RV Store Extended Lab

Docker and Docker Compose

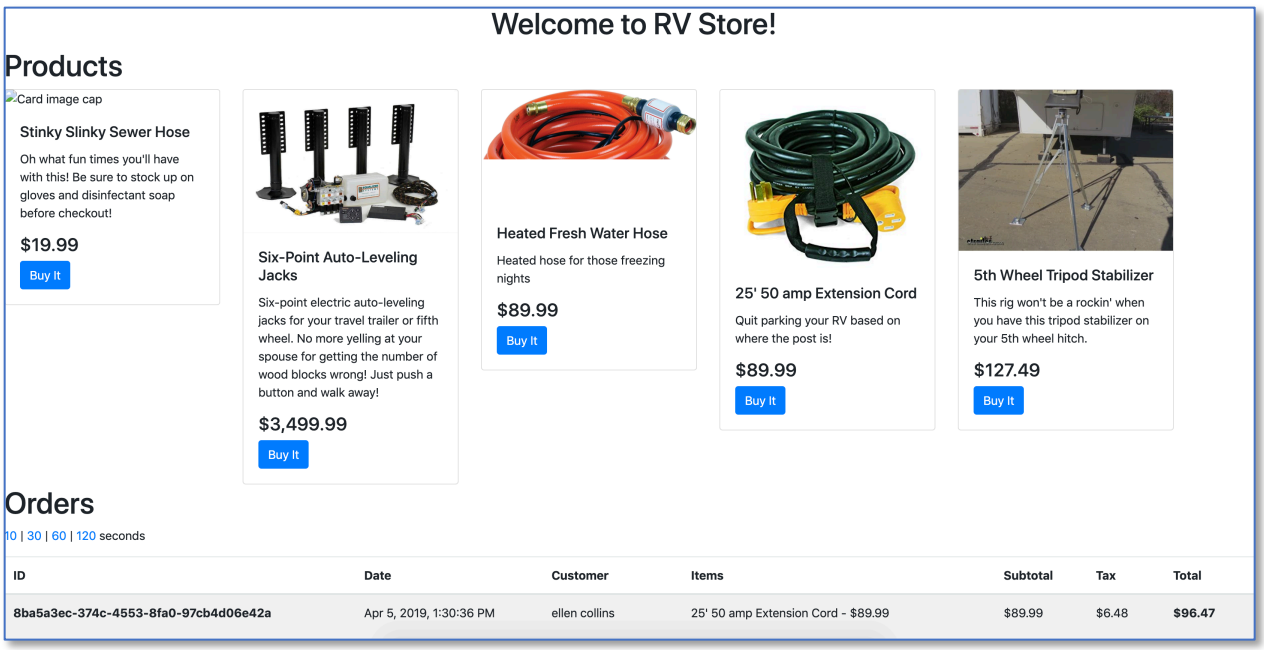
In this lab, you'll get a chance to put your new Docker and Docker Compose skills to the test in a real-world scenario. You'll build Docker images and containers of a real microservice application. Then you'll run them together to get a working application.

For this lab, I will be playing the part of the developer who has come to you to help me containerize my application. I'll provide you the application details necessary (although in true developer style I may miss some details). If you have questions, I'll try to answer them as a developer would. I will not simply give you the answer, but instead try to guide you to discover answers on your own. I want you to work through the process of systematically building up the application as containers. I fully expect you to struggle with some parts, but that's OK! The struggle is where the learning is!

STOP! Do not review the Dockerfiles included with the services in the code repository. Also do not look at the docker-compose.yaml file. These contain all the solutions needed for this lab. Only look at them as a last resort if you get stuck. Try your best to work through the lab without them. Doing so will reduce your knowledge retention!

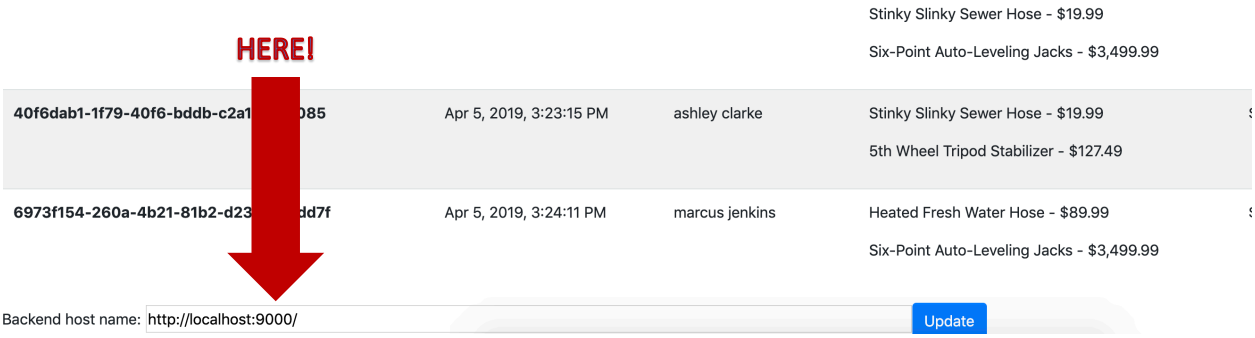
The RV Store

The RV Store is a made-up ecommerce application. Because your friendly instructor lives in an RV full-time and travels the country exploring, this is an obvious choice! It is completely contrived and shallow but shows how several Docker images can be put together to build a real microservice application.



It is made up of the following components:

1. An Angular web-based user interface. The UI calls the backend REST API asynchronously to get data about products and orders. This UI includes a text box that allows you to change the URL of the backend API depending on what port you ran it on. If you don't see products and orders, you may need to update this and click "Update" (ensure it is a fully qualified URL that includes http:// and ends with a trailing forward slash.



2. A product REST API to serve product information. This API responds to requests at the /products endpoint. This service runs on a separate port from the order API. You can test this API directly by visiting http://localhost:9001/products.
3. An order REST API to serve order information and create orders. This API responds to requests at the /orders endpoint. This service runs on a separate port from the product API. This service communicates with the Mongodb database by name. You can test this API directly by visiting http://localhost:9002/orders.
4. A gateway API service (AKA edge API or edge service). This service unifies the other two API services to put them on the same host port. This makes the entire API look like one to the outside world, despite being broken up into many services. It is aware of the other two services and inspects the path and routes the traffic to the appropriate service. It runs on its own port separate from the other two APIs. It communicates with the other two APIs by name. Test this API directly by visiting http://localhost:9000/products or http://localhost:9000/orders.
    a. For example, when a request comes in to http://localhost:9000/products, it will forward the request to http://localhost:9001/products. When a request comes in to http://localhost:9000/orders, it will forward to http://localhost:9002/orders.
5. A Mongo database to store orders. This makes it so that the order service can be scaled in/out and they can all respond with order information.
6. An order simulator service. This does not respond to requests, but instead just creates a new random order once a minute and calls the order service to create one and store it. This enables the UI to display orders that continue to grow. It communicates with the gateway service by name.

Service Details

Here are important details to help tie the components together.

UI Service

This is a static web site made up of images, HTML files, CSS, and Javascript. Pick a web server and serve up the static files located in the Github repo. I suggest Nginx.

- Container port: 80
- Container name: ui
- Static files located in Microservices/ui/dist/ui

API Gateway

This is a Java Spring Boot application that runs on Java 8. It is a runnable jar file, which contains its own application server.

- Container port: 9000
- Container name: rvstore-api-gateway
- Application code located in Microservices/gateway-service/target/gateway-service.jar
- Command to start the process: java -jar gateway-service.jar
- Environment variable needed: SPRING_PROFILES_ACTIVE = compose

Product API Service

This is a Java Spring Boot application that runs on Java 8. It is a runnable jar file, which contains its own application server.

- Container port: 9001
- Container name: rvstore-product-api
- Application code located in Microservices/product-api/target/product-api-0.0.1-SNAPSHOT.jar
- Command to start the process: java -jar product-api-0.0.1-SNAPSHOT.jar
- Environment variable needed: SPRING_PROFILES_ACTIVE = compose

Order API Service

This is a Java Spring Boot application that runs on Java 8. It is a runnable jar file, which contains its own application server.

- Container port: 9002
- Container name: rvstore-order-api
- Application code located in Microservices/order-api/target/order-api-0.0.1-SNAPSHOT.jar
- Command to start the process: java -jar order-api-0.0.1-SNAPSHOT.jar
- Environment variable needed: SPRING_PROFILES_ACTIVE = compose

Mongodb

This is the stock Mongo database. Find it on Docker Hub and follow the instructions to run it.

- Container port: 27017
- Container name: rvstore-orders-mongodb

Order Simulator

This container submits a fresh order to the order service once a minute. It does not need to expose any ports to receive traffic since it's a worker process. It is a Java Spring Boot application that contains the application server in it. It runs on Java 8.

- Container name: rvstore-order-simulator
- Application code located in Microservices/order-simulator/target/order-simulator-0.0.1-SNAPSHOT.jar
- Command to start the process: java -jar order-simulator-0.0.1-SNAPSHOT.jar
- Environment variable needed: SPRING_PROFILES_ACTIVE = compose

Lab Goals

1. Write a Dockerfile for each service and build the image. Get all of the services up and running on your workstation with straight Docker run commands. Get the application fully working. You should do this using your own images that you've built. Do not look at the existing Dockerfiles for the answers!
   Now you have the images built and working. The application works. Now get it running using Docker Compose so that you can quickly bring your application up and down with one command. Do not look at the existing docker-compose.yaml file!
3. Ensure that order information will not be lost if the Mongodb container is stopped and removed (simulate this by stopping it and removing the container).
4. Ensure that containers survive a crash and restart/heal themselves.