
DMRGenie Developer's Guide

Aaron Dayton

August 2, 2024

Contents

I ABSTRACT

II INTRODUCTION

This document covers how to add/modify features in the DMRGenie webapp. DMRGenie is built on a Model View Controller (MVC) infrastructure. For information on the implementation of the MVC framework with basic functionality, the package decisions, or how to run the app, see the `Readme.md` file. Some information is common between these documents.

Many instructions can be found in the Genie documentation here. For example, to learn how to create a new MVC group read this page.

III File structure

The main folder of concern for editing the application is `TenseEZ/app/resources`. Within this folder, all the models, views, and controllers are sorted into distinct folders. As of May 16, 2024, the only group included is the `tensornetwork` group. A supporting folder is `TenseEZ/public`. The file `TenseEZ/routes.jl` handles the exchange of data between the front and back end.

A The `tensornetwork` MVC group

Within the resources folder is the `tensornetworks` \hookrightarrow directory, which contains `TensorNetworks.jl` \hookrightarrow , `TensorNetworksController.jl`, `views`, and `TensorNetworksValidator.jl`.

The file `TensorNetworks.jl` contains the tensor network model. Think of it as an object which can be passed between the front and back end of the application. If you want any data to be exchanged between these ends, then you must include a new field with a default value in the model. Make sure that variable names are descriptive and follow the same format as existing names. Note that modifying this file requires a full restart of the application.

The file `TensorNetworksController.jl` contains the tensor network controller. This is responsible for serving up the HTML page with the relevant data. The main function within this file is `DMRGenie` which is run in `TenseEZ/routes.jl` when the website is first started and when a tensor network algorithm is subsequently run.

When the application is first started up `DMRGenie` \hookrightarrow `()`, with no input, is run. This generates a default tensor network as defined in the model, which is then passed into the `html` function from Genie. The `html` function takes the MVC group name (in this case `:tensornetworks`), the desired view name (in this case `:DMRGenie` corresponding to the file `views`

\hookrightarrow `/DMRGenie.jl.html`), and any additional input fields required by the view.

When a tensor network is built by the user and sent to the back end, it is processed by the `TenseEZ/routes.jl` file, which builds the desired `TensorNetwork` type (which just holds metadata) and sends it to `TensorNetworksController.jl`. In `TensorNetworksController.jl` the actual Tensor Network is produced in terms of MPS and MPO, then the desired algorithm is run on it with `DMRJtensor`. The results are then stored in variables and passed through the `html` function to be served up to the user.

The folder `views` contains the HTML for the application. As of May 16, 2024, the only file in this folder is `DMRGenie.jl.html`. This file defines the landing page of the application: every box, button, option, color, and word is included here or in another file linked to it.

Notice the file type of `.jl.html`. This indicates that the file is capable of running Julia code. To do this you must enclose any Julia code in `$(...)`. Only data passed in via the `html` function may be accessed in this file by invoking the same name. For example, `hamiltonain_measurement` is passed into `html` \hookrightarrow and is accessed by `$(hamiltonain_measurement` \hookrightarrow), or `tensor_network` is passed into `html` and the `default_model` field of it is accessed by `$(` \hookrightarrow `tensor_network.default_model)`.

JavaScript code can be run within an HTML file as well. External JS files and inline scripts are included at the top level within `<head>`. For example, the line `<script src="/js/drawing.js"></script>` \hookrightarrow `>` makes the functions within `TenseEZ/public/js/drawing.js`, like `update()`, available to the HTML file. Another example of JS being used in the file would be in something like the `onclick` field of a button which calls a JS function when a click action is performed on the button in question: `onclick="` \hookrightarrow `HideOrShowCorrelationOptions()`".

Learning HTML is crucial. Google is your best friend when working with it. Many components have already been built for other applications and are readily available in articles or stack overflow posts. It is often difficult to get spacing to work nicely in HTML right away, so it is recommended that you create every piece of a new feature in an ugly way before making it look nice to avoid redoing work.

The file `TensorNetworksValidator.jl` currently has no support with our application, but this should be added. Refer to DMRGenie documentation through this link.

B Public

Another important folder is **TensEZ/public**. Within it are **js**, which contains the JavaScript, **css**, which contains the styling CSS files, **img**, which contains any images or gifs, and **usergenerated**, which contains any user generated data such as downloadable output files. If you make any new JavaScript files, CSS files, images, etc., then they should go in the corresponding folder.

The **usergenerated** folder is for files generated by the user. As of now, partial density matrices are saved into files and given to the user as available downloads. This is done quite sloppily and will likely need to change. When the application is booted up this directory is automatically cleared, but this could cause issues with multiple users using the application at once. The files do not need to be kept around long term so a database may be overkill. It is unclear if it is a best practice to have them just sitting in the public folder though. *A decision will need to be made about this.*

C Routes

The final important file of note is **TensEZ/routes.jl**, which handles all the data transfer from the front end and back end via HTTP methods, such as POST requests, and the application bootup process.

There is one main POST request for the webpage currently which computes the following steps:

1. Initialize all the necessary variables
2. Begin a try-catch block for error handling in the case of malformed data or unforeseen issues
3. Get the data passed from the front end, stored in the name attribute of some tag, with the **postpayload** function which takes a Symbol of the corresponding name and the default value. The **parse** function is used in the case of numerical data.
4. Create a new **TensorNetwork** and pass it to the controller via the **DMRGenie** function.

The most complicated part of this process is parsing the data via **postpayload** functions, since each field must be handled individually. All it boils down to is just getting the name of what data you want to read, reading it, then storing it in the correct format. If there is some unknown number of fields which must be read, then use a loop to iterate the names with the same naming convention as is used on the front end until no data is recieved from **postpayload**.

It should be noted that this setup is fragile in some ways. The application must be restarted if the names of any routes are changed. Sometimes a

fill compute restart is required to get things working again.

The **TensEZ/routes.jl** file also handles the startup of the application with the following lines:

```
up()

file = open("bootup.log", "w")
write(file, "INITIALIZATION COMPLETE")
close(file)
```

The **bootup.log** file is there for the deployment infrastructure to work. When a docker image is deployed using the containerized application, the startup sequence is run through Julia Genie commands and the deployment infrastructure must wait until this is done. So it waits to read the **bootup.log** file, and once it does the webpage is made available.

IV Deployment

To deploy the DMRGenie website one must build a docker image, and deploy it on dockerhub. This section covers the relevant basics of docker and how to deploy.

A Dockerfile

There are three steps in the dockerization of an application:

1. Creating a dockerfile
2. Turning the dockerfile into an image
3. Booting up the image into a container environment where code can be executed

To deploy the application, it is turned into an image with Docker and deployed using software developed by Drew Leske of ARCsoft.

See the dockerfile for how the docker image and container are built (each command is a "layer"). The docker image is public on dockerhub due to a requirement of the deployment software by Drew Leske of ARCsoft.

Any dockerfile consists of a list of ordered commands which build layers of a docker image when built. The containerized version of the application is basically a specification for a computing environment. Docker images are made from the container and function as the environment where processes can be run. So what we built for Drew Leske is an image, which his software uses to instantiate docker container instances, which are then run on computing nodes when requests are made to the website.

Documentation for docker can be found here. Our dockerfile is built as follows:

- Specify a base image you wish to build on top of. For example, you could build on a specified version of the Ubuntu or Alpine docker images. We use the latest version of the julia image for linux/amd64
- Configure the environment: create directories, copy files over, edit permissions, etc.
- Add any packages used in the application
- Specify ports and environment variables
- Run the application via a CMD instruction. The server file specified just contains the line which starts the application: `julia --color=`
`→ yes --depwarn=no --project=@. -q -i`
`→ -- $(dirname $0)/../bootstrap.jl -s=`
`→ true "$@"`

B How to deploy

To deploy there are a few simple steps which we will write in plain English first, then in the CLI commands to be executed. The terminal commands should be executed in the `tensez/TensEz` directory which holds the dockerfile. Before doing any of that though, make sure you have Docker Desktop installed.

1. Build a docker image with a tag specifying the version (i.e. `v0.2.1`)
 Command:
`docker build -t <Your Docker Hub Account>/dmrgenie:v0.2.1 .`
 (Make sure to change the version tag!)
2. Deploy the image with the same tag to docker-hub
 Command:
`docker push <Your Docker Hub Account>/dmrgenie:v0.2.1`
 (Make sure to change the version tag!)
3. Email Drew Leske that a new docker image is up. Make sure to tell him the version tag (in this case we would tell him the new docker image is at `<Your Docker Hub Account>/dmrgenie:v0.2.1`)!

So far, all docker images have been deployed to `aarondayton87/dmrgenie`, but you will need to deploy them to your personal repo. You can see this public repo on dockerhub for a history of versions though.