Code Review and Design Document

Hello Anshu!

This document contains everything you need to complete this work sample section of the interview process. Rather than ask you to code, we’re asking you to do two coding related tasks:

1. Review a Pull Request that adds a feature to our models.
2. Write up a document to help us build a feature for our application.

If you have any questions at all, please contact [ethan@informedk12.com](mailto:ethan@informedk12.com)

This work shouldn’t take more than a couple of hours. We want to respect your time as much as possible while also getting a sense of how you’ll do with the things that you’ll be asked to do very well here. Teaching, planning, and written communication are almost as important as coding in the role we’re looking for you to fill here.

Thank you so much for your time!

## Part 1: Code Review

The goal for this part of the work is to review a branch for a new feature. Envision yourself in a mentorship role when reviewing this branch. You can assume that you have a great relationship with the person who submitted this code, and they are looking to learn from you in this review. However, normal code review assumptions can be ignored here, for example, if you have suggestions for larger refactors/overall comments that would have typically come up in discussions before the code review or you’d follow up with in person, please talk about them here.

The commit message and the code is below. Please add comments prepended with a # to share your thoughts on the code. If you want to edit the code, or add more expressive comments/ideas/concerns in the text (which is totally fine!), make sure your work is clearly visible.

### Commit Message:

Add orderable module for sorting on the server

Add a module to allow constructing a custom sort query.

E.g. we need to sort forms by statuses like

order = 'submitted in\ progress viewed unopened bounced'

FormParent.forms.order\_as\_specified(status: %w(order)).limit(10000000000)

### Orderable.rb

module Orderable

#direction should be taken as a separate param here and handled within *order\_as\_specified #* and not within *extract\_options*.

# something like the following

# def order\_as\_specified(direction: 'asc', \*\*options)

# attribute, values = extract\_options(options)

# direction = direction.to\_s.downcase

# unless %w(desc asc).include? direction

# message = "Invalid direction: #{direction}. Only 'asc' or 'desc' allowed."

# fail ArgumentError, message

# end

# query = values.map do |value|

# "#{table\_name}.#{attribute}='#{value} #{direction}' NULLS LAST"

# end.join(', ')

# order(query)

# end

# then we don’t need to handle direction at all in *extract\_options*  
#

def order\_as\_specified(options)

attribute, values = extract\_options(options)

# there should be a validation for attribute being present in table

query = values.map do |value|

"#{table\_name}.#{attribute}='#{value}' DESC NULLS LAST"

end.join(', ')

order(query)

end

def extract\_options(direction: 'asc', \*\*options)

fail ArgumentError, 'Invalid params' unless options.size == 1 # Could you write a more descriptive error message here

direction = direction.to\_s.downcase

unless %w(desc asc).include? direction

message = "Invalid direction: #{direction}. Only 'asc' or 'desc' allowed."

fail ArgumentError, message

end

attribute, values = options.first

values = values.reverse if direction == 'desc'

[attribute, values]

end

end

### 

### order\_as\_specified\_spec.rb

RSpec.describe Orderable, '.order\_as\_specified' do

with\_model :OrderableModel do

table do |t|

t.string :status

end

model do

extend Orderable

end

end

let!(:raw\_model) { OrderableModel.create!(status: 'raw') }

let!(:cooked\_model) { OrderableModel.create!(status: 'cooked') }

let!(:burnt\_model) { OrderableModel.create!(status: 'burnt') }

let!(:gone\_model) { OrderableModel.create!(status: 'eaten') }

let!(:model) { OrderableModel.create! }

it 'returns results in specified order' do

result = OrderableModel.order\_as\_specified(status: %w(cooked burnt raw))

expectation = [cooked\_model, burnt\_model, raw\_model, gone\_model, model]

expect(result).to eq expectation

result = OrderableModel.order\_as\_specified(status: %w(burnt raw cooked))

expectation = [burnt\_model, raw\_model, cooked\_model, gone\_model, model]

expect(result).to eq expectation

end

it 'returns results in reverse order' do

order\_options = { status: %w(burnt raw cooked), direction: 'desc' }

result = OrderableModel.order\_as\_specified(order\_options)

expectation = [cooked\_model, raw\_model, burnt\_model, gone\_model, model]

expect(result).to eq expectation

end

it 'returns results in ascending order' do

order\_options = { status: %w(burnt raw cooked), direction: 'asc' }

result = OrderableModel.order\_as\_specified(order\_options)

expected = [burnt\_model, raw\_model, cooked\_model, gone\_model, model]

expect(result).to eq expected

end

it 'returns results with a limit' do

order\_options = { status: %w(cooked burnt raw) }

result = OrderableModel.order\_as\_specified(order\_options).limit(2)

expect(result).to eq [cooked\_model, burnt\_model]

end

it 'throws an ArgumentError when given wrong number of options' do

expect do

OrderableModel.order\_as\_specified(status: 'foo', bad: 'bar')

end.to raise\_error(ArgumentError, /Invalid params/)

end

it 'throws an ArgumentError when direction is invalid' do

expect do

OrderableModel.order\_as\_specified(status: %w(cooked), direction: 'up!')

end.to raise\_error(ArgumentError, /Invalid direction/)

end

end

## Part 2: Design / Architecture

The goal for this part of the work is to describe in detail how you would build a “Route Builder”

What is a Route Builder? Good, question, it's an Informed K12 thing :\_) In the big picture, we automate the path that paperwork takes through a school. Each stop along the way is called a **step**. And steps are assigned to a person (a specific person, or someone from a list of people, or an initially unspecified person). Schools are tricky places, so we’re built for maximum flexibility. This UI comes from our application, and is the place where you would define the **route** that this specific **form** would take around the building. You can see a video of how the page works here: [**Video of interactions**](https://drive.google.com/file/d/1MSQ-wnX1z8gosvs0ZlTY5gctApalRwDO/view?usp=sharing)

Looking at the page, the first step is the **initiator step**, it's the person who has grabbed the paperwork and started filing things (because they have an expense to be repaid, equipment to request, a role to hire for, etc). The remaining steps can be attached to one or more people (or left as undefined!). Schools need multiple people listed on a step because often there are people with similar roles in different departments. Rather than create a form for each department (it gets really crazy really quickly), we want to let the people filling out the form control its path. So in the example of an expense report, a Spanish teacher might initiate the form, and then direct the form to their manager. Which would be a different person than if the form was being initiated by someone in the school’s IT department. So the route of the form might be defined as:

1. Initiator enters their information into the form
2. Initiator chooses who to send for approval:
   1. Spanish Expense Mgr / sarah@school.example.com
   2. IT Expense Mgr / qian@school.example.com
3. Sarah or Qian chooses who to send for approval:
   1. Teacher Budget / jen@school.example.com
   2. Site Budget / stef@school.example.com
4. Jen or Stef sends it to:
   1. Superintendent / bridget@school.example.com

So, this UI allows the school to build the route the form should take by specifying

how many steps there should be and who should be available as someone who can sign

off (approve) on that step.

Features of the page:

* Clicking the “Add step” link adds another step.
* Clicking the gear icon on a step loads a panel that allows adding multiple recipients. Clicking the gear again hides this panel.
* Clicking the “Add recipient” link renders new name/email inputs
* When a recipient name is put in and no email, show an error message (invalid case).
* When an email is put in but no recipient name, show an error message (invalid case).
* When an email is put in and is invalid, show an error message (invalid case).
* When both name and email are empty, show no error message (valid case).
* When there is at least one correctly filled out recipient (with name and email) and no invalid recipients, the “continue” button is enabled.
* If there are any invalid recipients, the “continue” button is disabled.

The work to do is to write a document that illustrates your approach (particularly the “whys”) to building this feature that you could then discuss with the team before implementing. Use common web development patterns, if you want to do it all in JavaScript, a mix of templates and JavaScript, or entirely from the server, it’s up to you. We’re looking for something that would be relatively easy to hand off to another developer for implementation

### Part 2: Your design document

[your description goes here]

Tables Needed

**Forms**

id

form\_type ( can be a dropdown or a text field)

initiator\_emp\_id ( foreign key \_ employees)

Status: [‘started’, ‘created’, ‘completed’]

Created\_at

Updated\_at

**Steps**

Id

step\_num

label

Form\_id ( foreign key of forms)

**Approvers**

id

Steps\_id ( foreign\_key steps table)

Employee\_id

Can\_add\_next\_approver: (True/ False)

**Tables Assumed already exist:**  
1. Employee ( containing employee id, name, position, email + other info not pertinent to this feature)

API endpoints needed at the backend

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Event** | **Resource** | **Method** | **URI** | **body** | **header** | **response** | **error message** |
| Begin Form' button is clicked | Form | P  OST | /forms | {form\_type, initiator\_emp\_id} |  | returns id of form created | "employee id is invalid" |
|  | Step | POST | /forms/:id/steps | {form\_id, step\_num = 1, step\_label} |  | returns id of step created |  |
| "Gear" is clicked or ‘Add recipient’ is clicked | Approver | POST | /steps/:id/approvers | {step\_id, approver\_emp\_id, can\_add\_next\_approver} |  | returns id of approver created | "employee id is invalid" |
|  | Approver | GET | /steps/:id/approvers |  |  | returns all the approvers for a given step |  |
| "Add Step" is clicked | Step | POST | forms/:id/steps | {form\_id, step\_num = last\_step\_num + 1, step\_label} |  | returns id of step created |  |

The problem with above solution is that there will be several abandoned forms and steps in the db if we POST at every step.

It will be faster, efficient to store all the intermittent data in front-end ( in a redux state for example ) and post it all to backend when ‘Continue’ is clicked. It will also prevent the hassle of having several abandoned forms in the db.

So the following approach will be more appropriate:

|  |  |  |
| --- | --- | --- |
| **Event** | **Action** | **Reducer** |
| Begin Form is clicked | AddForm() | newState = {form\_type, initiator\_emp\_id}, action= ADD\_FORM |
|  | AddStep() | newState = {form\_id, step\_num = 1, step\_label}, action= ADD\_STEP |
|  | AddApprover() | newState = {step\_id, approver\_emp\_id, canAddNextStep}, action= ADD\_APPROVER |
|  |  |  |
| "Gear" is clicked or ‘Add recipient’ is clicked | AddApprover() | newState = {step\_id, approver\_emp\_id, canAddNextStep}, action= ADD\_APPROVER |
| "Add Step" is clicked | AddStep() | newState = {form\_id, step\_num = 1, step\_label}, action= ADD\_STEP |
| X' is clicked | RemoveApprover() | newState = {approver\_id}, action=REMOVE\_APPROVER |
| "Approver in previous step will specify" is checked | setApproverSpecifiesNextApprover() |  |

**When user clicks ‘Continue’  
All the relevant data is posted to relevant resources as specified in the API endpoints table.**

**State Shape** : {

Entities : {

Forms : {

1: {

Type: ‘Trip Expense’,

Initiator\_id: 121,

Status: started,

CreatedAtDate: 10/12/20,

CreatedAtTime: 10:30

},

2 : {

Type: ‘Request for Open\_req for Maths teacher’,

Initiator\_id: 121,

Status: created,

CreatedAtDate: 9/9/20,

CreatedAtTime: 9:20

}

},

Steps: {

1: {

Step\_num: 1

Step\_label: ‘Principal’,

Form\_id: 1

},

2: {

Step\_num: 1

Step\_label: ‘Principal’,

Form\_id: 2

},

3: {

Step\_num: 2,

Step\_label: ‘School District Superintendent’,

Form\_id: 2

}

},

Approvers : {

1 : {

Steps\_id: 1,

Employee\_id: 159,

Can\_add\_next\_step: true

},

2: {

Steps\_id: 1,

Employee\_id: 159,

Can\_add\_next\_step: false

},

3: {

Steps\_id: 3,

Employee\_id: 237,

Can\_add\_next\_step: false

},

Employees: {

1: {

Name: ‘Rose Santos’,

Email\_id: ‘[rose.santos@nusd.](mailto:rose.santos@nusd.com)org’

},

…..

159: {

Name: ‘Beth Roberts’,

Email\_id: ‘beth.roberts@nusd.org’

},

237: {

Name: ‘Tim Lanford’,

Email\_id: ‘Tim.lanford@nusd.org’

}

}

}

Errors: {

Add\_recipient: ‘Recipient employee id is invalid’

}

Session: {currentUserid: 121}

}

**Front End considerations:**

Form-type ( can be a dropdown or text input field - I would normally pick dropdown to avoid spelling mistakes and also make data searching and analysis simpler later on but since the problem statement states that we want to provide maximum flexibility I would go with simple text input field in this case)

Add auto-complete to Employee name field to avoid spelling mistakes. ( I also considered adding alphabetically sorted drop down lists but if the number of employees is big then it will be very tedious to search a name in drop\_down also the dropdown will mask other form fields resulting in low quality UX)

Find the employee name in the Employees table - if not found, throw an error.

If employee name found - prefill the employee email but allow the user to change it.

Validate that email id is valid. If not, throw an error.

If both employee name and email are valid ‘Add recipient’ and ‘submit step’ buttons are enabled.