Docker Implementation

Railway Administration and Information Logical System

RAILS for Model Railroads

David Bristow
Version 2.2.0
February 5, 2026

Contents

Chapter 1

Introduction

Railway Administration and Information Logical System (RAILS) is a software model and implementation of an automated system to assist the model railroader achieve realism in the operation of a model railroad. The model then drives the development of hardware and or software.

1.1 Microservices Design Components

Figure 1.1 shows the microservices components that make up the design of RAILS.

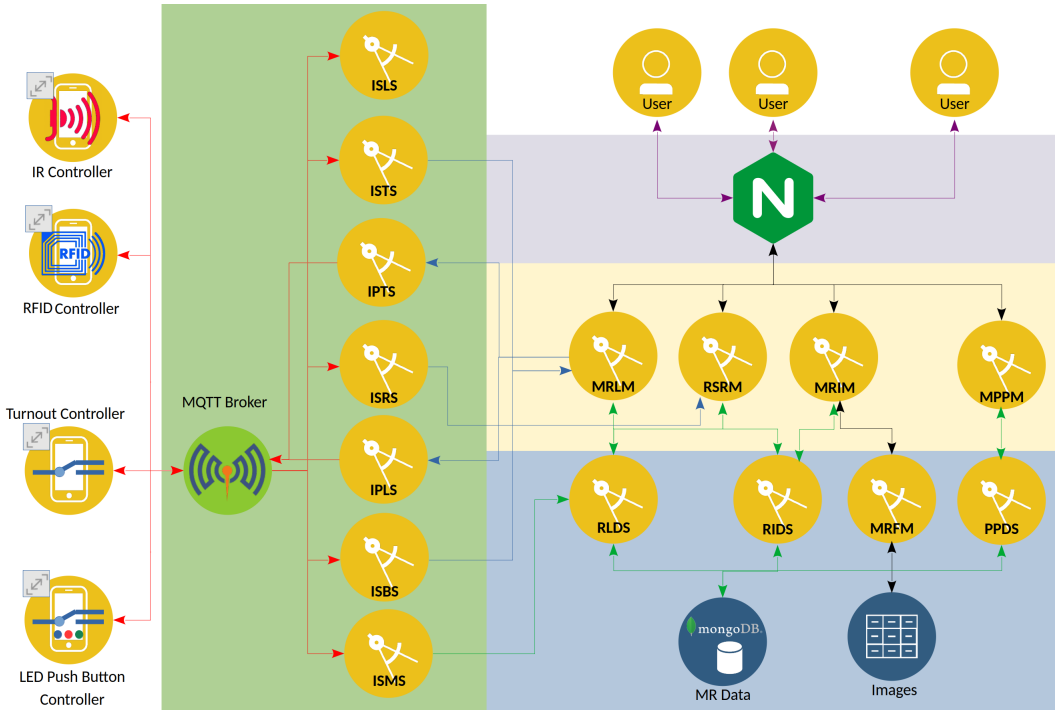

Figure 1.1: Microservices Component Architecture

The microservices design components are divided into three sets:

- Internet of Things (IoT) components, which are highlighted with the light green colored background in Figure 1.1.

- Data Services (DS) components, which are highlighted with the light blue colored background in Figure 1.1.
- Single Page Application (SPA) components, which are highlighted with the buff colored background in Figure 1.1.
- a reverse proxy, which is highlighted with the mauve colored background in Figure 1.1.

1.1.1 IoT Components

The components of the IoT set of design components are subdivided into:

- Micro Controllers using the Message Queuing Telemetry Transport (MQTT) protocol:
  - Radio Frequence Identification (RFID) Controller processes RFID tags obtained from a RFID reader and then publishes the value
  - Turnout Controller subscribes to turnout commands then to act on the command to cause the turnout to move. It then publishes the state of the turnout
  - infrared (IR) Controller (in planning) processes IR sensors and publishes their values
- MQTT Broker is the heart of any publish/subscribe protocol, is responsible for receiving messages, posting to designated topics and sending messages to clients subscribing to topics.
- The subscribers and publishers bridge the MQTT elements with the Graphic User Interface (GUI) applications:
  - IoT Publisher Turnout Panel Light Services (IPLS) publishes turnout panel light commands a Turnout Panel Controller
  - IoT Publisher Turnout Services (IPTS) publishes turnout commands to a Turnout Controller
  - IoT Subscriber Turnout Panel Button Services (ISBS) subscribes to push button events and pushes them via a web-socket to the MRLM component
  - IoT Subscriber Location Services (ISLS) (in planning) IoT subscribes to topics that provide location information i.e., IR Sensors and RFID sensors
  - IoT Subscriber Micro-controller Services (ISMS) subscribes to micros and adds or updates micros collection in RAILS. It also subscribes to micro heartbeats.
  - IoT Subscriber RFID Services (ISRS) subscribes to RFID tags and pushes them via a web-socket to the Rollingstock RFID Manager (RSRM) component
  - IoT Subscriber Turnout Services (ISTS) subscribes to turnout switch closures and pushes them via a web-socket to the Model Railroad Layout Manager (MRLM) component

1.1.2 DS Components

DS consist of all the components that handle and or store the model railroad data:

- MR Data – the document repository, MongoDB, to store complete collections of items such as rolling stock, industries (producers and consumers), track elements, turnouts, projects, purchases, etc.

- Railroad Inventory Data Services (RIDS) provides Representational State Transfer (REST) access to railroad inventory documents
- Plans and Purchases Data Services (PPDS) provides REST access to model railroad projects and purchases documents
- Railroad Layout Data Services (RLDS) provides REST access to model railroad layout documents
- Model Railroad File Manager (MRFM) provides the user the ability to upload image files for the use by the Model Railroad Inventory Manager (MRIM) component
- Images is the file store for the images uploaded by MRFM component and used by the MRIM component

1.1.3 SPA Components

GUI applications that provide user access to RAILS:

- RSRM, this SPA allows a user to match a RFID value to a rolling stock road name and number
- MRIM, this SPA allows a user to create, update and delete model railroad assets, such as rolling stock
- Model Projects and Purchase Manager (MPPM), this SPA allows a user to enter information about their projects and purchases
- MRLM, this SPA allows a user to enter information about their layout and control elements of it

1.1.4 Reverse Proxy

A server that sits in front of the Single Page Applications (SPAs), which allows users to access the SPAs from any browser on any Personal Computer (PC) in the network.

1.2 Docker

In the context of microservices, containers are used to package the code, libraries, and configuration files needed for a microservice to run as a single, executable unit. This makes it easy to deploy and run microservices in different environments, such as on-premises servers or in the cloud. By packaging all the dependencies for a microservice in a container, it can be guaranteed to run the same way regardless of the environment it's deployed in. This helps to reduce issues caused by differences between development, testing, and production environments. Containers also make it easy to scale up or down the number of instances of a microservice running, and to update or rollback a microservice without affecting other services.

Docker is a platform that makes it easy to create, deploy, and run applications in containers. It provides a command line interface (CLI) and a set of Application Program Interfaces (APIs) that make it simple to work with containers. Docker in conjunction with microservices is used to package and

deploy each service in its own container. This allows each service to be managed, deployed, and scaled independently of other services. The Docker platform provides a number of tools to help manage and orchestrate the containers that make up a microservice-based application, such as Docker Compose and Docker Swarm.

Docker is an open-source platform that enables developers to automate the deployment, scaling, and management of applications using containerization. Containers are lightweight, isolated environments that encapsulate an application and all its dependencies, including libraries, frameworks, and other runtime components. Docker allows the user to package an application into a container image, which can then be run consistently across different environments, such as development, testing, and production.

Docker provides the following features:

- Docker containers are portable and can run on any system that supports Docker, regardless of the underlying operating system or infrastructure. This eliminates the ”works on my machine” problem and ensures consistent behavior across different environments.
- Containers provide a high level of isolation, ensuring that applications and their dependencies are encapsulated and do not interfere with each other. This isolation improves security, as well as prevents conflicts between different software components.
- Docker containers are lightweight and share the host system’s operating system kernel. This means they require fewer resources compared to traditional virtual machines (VMs). Multiple containers can run on the same host without significant performance overhead.
- Docker makes it easy to scale applications horizontally by running multiple instances of containers across different hosts or a cluster of machines. This enables efficient utilization of resources and helps handle increased workload demands.
- Docker allows versioning of container images, making it easier to track changes and roll back to a previous version if needed. This simplifies the deployment and update process, reducing the risk of application downtime.
- Docker has a large and active community, resulting in a vast ecosystem of pre-built container images available from Docker Hub and other registries. These images can be easily pulled and used as a base for building and deploying applications, saving development time.

The key aspects of Docker are:

- **Docker Engine:** The core component of Docker, responsible for running and managing containers. It includes the Docker daemon, which runs on the host system, and the Docker CLI, which provides a command-line interface for interacting with the daemon.
- **Docker Image:** A read-only template that contains the application code, runtime, libraries, and other dependencies required to run a container. Images are used to create containers and can be shared and reused across different environments.
- **Docker Container:** A runnable instance of a Docker image. Containers are isolated environments that run applications and their dependencies, providing consistency and portability across different systems.

- **Dockerfile:** A text file that contains instructions for building a Docker image. It specifies the base image, environment variables, commands to run, and other configuration settings needed to create the image.
- **Docker Hub:** A cloud-based registry service provided by Docker for sharing and distributing container images. It hosts a vast collection of public and private images that can be used as a base for building and deploying applications.‘

For additional information on Docker see the official website.

1.3 Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. It uses a yet another markup language (YAML) file to configure the services, networks, and volumes required for a multi-container application, making it easy to manage complex deployments. Docker Compose allows developers to define a multi-container application in a single file, then use the CLI to create and start all the services defined in the file with a single command. This simplifies the process of managing the deployment and scaling of microservices, as well as the configuration of the networks and volumes needed to run the application. Docker Compose also provides a number of commands for managing the lifecycle of the application, such as starting, stopping, and scaling the services, as well as viewing the status of the running containers. This makes it easy to develop, test, and deploy multi-container applications, and to manage the dependencies between the different services that make up the application. Key features of Docker Compose include:

- **Multi-container Applications** It helps you define and manage applications composed of multiple interacting Docker containers. Each container can represent a different service within your application.
- **Simplified Configuration** By using a docker-compose.yaml file, you specify the configuration for each service, including the image to use, ports, volumes, networks, and environment variables. This centralizes configuration and makes it easier to manage complex applications.
- **Streamlined Workflow** Docker Compose provides commands to easily build, run, stop, and scale your multi-container application. You can manage all your services with a single command instead of managing individual Docker commands for each container.
- **Declarative Syntax** The docker-compose.yaml file uses a declarative syntax, meaning you specify the desired state of your application (services, configurations), and Docker Compose takes care of creating and managing the containers to achieve that state.
- **Dependency Management** Docker Compose automatically handles dependencies between services. If one service depends on another, Compose ensures the dependent service starts only after the required service is up and running.

For additional information on Docker Compose see the official website.

1.4 RAILS Docker Implementation

Docker based components are the partial realization of RAILS software model of an automated system. These components are the micro-services that are conatinerized and typically run on a

PC (Linux, Mac, or Windows) or Single Board Computer running Linux.

Table 1.1 lists the Docker images available from the Docker Hub repository. These images are the microservices depicted in Figure 1.1.

Table 1.1: Docker Images Table

Image	Name	Port	Version	Date
	——————— SPAs ———————			
dbristow/mppm	**M**odel **P**rojects and **P**urchase **M**anager	3008	4.0.9	2026-02-02
dbristow/mrim	**M**odel **R**ailroad **I**nventory **M**anager	3001	5.3.5	2026-02-02
dbristow/mrlm	**M**odel **R**ailroad **L**ayout **M**anager	3004	4.0.9	2026-02-02
dbristow/rsrm	**R**olling**s**tock **R**FID **M**anager	3002	6.0.9	2026-02-02
	——————— Data Services ———————			
dbristow/mrfm	**M**odel **R**ailroad **F**ile **M**anager	3030	2.6.0	2026-02-05
dbristow/ppds	**P**lans and **P**urchases **D**ata **S**ervices	3007	3.0.7	2026-02-02
dbristow/rids	**R**ailroad **I**nventory **D**ata **S**ervices	3000	3.0.7	2026-02-02
dbristow/rlds	**R**ailroad **L**ayout **D**ata **S**ervices	3006	3.0.9	2026-02-02
	——————— IoT Services ———————			
dbristow/ipls	**I**ot **P**ublisher **T**urnout **P**anel **L**ight **S**ervices	3013	1.1.19	2026-02-02
dbristow/ipts	**I**ot **P**ublisher **T**urnout **S**ervices	3011	2.1.19	2026-02-02
dbristow/isbs	**I**ot **S**ubscriber **T**urnout **P**anel **B**utton **S**ervices	3012	2.0.6	2026-02-02
dbristow/isms	**I**ot **S**ubscriber **M**icro-controller **S**ervices		3.0.18	2026-02-02
dbristow/isrs	**I**ot **S**ubscriber **R**FID **S**ervices	3005	2.0.6	2026-02-02
dbristow/ists	**I**ot **S**ubscriber **T**urnout **S**ervices	3010	2.0.6	2026-02-02

The RAILS Docker implementation is a multi-container application that uses Docker Compose to define and run the services required for the RAILS application. The Docker Compose file specifies the services, networks, and volumes needed to run the RAILS application, making it easy to manage the deployment and scaling of the microservices.

Chapter 2

Setup and Run

2.1 Docker Installation

The RAILS SPAs are implemented as Docker containers. The Docker containers are built and pushed to Docker Hub. The Docker containers are then pulled from Docker Hub and run on the host machine. The host machine must have Docker installed. Docker is a platform for developing, shipping, and running applications in containers. Docker can be installed on Windows, macOS, and Linux. The installation instructions for Docker can be found at https://docs.docker.com/get-docker/.

2.2 Docker Compose Installation

Docker Compose is a tool for defining and running multi-container Docker applications. Docker Compose uses a YAML file to configure the application's services. The installation instructions[1] for Docker Compose can be found at https://docs.docker.com/compose/install/. The Docker Compose file for the RAILS SPAs is shown in Appendix A.

2.3 Broker Configuration

The RAILS SPAs use the Mosquitto broker to communicate with each other. The Mosquitto broker is an open-source message broker that implements the MQTT protocol. The Mosquitto broker must be configured to allow the RAILS SPAs to communicate with each other. The Mosquitto broker configuration file must be edited to allow the RAILS SPAs to communicate with each other. The configuration file is found in the virtual storage whose volume name is "mosquitto". To establish and modify the configuration file the following steps are taken:

1. Create a folder/directory "RAILS" and open a terminal from that folder/directory (all docker commands should be run in that terminal).
2. Create a Docker volume for the Mosquitto broker.

```
docker volume create --name mosquitto
```

[1] Note that for Microsoft Windows the installation of the GUI Docker Desktop automatically installs Docker Compose.

3. Run a Docker container for the first time the Mosquitto broker, which will create the initial configuration file.

```
docker run -it --name myMqttBrkr -p 1883:1883 -p 9001:9001 --rm
-v mosquitto:/mosquitto -d eclipse-mosquitto
```

4. Stop the broker container.

```
docker stop myMqttBrkr
```

5. Locate the configuration file in the ”mosquitto” volume.

```
docker inspect mosquitto
```

6. Edit the configuration file modifying or adding the following lines:

```
listener 1883
allow_anonymous true
socket_domain ipv4
```

2.4 Create the RAILS Docker Environment

The RAILS SPAs are implemented as Docker containers that require an environment to run in. To create that environment, the following steps are taken:

1. Create a Docker volume for the Rails database.

```
docker volume create --name myRailsDb
```

2. Create a Docker volume for the Rails images.

```
docker volume create --name myRailsImages
```

3. Create a Docker network for the Rails containers to communicate over.

```
docker network create myRailsNet
```

2.5 Run the RAILS Docker Containers

The Docker containers are pulled from Docker Hub and run on the host machine. To run all of the RAILS Docker containers, either create the YAML file from Appendix A or retrieve the YAML file from the RAILS GitHub repository found at my GitHub repository. Additionally create the "nginx.conf" file from Appendix B or retreive it from the RAILS GitHub repository. Be sure that both files are put into the "RAILS" folder/directory. To fetch the images, create the containers, and start the conatiners running execute the following command in the terminal:

```
docker compose up -d
```

This command will pull the Docker images from Docker Hub and run the containers in detached mode. The RAILS SPAs will be running in the background, and the terminal will return to the command prompt. The RAILS SPAs will be running on the host machine. Point a browser, on the host machine, to the following URLs to access the RAILS SPAs:

- MRIM SPA - http://localhost/mrim (or `http://<MACHINE_IP>/mrim/`)
- RSRM SPA - http://localhost/rsrm (or `http://<MACHINE_IP>/rsrm/`)
- MPPM SPA - http://localhost/mppm (or `http://<MACHINE_IP>/mppm/`)
- MRLM SPA - http://localhost/mrlm (or `http://<MACHINE_IP>/mrlm/`)

Where <MACHINE_IP> is the IP address of the host machine running the Docker containers. The RAILS SPAs can be accessed from any browser on any Personal Computer (PC) in the network.

2.6 Stop and Remove the RAILS Docker Containers

To stop and remove the RAILS Docker containers, the following command is used:

```
docker compose down
```

See Appendix C for screenshots of the Docker commands used to create the Docker environment and run the RAILS SPAs.

Appendix A

YAML File

```yaml
# docker-compose.yaml
# &copy; David Bristow, 2020-2026
# VERSION 2.0.0
# LICENSE
# The code in this repository is licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

services:
  # Nginx Reverse Proxy Service
  nginx:
    image: nginx:latest # Use the official Nginx image
    container_name: myNginxProxy
    ports:
      - '80:80' # Expose Nginx on host port 80 (for HTTP)
    volumes:
      # Mount your custom nginx.conf into the container
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    networks:
      - myRailsNet # Connect Nginx to your internal network
    depends_on:
      # Nginx should start after all services it proxies to
      - myMongo
      - myPpds
      - myMppm
      - myRids
      - myMrfm
      - myMrim
      - myMqttBrkr
```

```
      - myRlds
      - myIsrs
      - myIsms
      - myRsrm
      - myIsts
      - myIsbs
      - myIpts
      - myIpls
      - myMrlm

  myMongo:
    image: 'mongo'
    container_name: myMongo
    networks:
      - myRailsNet
    volumes:
      - myRailsDb:/data/db

  myPpds:
    image: 'dbristow/ppds'
    container_name: myPpds
    networks:
      - myRailsNet
    depends_on:
      - myMongo

  myMppm:
    image: 'dbristow/mppm'
    container_name: myMppm
    networks:
      - myRailsNet # Add to network so Nginx can find it
    depends_on:
      - myPpds

  myRids:
    image: 'dbristow/rids'
    container_name: myRids
    networks:
      - myRailsNet
    depends_on:
      - myMongo

  myMrfm:
    image: 'dbristow/mrfm'
    # Environment variables for myMrfm's *backend* if it has one.
    # Frontend VITE_ variables are handled at image build time.
    environment:
      # These might be for internal communication *within* myMrfm's backend,
      # or if myMrfm's backend needs to call other services.
      # If myMrfm is purely a static file server, these might not be needed.
      - MYMRIM_URI1=http://myMrim:8080 # Use service name for internal Docker
          network
```

```
      - MYMRIM_URI2=http://myMrim:8080 # Use service name for internal Docker network
      - ALLOWED_CONTAINER_ORIGIN=http://myMrim
    container_name: myMrfm
    volumes:
      - myRailsImages:/usr/djb/src/uploads
    networks:
      - myRailsNet # Add to network so Nginx can find it

  myMrim:
    image: 'dbristow/mrim'
    container_name: myMrim
    networks:
      - myRailsNet
    depends_on:
      - myRids
      - myMrfm

  myMqttBrkr:
    image: 'eclipse-mosquitto'
    container_name: myMqttBrkr
    networks:
      - myRailsNet
    # Keep MQTT ports exposed if external clients need to connect directly
    ports:
      - '1883:1883'
      - '9001:9001'
    volumes:
      - mosquitto:/mosquitto

  myRlds:
    image: 'dbristow/rlds'
    container_name: myRlds
    networks:
      - myRailsNet
    depends_on:
      - myMongo

  myIsrs:
    image: 'dbristow/isrs'
    container_name: myIsrs
    networks:
      - myRailsNet
    depends_on:
      - myMqttBrkr

  myIsms:
    image: 'dbristow/isms'
    container_name: myIsms
    networks:
      - myRailsNet
    depends_on:
      - myMqttBrkr
```

```
      - myRlds

  myRsrm:
    image: 'dbristow/rsrm'
    container_name: myRsrm
    networks:
      - myRailsNet
    depends_on:
      - myRids
      - myRlds
      - myIsrs
      - myIsms

  myIsts:
    image: 'dbristow/ists'
    container_name: myIsts
    networks:
      - myRailsNet
    depends_on:
      - myMqttBrkr

  myIsbs:
    image: 'dbristow/isbs'
    container_name: myIsbs
    networks:
      - myRailsNet
    depends_on:
      - myMqttBrkr

  myIpts:
    image: 'dbristow/ipts'
    container_name: myIpts
    networks:
      - myRailsNet
    depends_on:
      - myMqttBrkr

  myIpls:
    image: 'dbristow/ipls'
    container_name: myIpls
    networks:
      - myRailsNet
    depends_on:
      - myMqttBrkr

  myMrlm:
    image: 'dbristow/mrlm'
    container_name: myMrlm
    networks:
      - myRailsNet
    depends_on:
      - myRlds
      - myIsts
```

```
194       - myIsbs
195       - myIpts
196       - myIpls
197
198 networks:
199   myRailsNet:
200     # It's good practice to define the network as internal for security,
201     # unless you explicitly need containers outside this compose file to join
          it.
202     # If `myRailsNet` is truly external (created manually with `docker network
          create`),
203     # then keep `external: true`. Otherwise, use `driver: bridge`.
204     external: true # Keep if you've already created it with `docker network
          create myRailsNet`
205     # driver: bridge # Use this if you want Docker Compose to manage its
          creation
206
207 volumes:
208   myRailsDb:
209     external: true
210   myRailsImages:
211     external: true
212   mosquitto:
213     external: true
```

Listing A.1: docker compose file

Appendix B

NGINX Configuration

```
# nginx.conf (for the Nginx Reverse Proxy container)

worker_processes auto;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;

    sendfile on;
    keepalive_timeout 65;

    # Log format for better debugging
    log_format combined_proxy '$remote_addr - $remote_user [$time_local] '
                              '"$request" $status $body_bytes_sent '
                              '"$http_referer" "$http_user_agent" '
                              '"$http_x_forwarded_for" "$upstream_addr"';

    access_log /var/log/nginx/access.log combined_proxy;
    error_log /var/log/nginx/error.log warn; # Set to info or debug for more
       verbose logging

    server {
        listen 80;
        server_name localhost; # Use your public domain name here in
           production (e.g., yourmodelrailroad.com)

        # --- Frontend Applications (SPAs) ---
        # Each SPA will be served from its own base path, e.g., /mrim/, /mppm/
        # The rewrite rule removes the path prefix before forwarding to the
           SPA's internal server.

        # myMrim Frontend
        location /mrim/ {
            rewrite ^/mrim/(.*)$ /$1 break; # Remove /mrim/ prefix
```

```
36              proxy_pass http://myMrim:8080; # myMrim service, internal port
                    8080
37              proxy_set_header Host $host;
38              proxy_set_header X-Real-IP $remote_addr;
39              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
40              proxy_set_header X-Forwarded-Proto $scheme;
41          }
42
43          # myMppm Frontend
44          location /mppm/ {
45              rewrite ^/mppm/(.*)$ /$1 break;
46              proxy_pass http://myMppm:8080; # myMppm service, internal port
                    8080
47              proxy_set_header Host $host;
48              proxy_set_header X-Real-IP $remote_addr;
49              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
50              proxy_set_header X-Forwarded-Proto $scheme;
51          }
52
53          # myMrlm Frontend
54          location /mrlm/ {
55              rewrite ^/mrlm/(.*)$ /$1 break;
56              proxy_pass http://myMrlm:8080; # myMrlm service, internal port
                    8080
57              proxy_set_header Host $host;
58              proxy_set_header X-Real-IP $remote_addr;
59              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
60              proxy_set_header X-Forwarded-Proto $scheme;
61          }
62
63          # myRsrm Frontend
64          location /rsrm/ {
65              rewrite ^/rsrm/(.*)$ /$1 break;
66              proxy_pass http://myRsrm:8080; # myRsrm service, internal port
                    8080
67              proxy_set_header Host $host;
68              proxy_set_header X-Real-IP $remote_addr;
69              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
70              proxy_set_header X-Forwarded-Proto $scheme;
71          }
72
73          # --- Backend API Services ---
74          # myRids API (e.g., accessible at http://yourdomain.com/api/rids/
                rslistall)
75          location /api/rids/ {
76              rewrite ^/api/rids/(.*)$ /$1 break; # Remove /api/rids/ prefix
77              proxy_pass http://myRids:3000; # myRids service, internal port
                    3000
78              proxy_set_header Host $host;
79              proxy_set_header X-Real-IP $remote_addr;
80              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
81              proxy_set_header X-Forwarded-Proto $scheme;
82          }
```

```
83
84          # myRlds API (e.g., accessible at http://yourdomain.com/api/rlds/
                some_endpoint)
85          location /api/rlds/ {
86              rewrite ^/api/rlds/(.*)$ /$1 break; # Remove /api/rlds/ prefix
87              proxy_pass http://myRlds:3006; # myRlds service, internal port
                    3006
88              proxy_set_header Host $host;
89              proxy_set_header X-Real-IP $remote_addr;
90              proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
91              proxy_set_header X-Forwarded-Proto $scheme;
92          }
93
94          # myPpds API (e.g., accessible at http://yourdomain.com/api/ppds/
                some_endpoint)
95          location /api/ppds/ {
96              rewrite ^/api/ppds/(.*)$ /$1 break; # Remove /api/ppds/ prefix
97              proxy_pass http://myPpds:3007; # myPpds service, internal port
                    3007
98              proxy_set_header Host $host;
99              proxy_set_header X-Real-IP $remote_addr;
100             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
101             proxy_set_header X-Forwarded-Proto $scheme;
102         }
103
104         # myMrfm Image Server (e.g., accessible at http://yourdomain.com/
                images/mikado.png)
105         location /images/ {
106             rewrite ^/images/(.*)$ /$1 break; # Remove /images/ prefix
107             proxy_pass http://myMrfm:3030; # myMrfm service, internal port
                    3030
108             proxy_set_header Host $host;
109             proxy_set_header X-Real-IP $remote_addr;
110             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
111             proxy_set_header X-Forwarded-Proto $scheme;
112         }
113
114         # --- MQTT-Interfacing API Services ---
115         # myIsrs API3005
116         location /api/isrs/ {
117             rewrite ^/api/isrs/(.*)$ /$1 break;
118             proxy_pass http://myIsrs:3005;
119
120             proxy_http_version 1.1;
121             proxy_set_header Upgrade $http_upgrade;
122             proxy_set_header Connection "upgrade";
123             proxy_set_header Host $host;
124             proxy_set_header X-Real-IP $remote_addr;
125             proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
126             proxy_set_header X-Forwarded-Proto $scheme;
127         }
128
129
```

```
        # myIpls API and Socket.IO internal port 3013
        location /api/ipls/ {
            rewrite ^/api/ipls/(.*)$ /$1 break;
            proxy_pass http://myIpls:3013;

            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # myIpts API and Socket.IO internal port 3011
        location /api/ipts/ {
            rewrite ^/api/ipts/(.*)$ /$1 break;
            proxy_pass http://myIpts:3011;

            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # myIsbs API and Socket.IO internal port 3012
        location /api/isbs/ {
            rewrite ^/api/isbs/(.*)$ /$1 break;
            proxy_pass http://myIsbs:3012;

            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # myIsms API and Socket.IO internal port
        location /api/isms/ {
            rewrite ^/api/isms/(.*)$ /$1 break;
            proxy_pass http://myIsms:8080; # Assuming internal port 8080 (
                common for Node/Java)
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

```

```
        # myIsts API and Socket.IO internal port 3010
        location /api/ists/ {
            rewrite ^/api/ists/(.*)$ /$1 break;
            proxy_pass http://myIsts:3010;

            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection "upgrade";
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_set_header X-Forwarded-Proto $scheme;
        }

        # --- Default/Root Fallback ---
        # If you have a primary SPA that should load at the root (e.g.,
           yourmodelrailroad.com/),
        # you would put its proxy_pass here. Otherwise, this serves a default
           Nginx page.
        location / {
            root /usr/share/nginx/html; # Default Nginx welcome page or static
                content
            index index.html index.htm;
            try_files $uri $uri/ =404;
        }

        # Error pages (optional)
        error_page 500 502 503 504 /50x.html;
        location = /50x.html {
            root /usr/share/nginx/html;
        }
    }
}
```

Listing B.1: nginx configuration file

Appendix C

Example Setup and Run Commands

The following screenshots show the Docker commands used to create the Docker environment and run the RAILS Docker containers.

C.1 Docker in a Windows Environment

Docker on Windows 11 relies on a component called Windows Subsystem for Linux 2 (WSL 2) to provide a Linux-like environment for running Docker containers. Docker Desktop installer provides two options:

- Use WSL 2 backend: This is the recommended approach for most users. It leverages WSL 2 to create a Linux environment for Docker to run containers.
- Use Hyper-V backend: This is an alternative that uses Hyper-V virtualization technology. However, WSL 2 is generally considered more performant and resource-efficient.

This setup is based on using WSL 2 as it:

- offers better performance for running Linux containers compared to Hyper-V.
- provides a more native Linux experience for Docker, ensuring compatibility with most Linux container images.
- uses the Windows kernel directly, leading to more efficient resource usage.

During Docker Desktop installation with the WSL 2 backend, Docker will automatically configure and set up WSL 2 if it's not already installed. It will also create a default Linux distribution (usually Ubuntu) within WSL 2. Once WSL 2 is set up, Docker Desktop installs the Docker daemon within the WSL 2 environment. This daemon is the core process that manages Docker containers. Docker Desktop provides a Docker CLI for Windows. This CLI interacts with the Docker daemon running in the WSL 2 environment through a remote API. So, when you run Docker commands on Windows, they are translated and sent to the Docker daemon in WSL 2 for execution.

In essence, Docker on Windows 11 leverages WSL 2 to create a Linux-like environment for running Docker containers. This approach provides a powerful and performant solution for containerization on Windows, while still offering a familiar Windows experience for interacting with Docker.

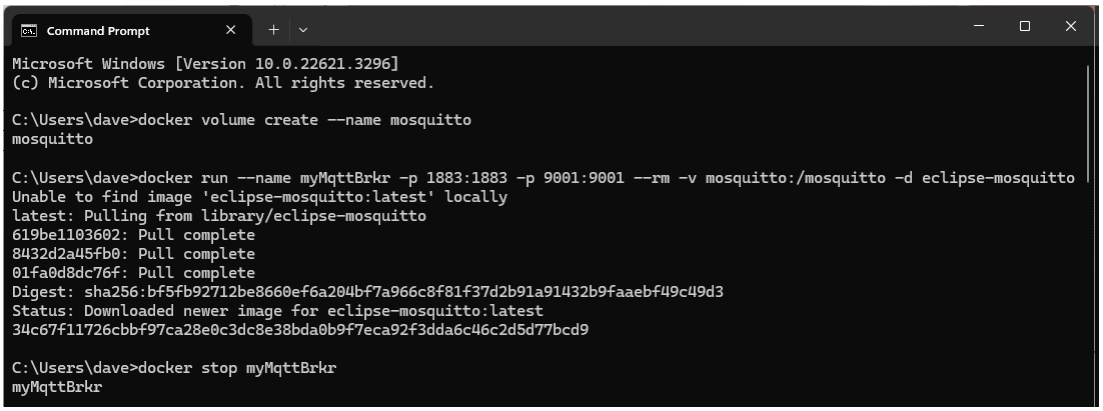

Figure C.1: Windows Docker mosquitto Setup

Docker Desktop with WSL 2 backend stores Docker images and volumes on the Windows file system, accessible from both Windows and WSL 2. This allows for easy sharing of container data between the two environments. However, it simpler to use Docker Desktop GUI to locate and edit the mosquitto configuration file. Figure C.2 shows the volumes created in the Windows environment.

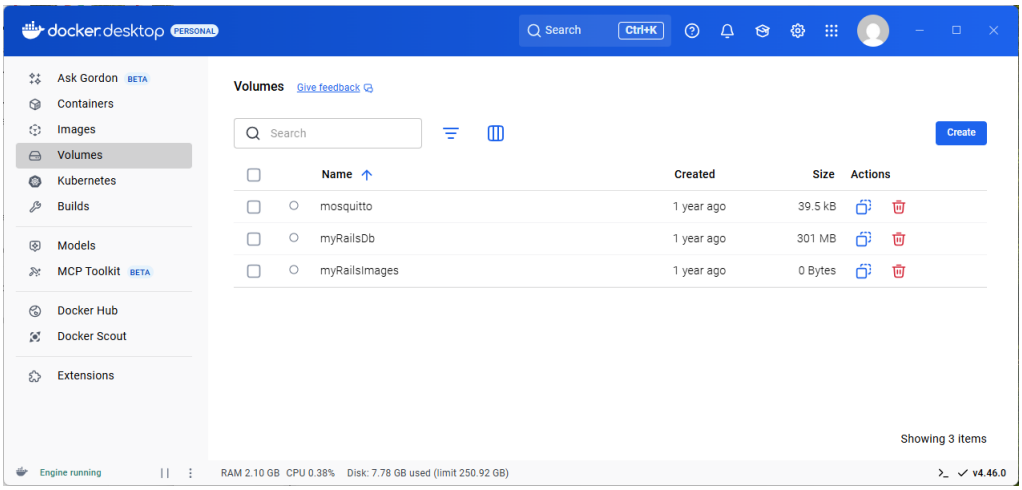

Figure C.2: Docker Desktop Volumes

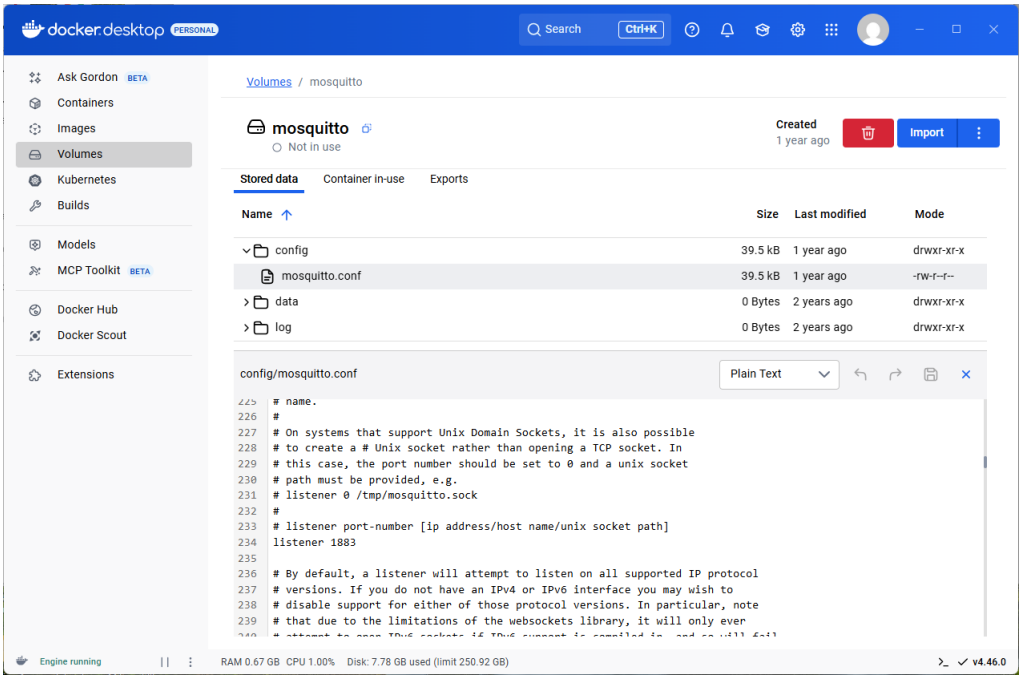

Figure C.3: Mosquitto Configuration File

The following screenshots show the Docker commands used to create the Docker environment and run all of the RAILS Docker containers.

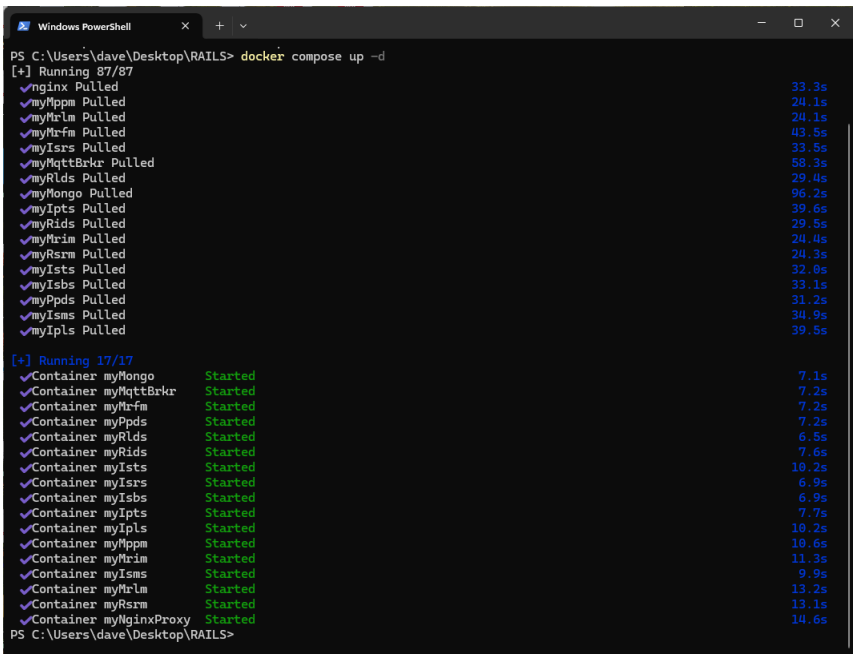

Figure C.4: Windows Docker Compose Up

The following screenshots confirm the Docker containers are running.

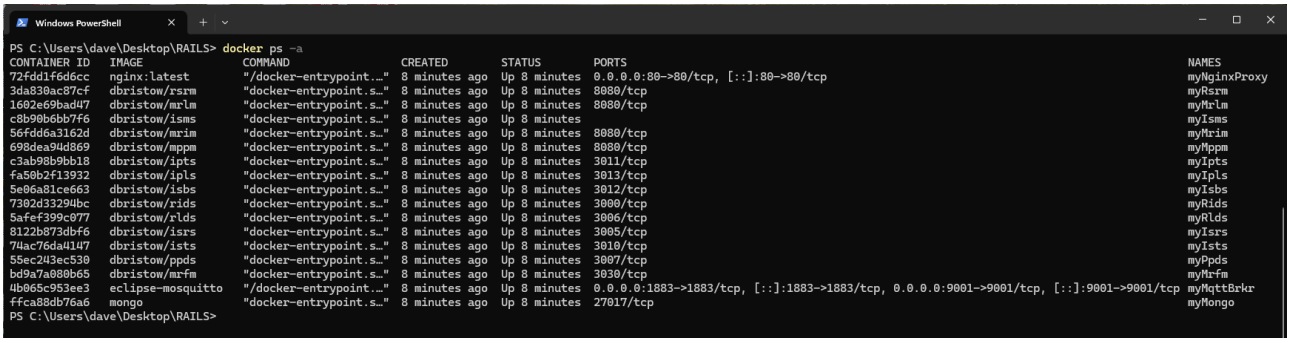

Figure C.5: Windows Docker PS

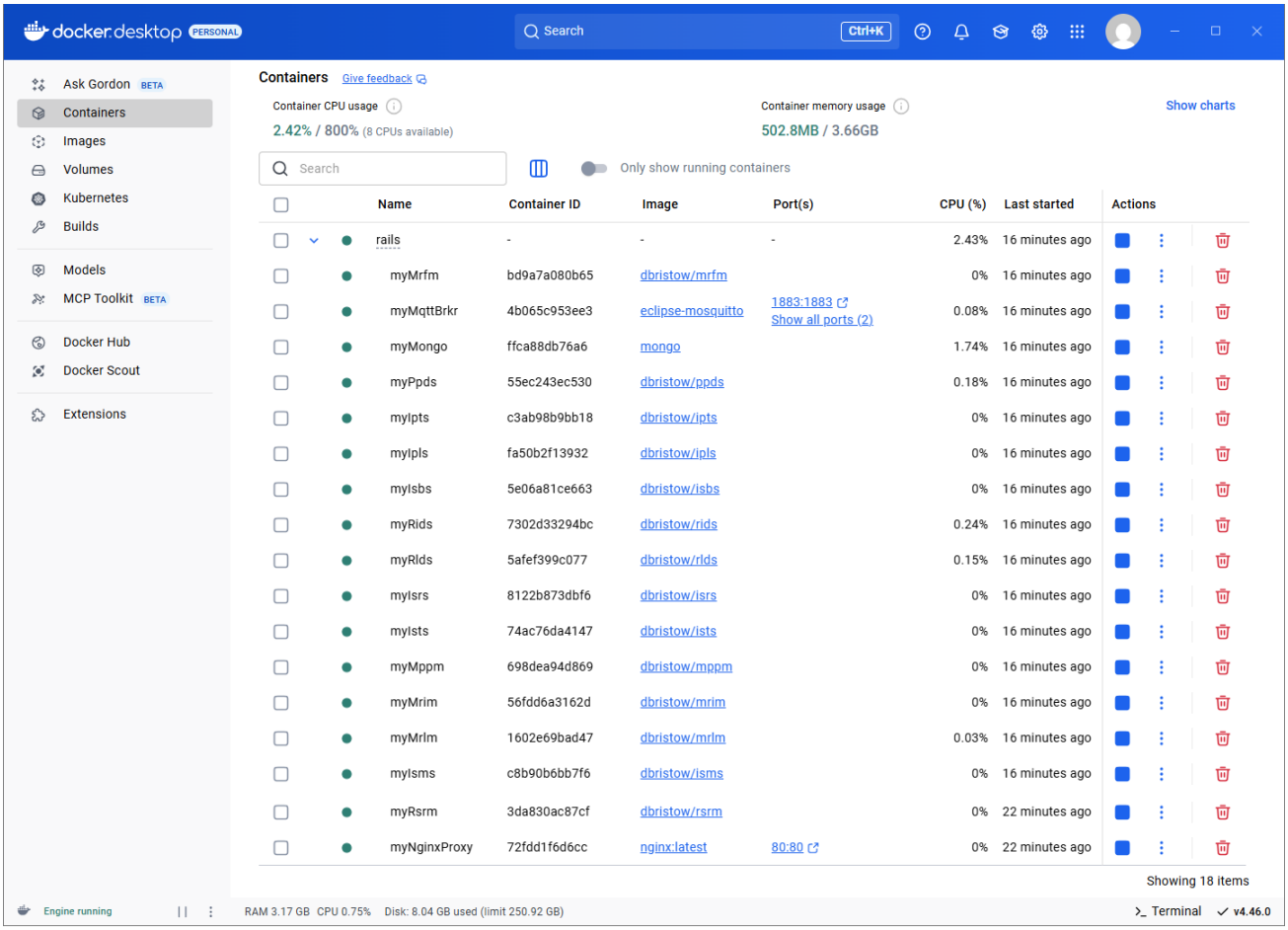

Figure C.6: Windows Docker Desktop Containers

Once the Docker containers are running, the RAILS SPAs can be accessed from a web browser (Chrome). The following screenshots show the RAILS SPAs running in a web browser.

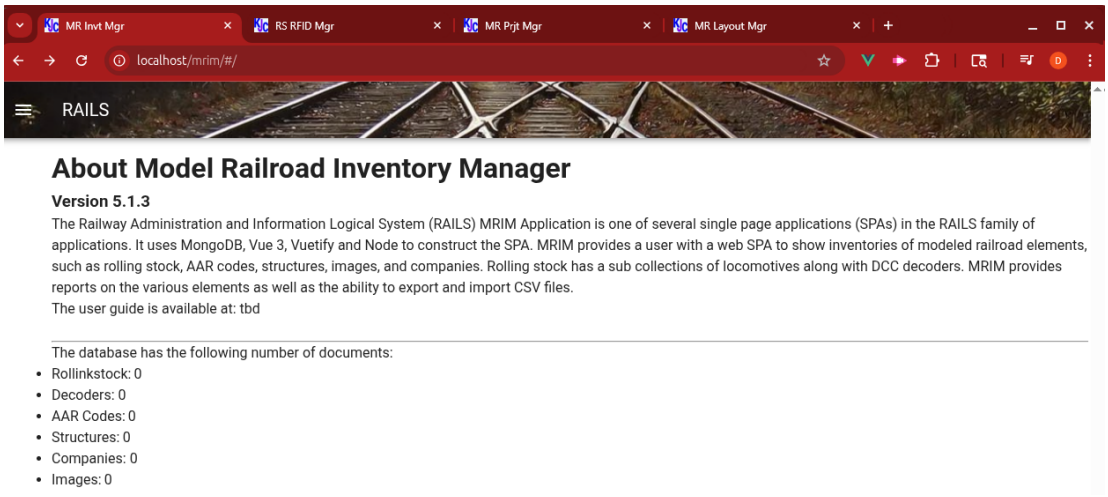

Figure C.7: RAILS MRIM Running

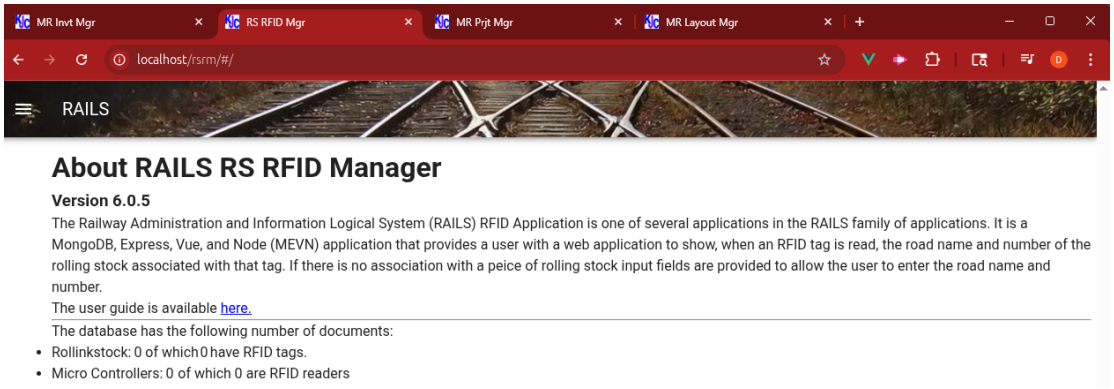

Figure C.8: RAILS RSRM Running

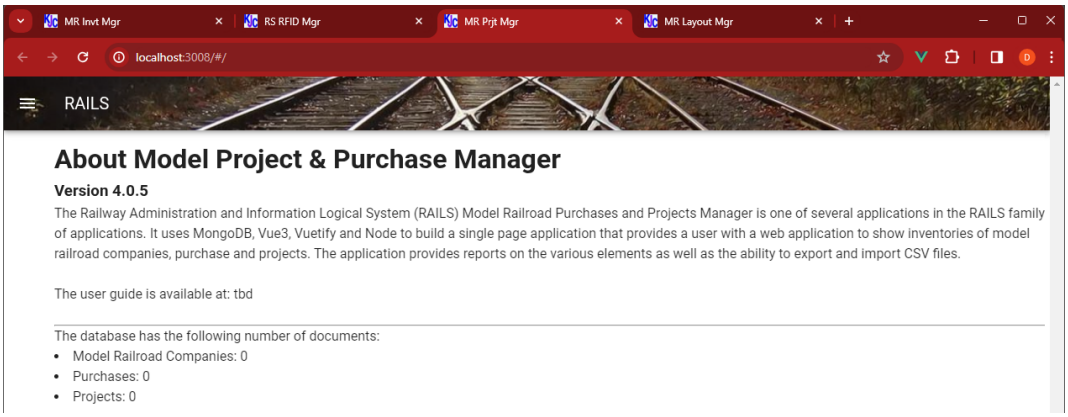

Figure C.9: RAILS MPPM Running

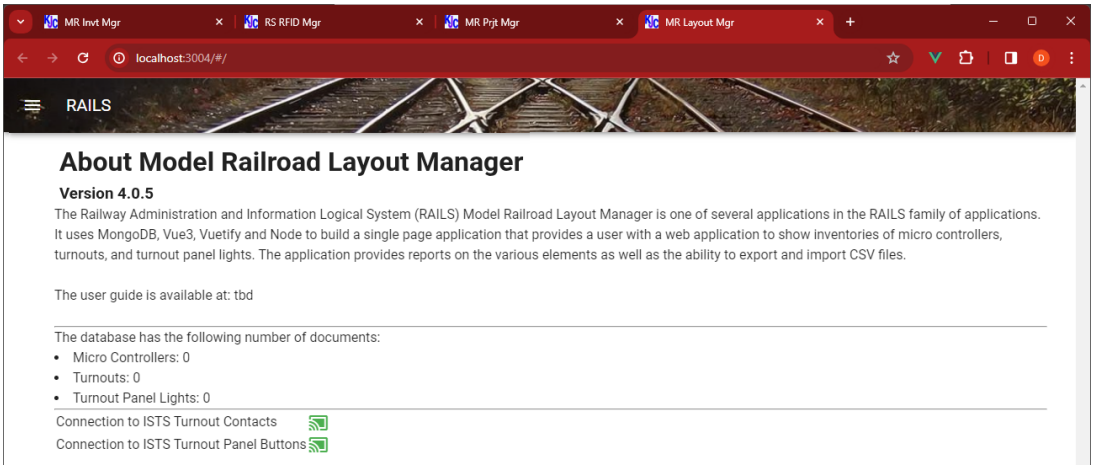

Figure C.10: RAILS MRLM Running

The following screenshot shows the stopping and removal of the RAILS Docker containers in a Windows environment.

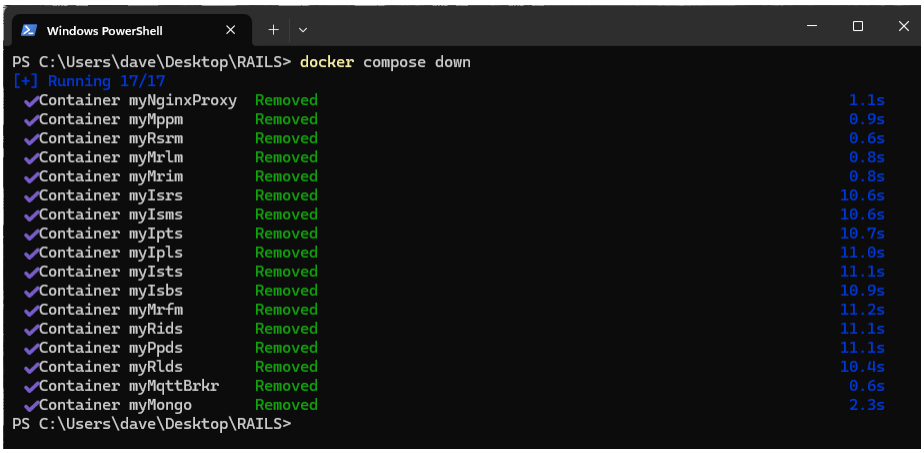

Figure C.11: Windows Docker Compose Down

C.2 Docker in a Linux Environment

The following screenshot shows the initial Docker commands used to create the Docker environment in a Linux environment. The yellow highlighted text represents the edits of the mosquitto configuration file using the vi editor.

```
root@kingston:/SoftwareDev/RAILS/Docker Based# docker volume create --name mosquitto
mosquitto
root@kingston:/SoftwareDev/RAILS/Docker Based# docker run -it --name myMqttBrkr -p 1883:1883 -p
9001:9001 --rm -v mosquitto:/mosquitto -d eclipse-mosquitto
919d31e8f767810ac4dfd0ce2152d8c7f1d246434d735453fc12771538461ddc
root@kingston:/SoftwareDev/RAILS/Docker Based# docker stop myMqttBrkr
myMqttBrkr
root@kingston:/SoftwareDev/RAILS/Docker Based# docker inspect mosquitto
[
    {
        "CreatedAt": "2024-03-30T09:37:18-06:00",
        "Driver": "local",
        "Labels": null,
        "Mountpoint": "/var/lib/docker/volumes/mosquitto/_data",
        "Name": "mosq",
        "Options": null,
        "Scope": "local"
    }
]
root@kingston:/SoftwareDev/RAILS/Docker Based# cd /var/lib/docker/volumes/mos/_data/config
root@kingston:/var/lib/docker/volumes/mos/_data/config# ls
mosquitto.conf
root@kingston:/var/lib/docker/volumes/mos/_data/config# vi mosquitto.conf
# ================================================================
# Listeners
# ================================================================
#
# On systems that support Unix Domain Sockets, it is also possible
# to create a # Unix socket rather than opening a TCP socket. In
# this case, the port number should be set to 0 and a unix socket
# path must be provided, e.g.
# listener 0 /tmp/mosquitto.sock
#
# listener port-number [ip address/host name/unix socket path]
listener 1883

# Set to `ipv4` to force the listener to only use IPv4, or set to `ipv6` to
# force the listener to only use IPv6. If you want support for both IPv4 and
# IPv6, then do not use the socket_domain option.
#
socket_domain ipv4

# ================================================================
# Security
# ================================================================
# Boolean value that determines whether clients that connect
# without providing a username are allowed to connect. If set to
# false then a password file should be created (see the
# password_file option) to control authenticated client access.
#
# Defaults to false, unless there are no listeners defined in the configuration
# file, in which case it is set to true, but connections are only allowed from
# the local machine.
allow_anonymous true

root@kingston:/SoftwareDev/RAILS/Docker Based# docker volume create --name myRailsDb
myRailsDb
root@kingston:/SoftwareDev/RAILS/Docker Based# docker volume create --name myRailsImages
myRailsImages
root@kingston:/SoftwareDev/RAILS/Docker Based# docker network create myRailsNet
cd6b2f49e1024dab42a7f28e85e00539f920fdfca8637fa8d6183bc42dc9e4ac
root@kingston:/SoftwareDev/RAILS/Docker Based#
```

Figure C.12: Docker Setup mosquitto

The following screenshots show the Docker command used to pull images and run the RAILS Docker containers in a Linux environment. The process starts by checking for images in the local repository, not finding any Docker will pull them from Docker Hub, the progress is displayed. Once all of the images are pulled, as seen in the top half of figure C.13 Docker will then start each container. The bottom half of half of figure C.13 shows the successful creation of all of the running containers.

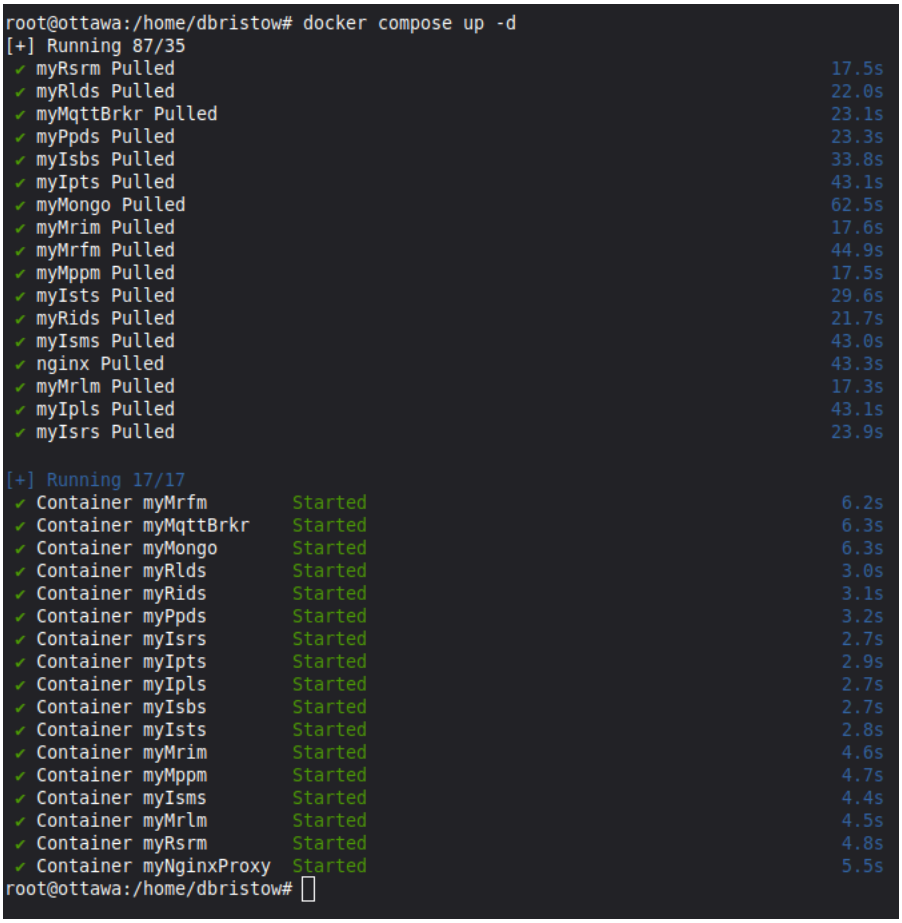

Figure C.13: Linux Docker-Compose Up

The following screenshot shows a RAILS SPAs running in a web browser (Chrome) in a Linux environment. The remaining SPAs can be accessed in the same manner.

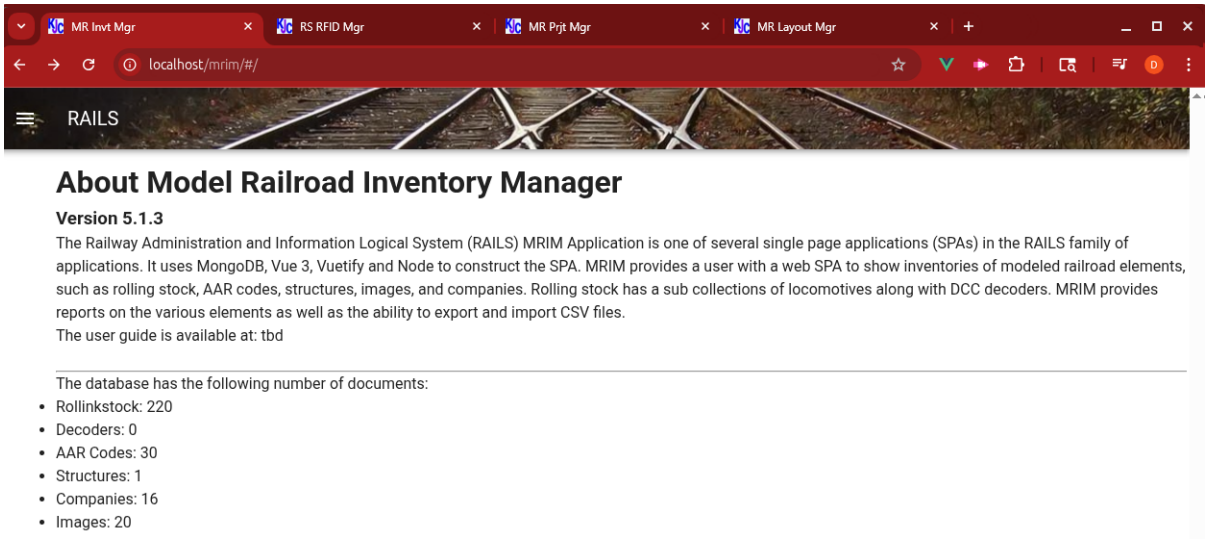

Figure C.14: RAILS MRIM Running

The following screenshot shows the stopping and removal of the RAILS Docker containers in a Linux environment.

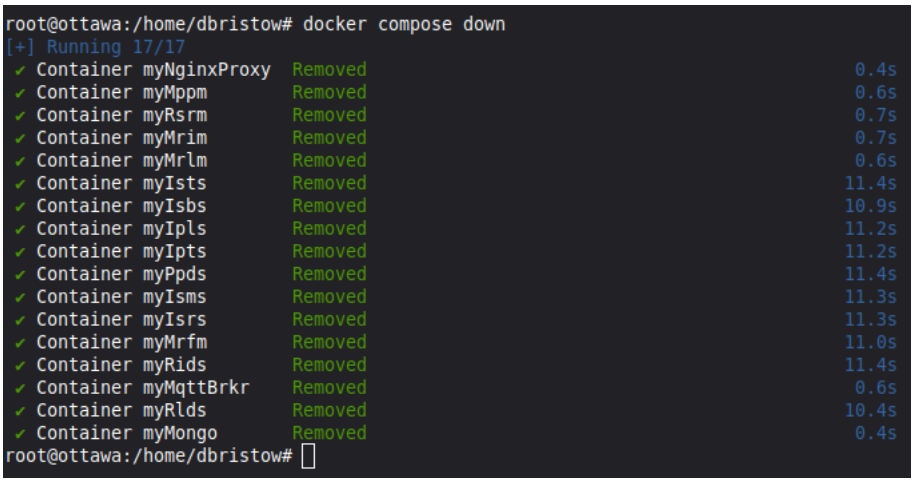

Figure C.15: Linux Docker-Compose Down

C.3 Docker in a MacOS Environment

Docker itself doesn't run directly on MacOS 14 (Mojave). This is because Docker relies on a Linux kernel for containerization, and MacOS uses a different kernel. Similar to Windows there is a Docker Desktop software package that provides a complete Docker experience on MacOS. It includes:

- Docker Desktop installs and runs a lightweight Linux VM in the background. This VM typically uses LinuxKit, a minimal Linux distribution optimized for running Docker.

- The Docker engine runs within the Linux VM. This engine is responsible for managing Docker images, containers, and networks.
- Docker Desktop provides a CLI for interacting with the Docker engine running in the VM. You can use familiar Docker commands to build, run, and manage containers.
- Docker Desktop offers an optional GUI that allows you to manage containers visually.

The following screenshot shows the initial Docker commands used to create the Docker environment in a MacOS environment. Like Windows it is simpler to use Docker Desktop GUI to locate and edit the mosquitto configuration file. Using the Docker Desktop to create the volumes and to edit the mosquitto configuration file, similarly to that done in Linux the result is shown in figure C.16

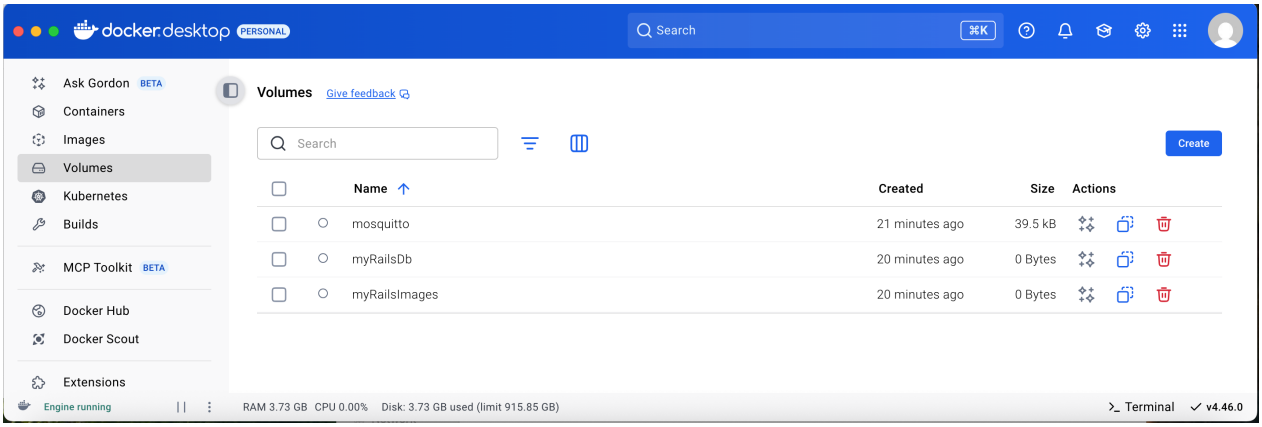

Figure C.16: Mac Docker Desktop Volumes

Open a Mac terminal to initiate Docker Compose to setup and run the RAILS container. Figure C.17 is a snapshot showing the pulling of the images from Docker Hub. Figure C.18 captures the process of pulling all of the images nearing completion. Figure C.19 shows the terminal once the “docker compose up -d” command has completed and all of teh containers are running.

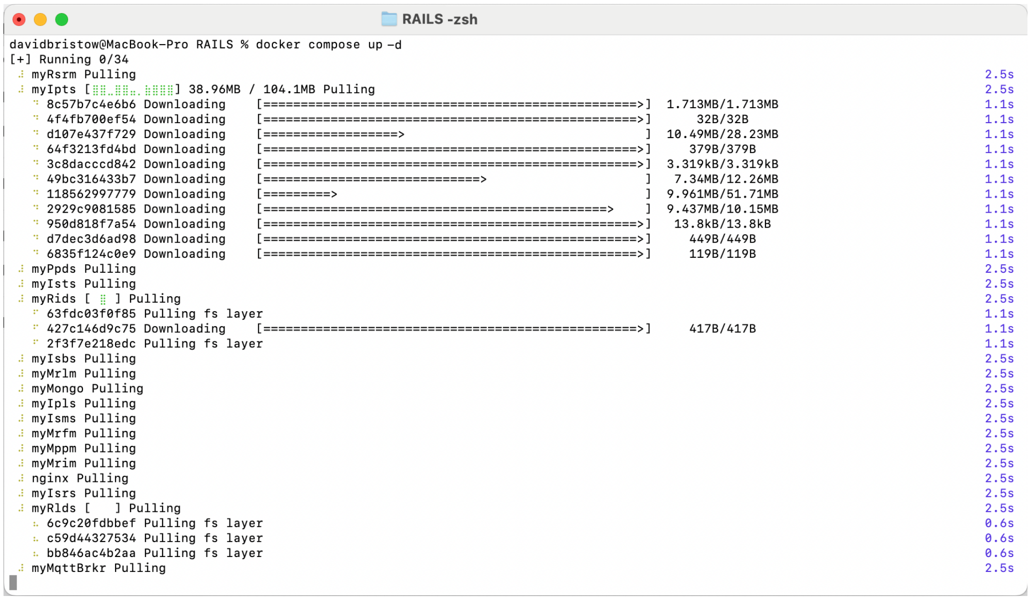

Figure C.17: MacOS Docker Compose Up Pulls

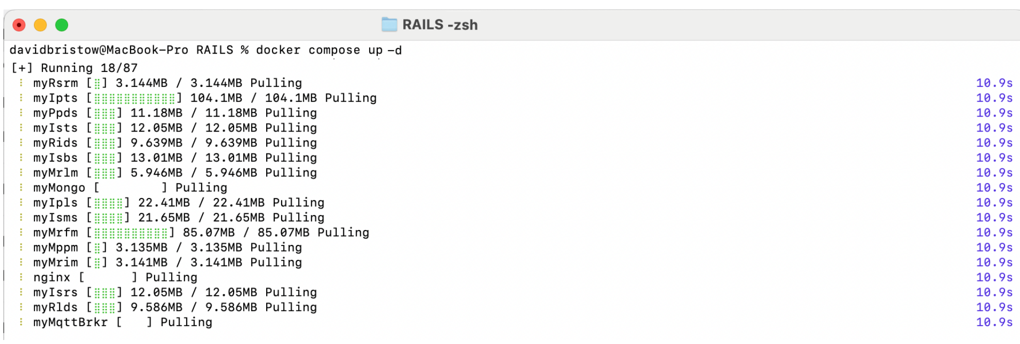

Figure C.18: MacOS Docker Compose Up Completing Pulls

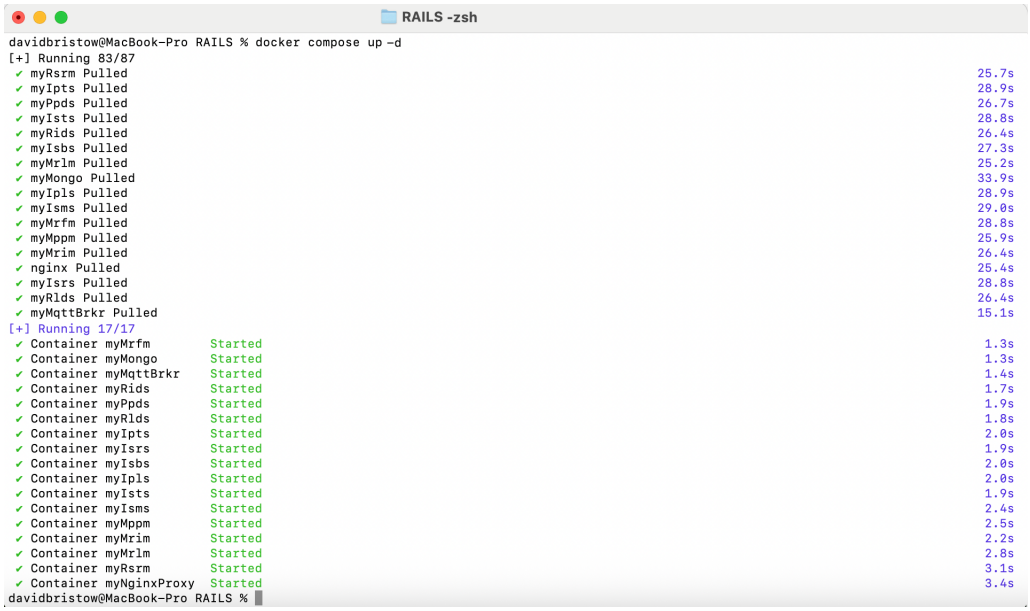

Figure C.19: MacOS Docker Compose Up Started

Figure C.20 shows in the Docker Desktop GUI that all the images are now local to the Mac.

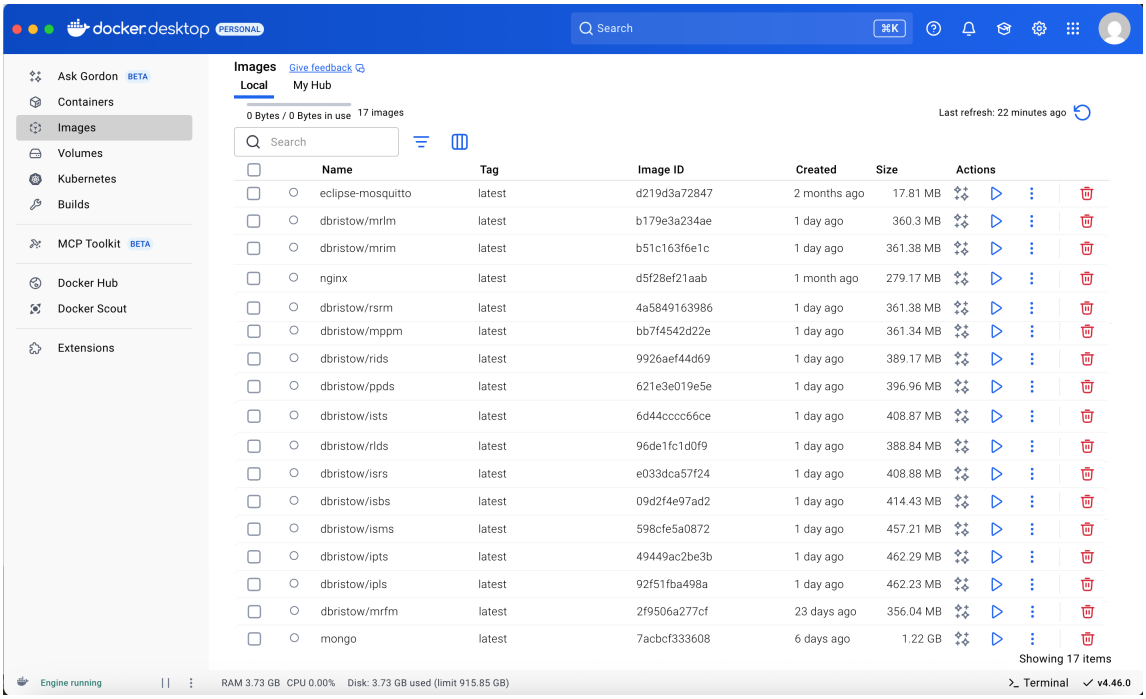

Figure C.20: MacOS Docker Desktop Images

The following screenshot shows a RAILS SPAs running in a web browser (Safari) in a MacOS environment. The remaining SPAs can be accessed in the same manner.

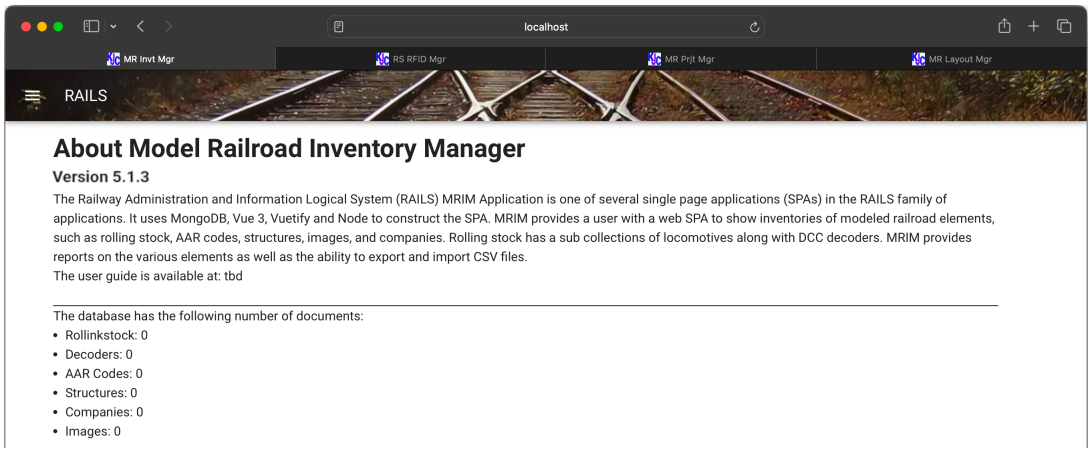

Figure C.21: RAILS MRIM Running

The following screenshot shows the stopping and removal of the RAILS Docker containers in a MacOS environment.

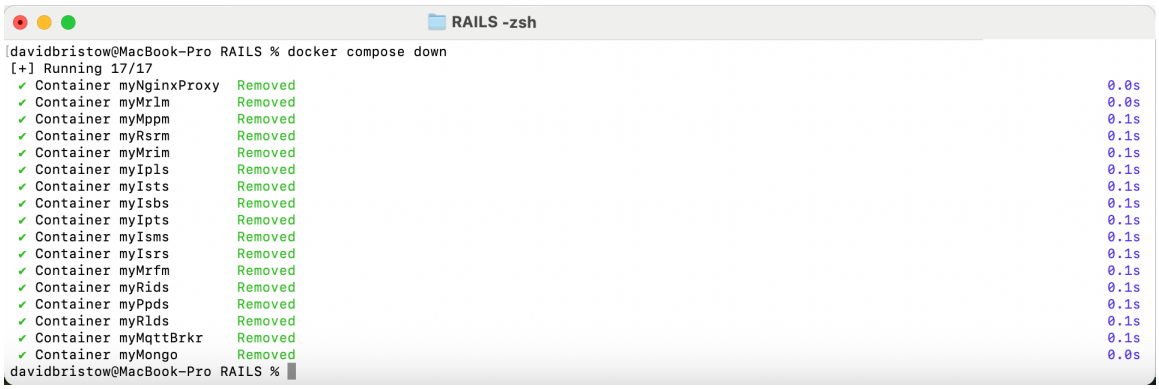

Figure C.22: MacOS DockerCompose Down

Appendix D

Docker Commands

The following Docker commands may be useful when working with Docker images and containers related to the RAILS SPAs.

- `docker ps` - List running containers.
- `docker ps -a` - List all containers including those that have been stopped.
- `docker images` - List tagged images
- `docker images ls -a` - List all images
- `docker rmi <image-id>` - Remove an image.
- `docker volume ls` - List volumes.
- `docker volume prune` - This will remove all local volumes not used by at least one container. This command is designed to clean up any volumes that are not currently attached to a container, which is a good thing to do because after running RAILS as Docker Compose creates some randomly named volumes. Be sure to do this while all of the RAILS conatiners are running otherwise the volumes: mosquitto, myRailsDb, and myRailsImages may be destroyed causing a complete lose of your data.
- `docker run -d --name <container-name> <image-name>` - Run a container from an image.
- `docker stop <container-name>` - Stop a running container.
- `docker rm <container-name>` - Remove a container.
- `docker inspect <container-name> | <volume-name>` - Display detailed information about a container or volume.
- `docker-compose up -d` - Start the services defined in the Docker Compose file (docker-compose.yaml).
- `docker-compose -f docker-compose-dev.yaml up -d` - Start the services defined in an alternate Docker Compose file.
- `docker-compose down` - Stop the services defined in the Docker Compose file.

Acronyms

Application Program Interface (API) acts as a middleman between different software applications. It defines a set of rules and specifications that allow applications to talk to each other and exchange data or functionality 4

command line interface (CLI) is a text based way to interact with a computer program by typing commands and seeing text based responses 4, 5, 6, 21, 30

Data Services (DS) is a class of RAILS components that provides an API permitting access to the MongoDB that stores the documents used by RAILS 3

Graphic User Interface (GUI) is the visual way you interact with a computer program using elements like windows, icons, menus, and buttons, instead of typing commands. GUIs are the standard way most people interact with computers today 3, 4, 8, 22, 30, 32

Internet of Things (IoT) refers to a vast network of physical devices embedded with sensors, software, and other technologies that allows them to connect and exchange data with other devices and systems over the internet or other communication networks 2, 3

IoT Publisher Turnout Panel Light Services (IPLS) 3

IoT Publisher Turnout Services (IPTS) 3

infrared (IR) is a type of electromagnetic radiation that lies just beyond the red end of the visible light spectrum. It's invisible to the human eye, but we can feel it as heat 3

IoT Subscriber Turnout Panel Button Services (ISBS) 3

IoT Subscriber Location Services (ISLS) 3

IoT Subscriber Micro-controller Services (ISMS) 3

IoT Subscriber RFID Services (ISRS) 3

IoT Subscriber Turnout Services (ISTS) 3

Model Projects and Purchase Manager (MPPM) 4, 10

Message Queuing Telemetry Transport (MQTT) is a lightweight messaging protocol designed for machine-to-machine (M2M) communication in resource-constrained environments, like those found in the Internet of Things (IoT) 3, 8

Model Railroad File Manager (MRFM) 4

Model Railroad Inventory Manager (MRIM) 4, 10

Model Railroad Layout Manager (MRLM) 3, 4, 10

Plans and Purchases Data Services (PPDS) 4

Railway Administration and Information Logical System (RAILS) 2, 3, 4, 6, 7, 8, 9, 10, 21, 23, 24, 26, 28, 29, 30, 32, 33, 34

Representational State Transfer (REST) is a set of architectural principles for designing web services. It's not a protocol itself (like HTTP), but rather a guideline for creating web services 4

Radio Frequence Identification (RFID) is technology that uses radio waves to wirelessly identify and track objects 3, 4

Railroad Inventory Data Services (RIDS) 4

Railroad Layout Data Services (RLDS) 4

Rollingstock RFID Manager (RSRM) 3, 4, 10

Single Page Application (SPA) a web application that loads a single HTML page in the user's browser and dynamically updates the content based on user interaction, without reloading the entire page. This creates a more fluid and responsive user experience, similar to what you might expect from a native mobile app 3, 4, 8, 9, 10, 24, 28, 32, 34

virtual machine (VM) is a software program that acts like a physical computer. It creates a virtualized environment that can run its own operating system (OS) and applications, just like a physical computer would 5, 29, 30

Windows Subsystem for Linux 2 (WSL 2) is a feature introduced by Microsoft that allows you to run Linux distributions directly on Windows 10 and 11, alongside your existing Windows environment 21, 22

yet another markup language (YAML) is a human-readable data serialization language used for configuration files, data exchange, and system automation, known for its use of indentation to define structure, similar to Python, and for being a strict superset of JSON 6, 8, 10