OPEN OnDemand

Contributor Guide Handout

March 2026

openondemand.org/guide

The Map of Open OnDemand

This guide attempts to layout structures and locations around much of the OOD ecosystem.

These structures range from types such as github repos, to ruby `gem`s, and even to components of OOD itself wrt its codebase.

If you hear a term you don't know, please ask it in Menti!

- https://osc.github.io/ood-documentation/latest/glossary.html

Pull Requests

You need to make a fork of the repo first:

- https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/fork-a-repo
- Generally you are working off the `latest` branch when you begin your work.
- Edit on GitHub button also works too!

OOD Documentation

Open OnDemand's documentation sits in a public GitHub repo, hosted out of GH pages.

The repo:

- https://github.com/OSC/ood-documentation
- Notice the repo has many branches but `latest` and `develop` are the 2 which matter because they generate GitHub Pages which serve the documentation for the community.
- Branch off `latest` and push to `develop` for *feautures not implemented*.
- For *typos and fixes*, push your branch back to `latest`.

GH Pages:

- https://osc.github.io/ood-documentation/master/
- https://osc.github.io/ood-documentation/latest/
- https://osc.github.io/ood-documentation/develop/

Gems

A Ruby `gem` is just a library or module or bundle of code that we can install using the

ruby package manager `bundler`.

- The `gem`s are listed in a project's `Gemfile`.
- After a `bundler` run a `Gemfile.lock` with version dependencies will also be generated.
- `bundler`: https://bundler.io/
  - commands: https://bundler.io/docs.html
  - `bundler config set --local path vendor/bundle` will be your friend later to work on the development dashboard code and our local `gem`s without polluting the system `gem`s.

OOD has 4 `gem`s itself, some of which can be largely ignored, some which are quite useful:

- `ood_packaging` : https://rubygems.org/gems/ood_packaging largely for OOD internal team to help with packaging and distribution of OOD.
- `ood_appkit` : https://rubygems.org/gems/ood_appkit
  - Provides an interface to work with OOD scientific apps, a `dataroot` for OOD apps to write data to and common assets and helper objects.
- `ood_support` : https://rubygems.org/gems/ood_support
  - Provides an interface to work with local OS installed on the HPC center's *web node*. This `gem` is often useful for both OOD and OOD apps.
- `ood_core` : https://rubygems.org/gems/ood_core
  - Provides *Adapters for Schedulers*, `batch_connect` *Templates* for the 3 types of OOD apps, *ACL* functionality, cluster interactivity, and Job interaction.

OOD Core (ood_core)

This is where the actual backend code to interact with your clusters or schedulers resides.

Schedulers and Adapters

OOD provides *adapters* for the various HPC *schedulers* which can all be seen here:

- https://github.com/OSC/ood_core/tree/master/lib/ood_core/job/adapters
- One thing to note is we have k8's adapter if you wish to use k8's as a scheduler.
- LinuxHost: This adapter is used to mimic a scheduler or resource manager, for remote desktop or IDE's.
- SystemD: Community contribution.

3 Species of OOD App

OOD ships with 3 types of scientific apps:

1. `basic` :
   https://github.com/OSC/ood_core/blob/master/lib/ood_core/batch_connect/templates/basic.rb
- HTTP server
- e.g. Jupyter Notebook:
  - https://github.com/OSC/bc_osc_jupyter/blob/db927470af05c71edac770ae321a1ff399caec33/submit.yml.erb#L39

2. `vnc` :
   https://github.com/OSC/ood_core/blob/master/lib/ood_core/batch_connect/templates/vnc.rb
- vnc server
- e.g. Qomsol, Remote Desktops
  - https://github.com/OSC/bc_osc_comsol/blob/69667f971076cd2ba2d23bbbdb508bebe20ebc63/submit.yml.erb#L18

3. `vnc_container` :
   https://github.com/OSC/ood_core/blob/master/lib/ood_core/batch_connect/templates/vnc_container.rb
- Less common, but it exists, vnc with container for sites that don't want to install X11, XFCE, GNOME, etc. on their host. All of these options are what you are setting when you select the `template` in your `submit.yml` files.

Clusters

- The code to work with your clusters and the corresponding cluters config files.
- https://github.com/OSC/ood_core/tree/master/lib/ood_core
  - Split out between 2 files
  - the `clusters.rb` file is to handle the clusters config files.
  - the `clutser.rb` file is to handle working with a cluster and its *scheduler*.

`ood_core` Dev Work

In order for this to work we need to actually touch our `Gemfile` in the `dashboard` and point to our local `ood_core`:

```
gem 'ood_core', :path=> '/full/path/to/checked/out/ood_core'
```

- You must issue the `bin/setup` command to rebuild your `dashboard` once you make these local changes to your `ood_core` code.

Scientific App Development and OOD Features

Now that we've seen the docs and how to edit them, here's a few places in the docs that will be a huge benefit to you and your center for OOD scientific app development.

The `batch_connect` Convention Overview

One of OOD's most powerful abstractions for interactive app development that will help you develop your apps with faster time to compute.

Scientific App File Structure:

```
my_app/
   ├── `form.yml.erb `     ← User-facing form
   ├── `manifest.yml `     ← App metadata
   ├── `connection.yml `   ← App server data needed on frontend (logins, hostnames, etc.)
   ├── `submit.yml.erb `   ← Job submission params
   └── `template/ `
       ├── `before.sh.erb `  ← Pre-launch setup
       ├── `script.sh.erb `  ← Main launch script
       └── `after.sh.erb `   ← Cleanup (Run at the end of the session)
```

Execution Flow:

1. `form.yml.erb` — Renders form → user fills it out
2. `submit.yml.erb` — Generates job submission config
3. `before.sh.erb` — Runs before main script
4. `script.sh.erb` — Launches the application
5. `after.sh.erb` — Cleanup when *session* ends (*Not* when the job ends)

- Docs: https://osc.github.io/ood-documentation/latest/how-tos/app-development/interactive.html
- Helpers source: https://github.com/OSC/ondemand/tree/master/apps/dashboard/app/helpers

The ERB objects

OOD uses Ruby for its backend language and a templating engine called `ERB` which stands for **"Embedded Ruby"**

- Notice, we can use this to generate or read data on the backend.
- This pattern prevalent in most apps you pull down from OSC and you can know which files use this convention by looking for file names that end in `*.erb` such as `script.sh.erb` or `form.yml.erb`, all that matters to `ERB` is that file extension name which then tells the templating engine to uptake that file and execute the ruby code found in the file betweem tse `ERB` tags.
- The engine will either then return a string in place of the expression, or it will render as blank ultimately but provide a ruby statement for a variable or branching logic or some type of code you need to run but wish to not actually return anything in the file itself.

Ruby `ERB` code runs between these tags in OOD:

```
<%= ruby_expression %>   <%- ruby_statement -%>
```

Notice the `=` in one expression and the `-` in the statement. This is *crucial* to working with `ERB` code to understand:

- When you need the `ERB` to render an actual string in the file, use `=` as seen beginning `ruby_expression` above.
- If you don't want an actual string returned, such as for a variable you or logic, use the `-` as seen around `ruby_statement` above.

ERB gives your app scripts access to **two powerful Ruby objects** that OOD populates for you:

- `session` — Info about the running session (job ID, `cluster`, `host`, etc.)
- `context` — The form values the user submitted

The `context` Object

The `context` object gives you access to every form field the user filled out on the backend using `ERB` in your `*.erb` files.

Every field in your `form:` array becomes a method on `context`. This is a powerful and promoted pattern in OOD that will save you a lot of time in your app development.

Given a `form.yml` like:

```
form:
- bc_num_hours
- bc_num_slots
- bc_account
- bc_queue
- version
- auto_modules_app
```

You can access any of those fields in your `script.sh.erb` by using `ERB` and calling the set attribute on `context`:

```
# Access any form field:
<%= context.version %>
<%= context.bc_num_hours %>
<%= context.bc_account %>
<%= context.bc_queue %>

# Use in conditionals:
<%- if context.version == "4.3" -%>
 module load R/4.3
<%- end -%>
```

The `session` Object

The `session` object provides runtime information about the current batch connect session.

Attribute	What it gives you
`session.id`	Unique session identifier
`session.job_id`	The scheduler job ID
`session.cluster`	Name of the cluster (matches cluster configs
`session.staged_root`	Path to the session's staged directory
`session.created_at`	When the session was created

```
# Example: Kubernetes-aware logic
<%- if session.cluster =~ /kubernetes/ -%>
 # K8s-specific setup here
<%- end -%>
```

Helper Methods: Stop Reinventing the Wheel!

OOD provides helper methods that we see users reinvent all the time. **Use these instead!**

`find_port`

Finds an available port on the compute node. No need to hardcode or guess.

```
# In script.sh.erb:
port=$(find_port ${host})
export port
```

`create_passwd`

Generates a secure random password of the given length. Great for app auth.

```
# In script.sh.erb:
password="$(create_passwd 16)"
export RSTUDIO_PASSWORD="${password}"
```

Common Useful Patterns

`connection.yml` entry

Suppose we have an app that wants to use its own auth mechanism. OOD would not be aware of this password which gets generated and it could create a complex work around in order to retrieve this data to share with OOD.

Well OOD has a pattern for this! We use the `connection.yml` and the `conn_params` in the `submit.yml.erb` file in conjunction with the app's `view.html.erb` file to generate this data, plug it in for the user, and never expose the credentials or involve sharing of those credentials.

We will use RStudio to show this pattern off, and to notice that this app needs a `csrf` token to launch, another quirk you may find in apps that we can handle with this pattern.

How this works is we use the `before.sh.erb` script to generate this needed `password` and `csrf_token` for our app:

```
# rstudio 1.4+ needs a csrf token
csrf_token=<%= SecureRandom.uuid %>
# Define a password and export it for RStudio authentication
password="$(create_passwd 16)"

export RSTUDIO_PASSWORD="${password}"
```

Note that we used a lowercase variable for `password` here, this is a necessary convention that you *must* follow for this pattern to work.

Next, we need to ensure we have the `csrf_token` for our app when it is submitted to the cluster using the `submit.yml.erb` like so:

```
---
batch_connect:
 template: "basic"
 conn_params:
  - csrf_token
...
```

This is awesome! We've generated the token and shared it with our job as it is spun up.

Now, let's combine all this together in the `view.html.erb` to see how we then put all this data into our app's session card when it's ready:

```
<script type="text/javascript">
(function () {
 let date = new Date();
 date.setTime(date.getTime() + (7*24*60*60*1000));
 let expires = "expires=" + date.toUTCString();
 let cookiePath = "path=/rnode/" + "<%= host.to_s %>" + "/" + "<%= port.to_s %>/";
 /**
  rstuido wants a cookie called csrf-token - but that's going to change in 2020!
 */
 let cookie =  csrf-token=<%= csrf_token %>;${expires};${cookiePath};SameSite=strict;secure ;
 document.cookie = cookie;
})();
</script>

<form action="/rnode/<%= host %>/<%= port %>/auth-do-sign-in" method="post" target="_blank">
 <input type="hidden" name="csrf-token" value="<%= csrf_token %>"/>
 <input type="hidden" name="username" value="<%= ENV["USER"] %>">
 <input type="hidden" name="password" value="<%= password %>">
 <input type="hidden" name="staySignedIn" value=1">
 <input type="hidden" name="appUri" value="">
 <button class="btn btn-primary" type="submit">
  <i class="fa fa-registered"></i> Connect to RStudio Server
 </button>
</form>
```

This provides and hides a few pieces of data for the user to connect:

- the `csrf` token: `<input type="hidden" name="csrf-token" value="<%= csrf_token %>"/>`
- the `password` we made: `<input type="hidden" name="password" value="<%= password %>">`
- the username needed: `<input type="hidden" name="username" value="<%= ENV["USER"] %>">`

You can see from this that all the work of usernames, passwords, and other data can all be handled from OOD and shared between the app's files in a way to make the user's experience seamless and free of login pop-ups.

Kubernetes-Aware Branching

A very common pattern you'll see in apps that need to support both traditional HPC schedulers and Kubernetes:

```
<%- if context.cluster =~ /kubernetes/ -%>
 source /bin/find_host_port      # K8s port assignment
 source /bin/save_passwd_as_secret # Store password securely
 host="$HOST_CFG"
 port="$PORT_CFG"
<%- else -%>
 port=$(find_port ${host})       # Traditional HPC
 password="$(create_passwd 16)"   # Generate password
<%- end -%>
```

Dynamic Forms with `form.yml.erb`

You can use `ERB` in your form definition to dynamically populate dropdowns from the filesystem:

```
# form.yml.erb — use ERB to make forms dynamic!
attributes:
 version:
  widget: select
  options:
   <%- Dir.glob('/software/R/*/').each do |d| -%>
   - ["<%= File.basename(d) %>", "<%= File.basename(d) %>"]
   <%- end -%>
```

Putting It All Together: `before.sh.erb` and the `script.sh.erb`

Here's a complete example showing `context`, `find_port`, and `create_passwd` working together using the `before.sh.erb` script and the `script.sh.erb`.

We need to generate our data for the script before it runs using the `before.sh.erb` pattern that OOD provides:

```
# Set up networking — use OOD's helpers!
export host=$(hostname)
export port=$(find_port ${host})

# Generate secure auth
export password="$(create_passwd 16)"
export RSTUDIO_PASSWORD="${password}"
```

Now we can use these variables we've set in our `script.sh.erb` like below:

```
#!/usr/bin/env bash

# Load modules based on user's form selection (context)
module load rstudio/<%= context.version %>

# Write connection info for OOD to read
echo "Starting RStudio on ${host}:${port}"

# Launch the application
rserver --www-port ${port} \
    --auth-none 0 \
    --auth-pam-helper-path /usr/lib/rstudio-server/bin/pam-helper \
    --server-data-dir /tmp/rstudio-data
```

These patterns are incredibly useful for any scientific app that needs to share data between the backend and the session card. And as we saw in the `connection.yml` pattern above, we can take this further by passing the data through to the `view.html.erb` giving users a seamless login experience, like never seeing a password prompt for example.

OOD Monorepo

Open OnDemand follows the *Mono-repo* pattern. What this means is that OOD has many various applications utilities all contained under the `Ondemand` namespace.

- https://github.com/OSC/ondemand

Notice we have our `apps` under this which leads to the `dashboard` and all the code a user would be familiar interacting with, but at this top level we see much more.

`ood-portal-generator`

Generates the NGINX and Apache configs given a valid `ood_portal.yml` file.

`nginx_stage`

Used to manage the PUN for OOD.

`ood_auth_map`

Connects the auth system to the web-node system users for OOD at login.

`packaging`

Used to help with the distribution of OOD software.

`mod_ood_proxy`

Used to manage the proxy for OOD.

`dashboard`

This is the frontend `rails` code most users and admins are familiar interacting with.

OOD Dashboard Code Components

OOD has several components within the `rails` application code

- `active_jobs`
- `bc_desktop`
- `dashboard`
- `file-editor`
- `files`
- `myjobs`
- `projects`
- `shell`
- `system-status`

MVC Components

OOD uses the MVC pattern of web-development which is default in `rails`.

- https://guides.rubyonrails.org/v7.1/getting_started.html#mvc-and-you
- Rails philosophy is *convention over configuration*.

We can see the OOD conventions by looking at the various `Models`, `Controllers` and `Views` within the `rails` code itself. For a detailed walkthrough of MVC in OOD, see MVC in the Project Manager (https://github.com/OSC/contributor_guide/blob/main/contributor_jam_guide.md#model-view-controller-in-the-project-manager).

Models

- https://github.com/OSC/ondemand/tree/master/apps/dashboard/app/models
- All the data OOD is aware of is defined in this directory.

Controllers

- https://github.com/OSC/ondemand/tree/master/apps/dashboard/app/controllers
- Here we see what data the model can present to a view.

Views

- https://github.com/OSC/ondemand/tree/master/apps/dashboard/app/views
- This directory can be daunting as it contains all code used to present the data to users. As such, there can be a great many components to any piece of OOD.
    - e.g. https://github.com/OSC/ondemand/tree/master/apps/dashboard/app/views/batch_connect
    - While this is one of the more complex views to deal with, it gives a sense of how complex some of these files can become.
    - The goal for these when working is to try and make things *modular* and *logical*.
    - Rails also uses the notion of *partials* to provide components of views.
        - These files start with an underscore `_my_partial`.
        - https://osc.github.io/ood-documentation/latest/customizations.html#overriding-pages

Utilities

OOD also has some helpers for things like `rclone`, some `rake` tasks, and other functionality all contained here:

- https://github.com/OSC/ondemand/tree/master/apps/dashboard/lib

`batch_connect` convention

One of the more powerful and useful abstractions to be aware of in OOD.

- https://github.com/OSC/ondemand/tree/master/apps/dashboard/app/models/batch_connect
- This model provides most data you see in a session card and when interacting with forms.
- `helpers` are a big piece of the puzzle here:
  - https://github.com/OSC/ondemand/tree/master/apps/dashboard/app/helpers

Code Changes

Now that we've had a breakneck tour, let's make changes.

This portion assumes you have setup a dev dashboard on your OOD instance as described below:

- https://osc.github.io/ood-documentation/latest/how-tos/app-development/enabling-development-mode.html

And that you have forked, cloned, and built your dashboard as described here:

- https://github.com/OSC/ondemand/blob/master/DEVELOPMENT.md#developing-the-dashboard

Seeing Changes

Some changes will only require we reload the browser:

- If a config change is not taking effect after several reloads, try a PUN restart.
- **Environment variable changes require a rebuild of the PUN with `bin/setup`**

Sample PR Walk Through

GitHub Issue Page

- https://github.com/OSC/ondemand/issues

Resources:

- Contributing guide: https://github.com/OSC/ondemand/blob/master/CONTRIBUTING.md
- Dev Dashboard Setup: https://github.com/OSC/ondemand/blob/master/DEVELOPMENT.md#developing-the-dashboard
- Development guide: https://github.com/OSC/ondemand/blob/master/DEVELOPMENT.md
- Dockerfile: https://github.com/OSC/ondemand/blob/master/Dockerfile
- Security and Reporting: https://github.com/OSC/ondemand/blob/master/SECURITY.md
- Code of Conduct: https://github.com/OSC/ondemand/blob/master/CODE_OF_CONDUCT.md
- Rails Guides: https://guides.rubyonrails.org/v7.1/

Deep Dives

Model View Controller in the Project Manager

For a practical example of the MVC paradigm in action, we can take a closer look at the Project Manager. Although the Project Manager is a single component, it is composed of three different entities: Projects, Launchers, and Workflows. Each of these entities have their own model, view, and controller, but interact with one another to manage their relationships and data.

Relationships

The basic relationships necessary for a working project are

- A User has many Projects
- A Project has many Launchers
- A Project has many Workflows
- A Workflow has many Launchers

In a typical web app, these relationships would be defined in a database schema. However the OnDemand dashboard does not use a database, instead managing its data through the filesystem. So where do these relationships 'live'? The first time these come up is during **routing**.

```
# apps/dashboard/config/routes.rb

Rails.application.routes.draw do
 if Configuration.can_access_projects?
  get 'projects/possible_imports' => 'projects#possible_imports', :as => 'project_possible_imports'
  post 'projects/import' => 'projects#import_save', :as => 'project_import_save'


  resources :projects do
   root 'projects#index'
   get '/jobs/:cluster/:jobid' => 'projects#job_details', :defaults => { :format => 'turbo_stream' }, :as => 'job_details'
   delete '/jobs/:cluster/:jobid' => 'projects#delete_job', :as => 'delete_job'
   post '/jobs/:cluster/:jobid/stop' => 'projects#stop_job', :as => 'stop_job'


   resources :workflows do
    member do
     post 'submit'
     post 'save'
     get 'load'
     get 'clone'
    end
   end


   resources :launchers do
    post 'submit', on: :member
    post 'save', on: :member
    get 'render_button', on: :member
    get 'clone', on: :member
   end
  end
end
```

At the very top are the routes that are always static for a given user, and thus do not require any parameters to generate their pages. For example, 'projects/possible_imports' detects projects that you can access based upon your UNIX group and shared space configurations, and does not have to be connected to an individual project.

Next, we have the line `resources :projects do`, which starts a block that contains the rest of the snippet.The line is an example of Rails Resource Routing (https://guides.rubyonrails.org/routing.html#resource-routing-the-rails-default), a shortcut that automatically defines some common routes for a given entity. Each route defined within this block automatically receives a `/:project/` parameter at the start of their url, meaning they are defined for each project that a user has access to.

Finally, the `resources :workflows do` and `resources :launchers do` lines serve the same function as `resources :projects`, defining basic routes and containing a block of routes that require both a `:project` parameter and a `:workflow` or `:launcher` parameter respectively, defining these routes for each workflow or launcher that a project contains.

Controllers

Each **route** defined above directs the request parameters to a method on a **controller** in order to render that page or perform that action. For some routes, the controller action is explicitly defined while others do so implicitly. For example, the line `post '/jobs/:cluster/:jobid/stop' => 'projects#stop_job', :as => 'stop_job'` explicitly points the url to `projects#stop_job`, which Rails interprets as the `stop_job` method defined on `ProjectsController`. On the other hand, the line `post 'submit'` does not contain a url or a controller action in the definition. For this route, Rails uses both the `resources :projects do` and `resources :workflows do` blocks containing the route to generate the url fragment `/:project/:workflow/submit` and direct this to the `submit` method on `WorkflowsController`.

Following a submit request to WorkflowsController#submit, we see

```
# apps/dashboard/app/controllers/workflows_controller.rb

 def submit
  return unless load_project_and_workflow_objects(render_json: true)
  metadata = metadata_params(permit_json_data)
  @workflow.update(metadata)
  submit_param = Workflow.build_submit_params(metadata, project_directory)
  result = @workflow.submit(submit_param)
  if !result.nil?
   render json: { message: I18n.t('dashboard.jobs_workflow_submitted'), job_hash: result }
  else
   msg = I18n.t('dashboard.jobs_workflow_failed', error: @workflow.collect_errors)
   render json: { message: msg }, status: :unprocessable_entity
  end
 end

 private

 def load_project_and_workflow_objects(render_json: false)
  @project = Project.find(project_id)
  @workflow = Workflow.find(workflow_id, project_directory)
  return true if @workflow.present?

  if render_json
   render json: { message: I18n.t('dashboard.jobs_workflow_not_found', workflow_id: workflow_id) }, status: :not_
found
  else
   redirect_to project_path(project_id), notice: I18n.t('dashboard.jobs_workflow_not_found', workflow_id:
workflow_id)
  end
  false
```

```
end

def index_params
  params.permit(:project_id).to_h.symbolize_keys
end

def project_id
  params.permit(:project_id)[:project_id]
end

def workflow_id
  params.require(:id)
end
```

Notice that **controller actions** are always public methods, and everything under the `private` flag is used within actions, but is not an action itself. Starting from the top of `submit`, we see the order in which the logic is executed.

- Fetch project and workflow objects based on request parameters
- Fetch metadata from request parameters
- Update the workflow object with metadata
- Create scheduler parameters from workflow object
- Submit scheduler parameters and collect response
- Return a JSON response stating success or failure.

While it is a bit hard to see in the code above, all the parameters included with the request must be accessed through the `params` object, which is available everywhere in the controller. In this case, since `Workflows#submit` corresponds to an action, not a page, we just send back a JSON response that is rendered by the page the user is currently on (`Workflows#show` in this example).

As we see with `Workflows#submit`, not all controller actions correspond to views. For an example that does render a view at the end, consider the action for `Workflows#show`.

```
def show
  return unless load_project_and_workflow_objects
  launcher_ids = @workflow.launcher_ids

  @launchers = Launcher.all(project_directory).select { |l| launcher_ids.include?(l.id) }
end
```

Following line-by-line again we see

- Project and workflow objects fetched with the same private method as above
- Workflow provides a list of launchers it 'has'
- List of ids from workflow is compared with the launcher objects in the projects
- List of actual launcher objects is stored in `@launchers`

We can tell that it returns a standard HTML view response because there is no explicit `render` line like we saw above in `submit`.

Views

To find the specific view file used by the `show` action, we look for `show.html.erb` in `apps/dashboard/app/views/workflows/`.

```
# apps/dashboard/app/views/workflows/show.html.erb

<%= javascript_include_tag 'workflows', nonce: true, defer: true %>

<input type="hidden" id="project-id" value="<%= @project.id %>">
<input type="hidden" id="workflow-id" value"<%= @workflow.id %>">
<input type="hidden" id="base-workflow-url" value="<%= project_workflow_path(@project.id, @workflow.id) %>">
<input type="hidden" id="base-launcher-url" value="<%= project_launchers_path(@project.id) %>">

<div id="workflows_app">
 <div class="toolbar" aria-label="toolbar">
  <% hidden_class = @workflow.editable? ? '' : 'd-none' %>
  <%= select_tag "select_launcher", options_from_collection_for_select(@launchers, :id, :title), include_blank:
false, class: "form-control w-25 #{hidden_class}" %>
  <button id="btn-add" class="<%= hidden_class %>">Add Launcher</button>
```

In this small snippet, we can see the view using the model objects we stored in variables in `Workflows#show`. The top few lines with `type="hidden"` pass relevant data to javascript, like ids and url paths specific to the project and workflow, and further down we see the `@launchers` variable being used to seed a select input. All of the helper methods seen here, like `project_workflow_path` or `options_from_collection_for_select`, are built-in Rails helpers. See Action View Helpers (https://guides.rubyonrails.org/action_view_helpers.html) and Action View Form Helpers (https://guides.rubyonrails.org/form_helpers.html) for an overview of the built-in helpers available in every view.

We can also see from this snippet how Rails views use ERB, or Embedded Ruby. For example the line `<% hidden_class = @workflow.editable? ? '' : 'd-none' %>` contains a line of ruby code that stores a string in the `hidden_class` variable, based on the `editable?` method in the Workflow model. That `hidden_class` class variable is then added to each element on the page that we want to hide if `@workflow.editable? == false`.

Models

In the controller and view snippets above, we saw how they identify and store the relevant models (`@project`, `@workflow`, `@launchers`), and how they call methods on these models to determine behavior. In the controller case with `Workflows#submit`, we see it use the result of `@workflow.submit(submit_param)` to determine whether to return a success response or a failure response. In the view, it calls `@workflow.editable?` to determine whether a class is included in certain elements, and by extension, which elements appear on the page.

This is the primary function of models, to manage data and provide endpoints for the controllers and views to interact with this data. In particular, models are helpful because they isolate the logic and complexity in a single class, allowing us to keep the controllers and views as simple as possible.

```
# apps/dashboard/app/models/workflow.rb

 def manifest_file
  Workflow.workflow_dir(@project_dir).join("#{@id}.yml")
 end

 def update(attributes, override = false)
  update_attrs(attributes, override)
  return false unless valid?(:update)

  save_manifest(:update)
 end
```

```
def update_attrs(attributes, override = false)
 [:name, :description, :launcher_ids, :metadata].each do |attribute|
  next unless override || attributes.key?(attribute)
  instance_variable_set("@#{attribute}".to_sym, attributes.fetch(attribute, ''))
 end
end

def editable?
 manifest_file.writable? || !shared?(manifest_file)
end
```

In this small snippet, we get a good overview of what a basic model contains

- `manifest_file` constructs a path where workflow settings are saved as YAML
- `update` is directly used in WorkflowsController to modify workflow settings
- `update_attrs` is an internal method that facilitates the modification
- `editable?` reads the state of the manifest file to determine if the user has permission to overwrite it.

The biggest advantage of MVC is it allows us to independently develop our models, views, and controllers. This means that as long as the `update` and `editable?` methods continue to exist on the Workflow model, we can update the underlying logic (like what makes a workflow 'editable') in a single place, while its interactions (like hiding certain elements when it is not) remain the same.