.

display_fields()
Gets condition fields in form dict format.

display_status()
Gets 'plain English' display of status.

get_timeout()
Get when condition closes, aka the voting deadline.

has_voted(*actor*)
Returns True if actor has voted, otherwise False.

user_condition_status(*user*)
Checks whether a user has voted already.

voting_deadline()
Gets deadline of vote given starting point and length of vote.

voting_time_remaining()
Gets time remaining to vote.

yeas_have_majority()
Helper method, returns True if yeas currently have majority.

yeas_have_plurality()
Helper method, returns True if yeas currently have plurality.

concord.conditionals.models.**conditionModel**
alias of *concord.conditionals.models.ConsensusCondition*

concord.conditionals.models.**retry_action**(*sender*, *instance*, *created*, ***kwargs*)
Signal handler which retries the corresponding action or action container when condition has been updated.

State Changes

State Changes for conditional models

class concord.conditionals.state_changes.**AddVoteStateChange**(*vote*)
State change for adding a vote.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
To validate the vote, we need to check that: a) the voter hasn't voted before b) if the vote is abstain, abstentions are allowed

class concord.conditionals.state_changes.**ApproveStateChange**
State change for approving a condition.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.conditionals.state_changes.**RejectStateChange**
State change for rejecting a condition..

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Checks if actor is the same user who sent the action that triggered the condition and, unless self approval is allowed, rejects them as invalid.

class concord.conditionals.state_changes.**ResolveConsensusStateChange**
State change for resolving a consensus condition.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Checks that the actor is a participant.

class concord.conditionals.state_changes.**RespondConsensusStateChange**(*response*)
State change for responding to a consensus condition

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Checks that the actor is a participant.

class concord.conditionals.state_changes.**SetConditionOnActionStateChange**(*, *condition_type*, *condition_data=None*, *permission_data=None*, *permission_pk=None*, *community_pk=None*, *leadership_type=None*)
State change which actually creates a condition item associated with a specific action. I'm not actually 100% sure this should be a state change, since as far as I can tell this will always be triggered by the system internally, but we're doing it this way for now. Also not sure if this should be split up into permission condition and leadership condition.

generate_source_id()
Generates a source_id to use when creating condition item.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

get_condition_class()
Gets the condition class object given the condition type.

get_condition_verb()
Get the verb of the associated condition.

get_owner()
The owner of the condition should be the community in which it is created. For now, this means looking up permission and getting owner, or using community if community is set.

Note that if multiple community models are being used, and the community pk passed in is not the primary/default model, this will break.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

Client

Clients for conditionals.

class concord.conditionals.client.**ApprovalConditionClient**(*actor=None*, *target=None*)
The target of the ApprovalConditionClient must always be an ApprovalCondition instance.

approve() → Tuple[int, Any]
Approve the target condition.

reject() → Tuple[int, Any]
Reject the taret condition.

class concord.conditionals.client.**ConditionalClient**(*actor=None*, *target=None*)
ConditionalClient is largely used as an easy way to access all the specific conditionclients at once, but can also has some helper methods and one state change - add_condition_to_action.

get_condition_as_client(*, *condition_type: str*, *pk: int*) → concord.actions.client.BaseClient
Given a condition type and pk, gets that condition object and returns it wrapped in a client. Note: condition type MUST be capitalized to match the client name.

get_condition_class(*, *condition_type*)
Get condition class object given condition type.

get_condition_item(*, *condition_pk*, *condition_type*)
Get condition item given pk and type.

get_condition_item_given_action_and_source(*, *action_pk: int*, *source_id: str*) → django.db.models.base.Model
Given the action_pk and source_id corresponding to a condition item, get that item.

get_condition_item_on_community(*, *action_pk: int*, *community_pk: int*, *leadership_type: str*)
Get condition item on an action given the community & leadership type that triggered it.

get_condition_item_on_permission(*, *action_pk: int*, *permission_pk: int*)
Get condition item corresponding to a specific action and permission.

get_condition_items_for_action(*, *action_pk*)
Get all condition items set on an action.

get_or_create_condition_on_community(*action*, *community*, *leadership_type*)
Given an action and community & leadership type, if a condition item exists get it, otherwise create it.

get_or_create_condition_on_permission(*action*, *permission*)
Given an action and permission, if a condition item exists get it, otherwise create it.

get_possible_conditions()
Get all possible conditions.

set_condition_on_action(*condition_type*, *condition_data=None*, *permission_pk=None*, *community_pk=None*, *leadership_type=None*, *permission_data=None*)
This is almost always created as a mock to be used in a condition TemplateField. Typically when creating the mock we want to supply condition_type and condition_data but leave the rest to be supplied later. When the action is actually run, the target should *always* be an action!

trigger_condition_creation(*, *action*, *permission=None*, *community=None*, *leadership_type=None*)
Create a condition item given action and corresponding info about the condition that triggered it.

trigger_condition_creation_from_source_id(*, *action*, *source_id*)
Trigger condition creation given an action and the source_id that triggered it.

class concord.conditionals.client.**ConsensusConditionClient**(*actor=None*, *target=None*)
The target of the ConsensusConditionClient must always be a ConsensusCondition instance.

get_current_results() → Dict
Gets current results of vote condition.

resolve() → Tuple[int, Any]
Resolve consensus condition.

resolveable() → tuple
Returns True if condition is ready to resolve, or False if not. If not, returns time until it will be ready to resolve.

respond(*, *response: str*) → Tuple[int, Any]
Add response to consensus condition.

class concord.conditionals.client.**VoteConditionClient**(*actor=None*, *target=None*)
The target of the VoteConditionClient must always be a VoteCondition instance.

can_abstain() → bool
Returns True if users can abstain, otherwise False.

get_current_results() → Dict
Gets current results of vote condition.

publicize_votes() → bool
Returns True if condition is set to publicize votes, otherwise False.

vote(*, *vote: str*) → Tuple[int, Any]
Add vote to condition.

3.2.9 Autodocumentation of the Permissions (permission_resources) module

Models

Permission Resource models.

class concord.permission_resources.models.**PermissionsItem**(**args*, ***kwargs*)

Permission items contain data for who may change the state of the linked object in a given way.

content_type, object_id, permitted object -> specify what object the permission is set on change_type -> specifies what action the permission covers

actors -> individually listed people roles -> reference to roles specified in community

If someone matches an actor OR a role they have the permission. actors are checked first.

exception DoesNotExist

exception MultipleObjectsReturned

actor_in_permission(*actor*)

Check if actor is in the permission, either as individual or as role. Returns Boolean plus the role if in permission via role, otherwise returns None.

add_role_to_permission(*, *role: str*)

Add a role to the permission.

change_display_string()

Helper method for displaying the change element of permissions.

display_string()

Helper method for displaying permissions.

get_actors(*as_instances=False*)

Get the actors that have the permission.

get_all_context_keys()

Helper method to get context keys for the change type.

get_change_fields()

Helper method used to get a list of required fields for init of this permission's change type.

get_change_type()

Get the ChangeType (short version) of the permission.

get_condition_data(*info='all'*)

Uses the change data saved in the mock actions to instantiate the condition and permissions that will be created and get their info, to be used in forms

get_configuration()

Get the configuration of the permissoin.

get_configured_field_data()

Get the field data corresponding to the configuration.

get_name()

Get permission name.

get_nested_objects()

Gets objects that the model is nested within.

Nested objects are often things like the owner of instance or, for example, a forum that a post is posted within.

Called by the permissions pipeline in *permissions.py*.

get_permitted_object()
Gets the permitted object, that is the permissioned model that the permission is set on.

get_roles()
Get the roles that have the permission.

get_state_change_object()
Get the state change object associated with the change_type of the permission.

has_condition()
Returns True if PermissionsItem has condition, otherwise False.

has_role(*, *role: str*)
Check if a given role has the permission.

match_actor(*actor_pk*)
Determines if actor in the permission. If inverse is toggled, returns the oppposite - such that they would NOT match if they're listed in an inverse permission.

match_change_type(*change_type*)
Checks if the given change type matches the PermissionItem's change_type.

remove_role_from_permission(*, *role: str*)
Remove a role from the permission.

set_configuration(*configuration_dict*)
Set the configuration of the permission.

set_fields(*, *owner=None*, *permitted_object=None*, *anyone=None*, *change_type=None*, *inverse=None*, *actors=None*, *roles=None*, *configuration=None*)
Helper method to make it easier to save permissions fields in the format our model expects.

concord.permission_resources.models.**delete_empty_permission**(*sender*, *instance*, *created*, ***kwargs*)
Toggle is_active so it is only true when there are actors or roles set on the permission.

State Changes

Get state changes for permissions resources.

class concord.permission_resources.state_changes.**AddActorToPermissionStateChange**(*, *actor_to_add: str*)
State change to add an actor to a permission.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.permission_resources.state_changes.**AddPermissionConditionStateChange**(*, *condition_type*, *condition_data*, *permission_data*)

State change to add a condition to a permission.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

generate_mock_actions(*actor*, *permission*)
Helper method with template generation logic, since we're using it in both validate and implement. The actions below are stored within the template, and copied+instantiated when a separate action triggers the permission to do so.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.permission_resources.state_changes.**AddPermissionStateChange**(*change_type*, *actors*, *roles*, *configuration=None*, *anyone=False*, *inverse=False*)

State change to add a permission to something.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

implement(*actor*, *target*)
Method that carries out the change of state.

input_target
alias of *concord.permission_resources.models.PermissionsItem*

is_conditionally_foundational(*action*)
Some state changes are only foundational in certain conditions. Those state changes override this method to apply logic and determine whether a specific instance is foundational or not.

validate(*actor*, *target*)
We need to check configuration of permission is valid. Also need to check that the given permission can be set on the target.

class concord.permission_resources.state_changes.**AddRoleToPermissionStateChange**(*, *role_name: str*)
State change to add a role to a permission.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.permission_resources.state_changes.**ChangeInverseStateChange**(*, *change_to: bool*)
State change to toggle the inverse field of a permission.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable

targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.permission_resources.state_changes.**ChangePermissionConfigurationStateChange**(

State change to change the configuration of a permission.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.permission_resources.state_changes.**DisableAnyoneStateChange**
State change which takes a permission that has 'anyone' enabled, so anyone can take it, and disables it so only the roles and actors specified can take it..

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.permission_resources.state_changes.**EditTemplateStateChange**(*template_object_id*, *field_name*, *new_field_data*)

State change to edit a template.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.permission_resources.state_changes.**EnableAnyoneStateChange**
State change to set a permission so anyone can take it.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.permission_resources.state_changes.**RemoveActorFromPermissionStateChange**(*, *actor_to_*, *str*)

State change to remove an actor from a permission.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable

targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.permission_resources.state_changes.**RemovePermissionConditionStateChange**
State change to remove a condition from a permission.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.permission_resources.state_changes.**RemovePermissionStateChange**
State change to remove a permission from something.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.permission_resources.state_changes.**RemoveRoleFromPermissionStateChange**(*, *role_na*
str)
State change to remove a role from a permission.

check_configuration(*action*, *permission*)
All configurations must pass for the configuration check to pass.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

classmethod get_uninstantiated_description(**configuration_kwargs*)
Takes in an arbitrary number of configuration kwargs and uses them to create a description. Does not reference fields passed on init.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

Client

Client for permissions

class concord.permission_resources.client.**PermissionResourceClient**(*actor=None*, *target=None*)
Client for interacting with Permission resources. Target is usually the PermissionedModel that we're setting permissions on, but is occasionally the PermissionedModel itself.

actor_satisfies_permission(*, *actor*, *permission: concord.permission_resources.models.PermissionsItem*) → bool
Returns True if given actor satisfies given permission.

add_actor_to_permission(*, *actor: str*) → Tuple[int, Any]
Add actor to permission.

add_condition_to_permission(*, *condition_type*, *condition_data=None*, *permission_data=None*)
Add a condition to the permission.

add_permission(*, *permission_type: str*, *permission_actors: list = None*, *permission_roles: list = None*, *permission_configuration: dict = None*, *anyone=False*) → Tuple[int, Any]
Add permission to target.

add_role_to_permission(*, *role_name: str*) → Tuple[int, Any]
Add role to permission.

change_configuration_of_permission(*, *configurable_field_name: str*, *configurable_field_value: str*) → Tuple[int, Any]
Change the configuration of the permission.

change_inverse_field_of_permission(*, *change_to: bool*) → Tuple[int, Any]
Toggle the inverse field on the permission.

get_all_permissions() → concord.permission_resources.models.PermissionsItem
Get all permissions on the client target.

get_all_permissions_in_db()
Gets all permissions in the DB. We should swap this out with getting all permissions in a group plus all of its owned objects but for now, this is what we have.

get_condition_data(*info='all'*) → dict
Get condition data on the target.

get_permission(*, *pk: int*) → concord.permission_resources.models.PermissionsItem
Gets permissions item given pk.

get_permissions_associated_with_actor(*actor: int*) → List[concord.permission_resources.models.PermissionsItem]
Given an actor, get all permissions on the target they are listed as an individual actor within.

get_permissions_associated_with_role_for_target(*, *role_name: str*) → List[concord.permission_resources.models.PermissionsIt
Get any permissions on the target associated with the given role.

get_permissions_for_role(*, *role_name*)
Given a role, get all permissions associated with it.

get_permissions_on_object(*, *target_object: django.db.models.base.Model*) → concord.permission_resources.models.PermissionsItem
Given a target object, get all permissions set on it.

get_roles_associated_with_permission(*, *permission_pk: int*)
Given a permission, get all roles set on it.

get_settable_permissions(*return_format='tuples'*) → List[Tuple[str, str]]
Gets a list of permissions it is possible to set on the target, in various formats

get_settable_permissions_for_model(*model_class*)
Given a model class (or, optionally, an instance of a model class), gets the state change objects which may be set on that model via a permission.

get_specific_permissions(*, *change_type: str*) → concord.permission_resources.models.PermissionsItem
Get all permissions on the client target matching the given change_type.

give_anyone_permission()
Make it so everyone has the permission.

has_permission(*client*, *method_name*, *parameters*, *exclude_conditional=False*)
Checks results of running a given (mock) action through the permissions pipeline. Note that this says nothing about whether the given action is valid, as the validate step is called separately.

remove_actor_from_permission(*, *actor: str*) → Tuple[int, Any]
Remove actor from permission.

remove_anyone_from_permission()
Remove the ability for everyone to have the permission.

remove_condition_from_permission()
Remove a condition from the permission.

remove_permission() → Tuple[int, Any]
Remove permission from target.

remove_role_from_permission(*, *role_name: str*) → Tuple[int, Any]
Remove role from permission.

update_actors_on_permission(*, *actor_data*, *permission*)
Given a list of actors, updates the given permission to match those actors.

update_configuration(*, *configuration_dict: dict*, *permission*)
Given a dict with the new configuration for a permission, change individual fields as needed.

update_roles_on_permission(*, *role_data*, *permission*)
Given a list of roles, updates the given permission to match those roles.

Custom Fields

Custom Fields used by Permission Resource models.

class concord.permission_resources.customfields.**ActorList**(*actor_list=None*)
This custom object allows us to preferentially manipulate our actors as a list of PKs, so we don't have to constantly query the DB. If pks and instances are not identical, the pk_list is assumed authoritative.

actor_in_list(*actor*)
Checks if a given actor is in the actor list.

add_actors(*actors*)
If actors are User instances, add to instance_list and pk_list; if pks, add only to pk_list.

as_instances()
Get actors as instances, checking first to make sure the list of instances is the same length as pks.

as_pks()
Get actors as pk.

is_empty()
Returns true if there's no actors in lists.

lists_are_equivalent()
Helper method to check that pk_list and instance_list are equivalent.

remove_actors(*actors*, *strict=True*)
If actors are user instances, remove from instance_list and pk_list; if pks, remove only from pk_list. If strict is true, all actors to be removed MUST be in the pk_list.

class concord.permission_resources.customfields.**ActorListField**(**args*, ***kwargs*)
This custom field allows us to access a list of user objects or a list of user pks, depending on our needs.

db_type(*connection*)
Return the database column data type for this field, for the provided connection.

deconstruct()
Return enough information to recreate the field as a 4-tuple:

- The name of the field on the model, if contribute_to_class() has been run.
- The import path of the field, including the class:e.g. django.db.models.IntegerField This should be the most portable version, so less specific may be better.
- A list of positional arguments.
- A dict of keyword arguments.

Note that the positional or keyword arguments must contain values of the following types (including inner values of collection types):

- None, bool, str, int, float, complex, set, frozenset, list, tuple, dict

- UUID
- datetime.datetime (naive), datetime.date
- top-level classes, top-level functions - will be referenced by their full import path
- Storage instances - these have their own deconstruct() method

This is because the values here must be serialized into a text format (possibly new Python code, possibly JSON) and these are the only types with encoding handlers defined.

There's no need to return the exact way the field was instantiated this time, just ensure that the resulting field is the same - prefer keyword arguments over positional ones, and omit parameters with their default values.

get_prep_value(*value*)
Perform preliminary non-db specific value checks and conversions.

to_python(*value*)
Convert the input value into the expected Python data type, raising django.core.exceptions.ValidationError if the data can't be converted. Return the converted value. Subclasses should override this.

class concord.permission_resources.customfields.**RoleList**(*role_list=None*)
Helper object to manage roles set in permission.

add_roles(*role_list*)
Add roles to role_list.

get_roles()
Gets role list.

is_empty()
Returns True if role_list is empty.

remove_roles(*role_list*)
Remove roles from role_list.

role_name_in_list(*role_name*)
Returns true if given role_name in role list.

class concord.permission_resources.customfields.**RoleListField**(**args*, ***kwargs*)
This custom field allows us to access our list of role pairs.

db_type(*connection*)
Return the database column data type for this field, for the provided connection.

deconstruct()
Return enough information to recreate the field as a 4-tuple:

- The name of the field on the model, if contribute_to_class() has been run.
- The import path of the field, including the class:e.g. django.db.models.IntegerField This should be the most portable version, so less specific may be better.
- A list of positional arguments.
- A dict of keyword arguments.

Note that the positional or keyword arguments must contain values of the following types (including inner values of collection types):

- None, bool, str, int, float, complex, set, frozenset, list, tuple, dict
- UUID
- datetime.datetime (naive), datetime.date

- top-level classes, top-level functions - will be referenced by their full import path
- Storage instances - these have their own deconstruct() method

This is because the values here must be serialized into a text format (possibly new Python code, possibly JSON) and these are the only types with encoding handlers defined.

There's no need to return the exact way the field was instantiated this time, just ensure that the resulting field is the same - prefer keyword arguments over positional ones, and omit parameters with their default values.

get_prep_value(*value*)
Perform preliminary non-db specific value checks and conversions.

to_python(*value*)
Convert the input value into the expected Python data type, raising django.core.exceptions.ValidationError if the data can't be converted. Return the converted value. Subclasses should override this.

concord.permission_resources.customfields.**parse_actor_list_string**(*actor_list_string*)
Helps deserialize actor list.

concord.permission_resources.customfields.**parse_role_list_string**(*role_list_string*)
Helper method to deserialize role list object.

Utilities

Permission Resource utilities.

concord.permission_resources.utils.**check_configuration**(*action*, *permission*)
Given a permission, check whether the action matches the configuration.

concord.permission_resources.utils.**delete_permissions_on_target**(*target*)
Given a target PermissionedModel object, find all permissions set on it and delete them.

concord.permission_resources.utils.**get_settable_permissions**(*, *target*)
Gets a list of all permissions that may be set on the model.

3.2.10 Autodocumentation of the Resources module

Models

Resource models.

class concord.resources.models.**Comment**(**args*, ***kwargs*)
Comment model.

exception DoesNotExist

exception MultipleObjectsReturned

get_name()
Gets name of Model. By default, gets string representation.

class concord.resources.models.**CommentCatcher**(**args*, ***kwargs*)
The comment catcher model is a hack to deal with leaving comments on non-permissioned models. Right now, the only model we're doing this for is Action.

exception DoesNotExist

exception MultipleObjectsReturned

get_name()
Get name of object.

class concord.resources.models.**Item**(**args*, ***kwargs*)
Simple item model.

Will eventually be removed when more usable resource is added.

exception DoesNotExist

exception MultipleObjectsReturned

get_name()
Get name of item.

class concord.resources.models.**Resource**(**args*, ***kwargs*)
Simple resource model.

Will eventually be removed when a more usable resource is added.

exception DoesNotExist

exception MultipleObjectsReturned

get_items() → List[str]
Gets item associated with resource.

get_name()
Gets name of abstract resource.

get_nested_objects()
Get objects that Resource is nested on, in this case the owner.

class concord.resources.models.**SimpleList**(**args*, ***kwargs*)
Model to store simple lists with arbitrary fields.

exception DoesNotExist

exception MultipleObjectsReturned

add_row(*row*, *index=None*)
Add a row to the list.

adjust_rows_to_new_configuration(*configuration*)
Given a new row configuration, goes through existing rows and adjusts them them.

check_row_against_configuration(*row*)
Given a row, check that it's valid for the row configuration.

delete_row(*index*)
Delete a row from the list.

edit_row(*row*, *index*)
Edit a row in the list.

get_name()
Get name of item.

get_nested_objects()
Get models that permissions for this model might be set on.

get_row_configuration()
Gets row configuation json and loads to Python dict.

get_rows()
Get the rows in the list.

handle_missing_fields_and_values(*row*)
Given a row, check that it's valid for the row configuration.

move_row(*old_index*, *new_index*)
Moves a row from old index to new index.

set_row_configuration(*row_configuration*)
Given a row configuration with format, validated and saves to DB.

validate_configuration(*row_configuration*)
Checks that a given configuration is valid. Should have format:

{ field_name : { 'required': True, 'default_value': 'default'}}

If required is not supplied, defaults to False. If default_value is not supplied, defaults to None.

State Changes

Resource State Changes

class concord.resources.state_changes.**AddCommentStateChange**(*text*, *original_creator_only=False*)
State Change to add a comment.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

implement(*actor*, *target*)
Method that carries out the change of state.

input_target
alias of *concord.resources.models.Comment*

class concord.resources.state_changes.**AddItemStateChange**(*name*)
State Change to add item to a resource.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
An AddItem permission can be set on an item, a resource, or on the community that owns the item or resource.

implement(*actor*, *target*)
Method that carries out the change of state.

input_target
alias of *concord.resources.models.Item*

class concord.resources.state_changes.**AddListStateChange**(*name*, *configuration*, *description=None*)
State Change to create a list in a community (or other target).

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

implement(*actor*, *target*)
Method that carries out the change of state.

input_target
alias of *concord.resources.models.SimpleList*

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.resources.state_changes.**AddRowStateChange**(*row_content*, *index=None*)
State Change to add a row to a list.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.resources.state_changes.**ChangeResourceNameStateChange**(*name*)
State Change to change a resource name.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.resources.state_changes.**DeleteCommentStateChange**(*commenter_only=False*, *original_creator_only=False*)
State Change to delete a comment.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

classmethod get_configured_field_text(*configuration*)
Gets additional text for permissions item instance descriptions from configured fields.

get_context_instances(*action*)
Returns the comment and the commented object. Also returns the commented object by its model name, to handle cases where the referer knows the model type vs doesn't know the model type.

classmethod get_settable_classes()
Comments may be made on any target - it's up to the front end to decide what comment functionality to expose to the user.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.resources.state_changes.**DeleteListStateChange**
State Change to delete an existing list.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.resources.state_changes.**DeleteRowStateChange**(*index*)
State Change to delete a row in a list.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.resources.state_changes.**EditCommentStateChange**(*text*, *commenter_only=False*, *original_creator_only=False*)
State Change to edit a comment.

classmethod check_configuration_is_valid(*configuration*)
Used primarily when setting permissions, this method checks that the supplied configuration is a valid one. By contrast, check_configuration checks a specific action against an already-validated configuration.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_configurable_fields()
Gets the fields of a change object which may be configured when used in a Permission model.

classmethod get_configured_field_text(*configuration*)
Gets additional text for permissions item instance descriptions from configured fields.

get_context_instances(*action*)
Returns the comment and the commented object. Also returns the commented object by its model name, to handle cases where the referer knows the model type vs doesn't know the model type.

classmethod get_settable_classes()
Comments may be made on any target - it's up to the front end to decide what comment functionality to expose to the user.

implement(*actor*, *target*)
Method that carries out the change of state.

class concord.resources.state_changes.**EditListStateChange**(*name=None*, *configuration=None*, *description=None*)

State Change to edit an existing list.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.resources.state_changes.**EditRowStateChange**(*row_content*, *index*)

State Change to edit a row in a list.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.resources.state_changes.**MoveRowStateChange**(*old_index*, *new_index*)

State Change to move a row in a list.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

validate(*actor*, *target*)
Method to check whether the data provided to a change object in an action is valid for the change object. Optional exclude_fields tells us not to validate the given field.

class concord.resources.state_changes.**RemoveItemStateChange**
State Change to remove item from a resource.

description_past_tense()
Returns the description of the state change object, in past tense.

description_present_tense()
Returns the description of the state change object, in present tense.

classmethod get_allowable_targets()
Returns the classes that an action of this type may target.

classmethod get_settable_classes()
Returns the classes that a permission with this change type may be set on. This overlaps with allowable targets, but also includes classes that allowable targets may be nested on. Most likely called by the validate method in AddPermissionStateChange.

implement(*actor*, *target*)
Method that carries out the change of state.

Client

Client for Resources.

class concord.resources.client.**CommentClient**(*actor=None*, *target=None*)
Client for interacting with Comment model.

add_comment(*text*)
Add a comment to the target.

delete_comment()
Delete a comment from the target.

edit_comment(*text*)
Edit a comment on the target.

get_all_comments_on_target()
Gets all comment son the current target.

get_comment(*pk*)
Gets specific comment given pk.

swap_target_if_needed(*create=False*)
The target of CommentClient needs to be the CommentCatcher object, but sometimes the target is set to action instead. We automatically handle that instead of making the user do it.

class concord.resources.client.**ListClient**(*actor=None*, *target=None*)
Client for interacting with Lists.

class concord.resources.client.**ResourceClient**(*actor=None*, *target=None*)

The target of a resource client, if a target is required, is always a resource model. As with all Concord clients, a target must be set for all methods not explicitly grouped as target-less methods.

add_item(*, *item_name: str*) → Tuple[int, Any]

Add item to resource.

change_name(*, *new_name: str*) → Tuple[int, Any]

Change name of resource.

create_resource(*, *name: str*) → concord.resources.models.Resource

Create a resource given name of resource to be created.

get_all_resources() → django.db.models.query.QuerySet

Get all resources in the system.

get_items_on_resource() → List[str]

Get items on targtet resource.

get_resource_given_name(*, *resource_name: str*) → django.db.models.query.QuerySet

Get a resource given a unique name.

get_resource_given_pk(*, *pk: int*) → django.db.models.query.QuerySet

Get a resource given pk.

remove_item() → Tuple[int, Any]

Remove item from resource.

3.2.11 Index

PYTHON MODULE INDEX

C

INDEX

A

B

C

D

E

F

G

H

I

S

T

U

V

Y