The Sock Shop Case Study

Sock Shop (https://microservices-demo.github.io) is an open-source web-based application simulating the user-facing part of an e-commerce website selling socks. It is developed and maintained by *Weaveworks* and *Container Solutions*, and its goal is to allow to test and showcase solutions and tools for orchestrating multi-component applications. *Sock Shop* is indeed a multi-component application, whose components are the followings:

- A *Frontend* displays a web-based graphical user interface for e-shopping socks.
- Different pairs of services and databases allow to store and manage the catalogue of available
- socks (i.e., *Catalogue* and *CatalogueDB*), the users of the application (i.e., *Users* and *UsersDB*), the users' shopping carts (i.e., *Carts* and *CartsDB*), and the users' orders (i.e., *Orders* and *OrdersDB*).
- Two services (i.e., *Payment* and *Shipping*) simulate the payment and shipping of orders.
- A message queue (i.e., *RabbitMQ*) allows to enqueue shipping requests, which are then consumed by a service (i.e., *Queue Master*) simulating the actual shipping of orders.

Modelling *Sock Shop* with TOSCA

We specified *Sock Shop* in TOSCA by exploiting the TOSCA-based representation given by TosKer (https://di-unipi-socc.github.io/tosker-types). We modelled all databases and infrastructure components as nodes of type *tosker.nodes.Container*, whose actual implementations were given by the Docker containers already configured by *WeaveWorks* and *Container Solutions*. We instead specified the services *Frontend*, *Catalogue*, *Users*, *Carts*, *Orders*, *Payment* and *Shipping* as nodes of type *tosker.nodes.Software*, each hosted on a node of type *tosker.nodes.Container* providing the runtime environment it needs (e.g., as *Frontend* requires NodeJS, it is hosted on a container implemented with the Docker image *node:6*). The resulting TOSCA-based representation of the application is shown in Fig. 1.

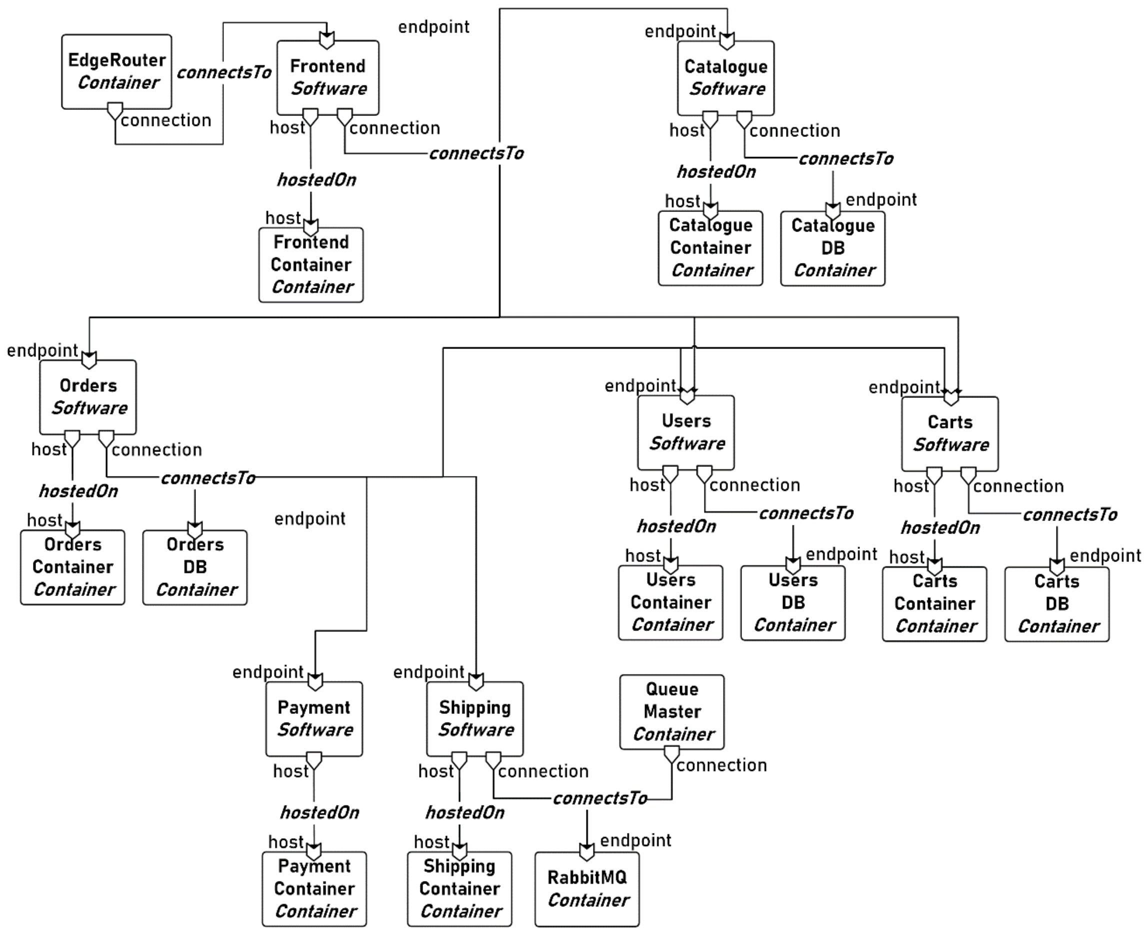


Figure 1. TOSCA-based representation of Sock Shop, obtained by exploiting the TOSCA types defined by TosKer.

We also implemented the operations to *create*, *configure*, *start*, *stop* and *delete* the services mentioned above, each with a different shell script. In other words, for each service, the shell script *install.sh* creates the service in a dedicated folder of its hosting container, by cloning the GitHub repository containing its sources within such folder and by compiling (if needed) such sources. The script *configure.sh* configures the endpoints to be offered by the service. The scripts *start.sh* and *stop.sh* start and kill the process corresponding to the service, respectively. Finally, the script *uninstall.sh* deletes the folder containing the service installation.

The CSAR archive packaging the above described TOSCA application specification is publicly available on GitHub at:
https://github.com/di-unipi-socc/toskose-packager/blob/master/tests/data/sockshop/sockshop.csar.

Generating a Deployable Artifact for Sock Shop

We generated the Docker Compose file enabling a component-aware orchestration of *Sock Shop* on Docker-based container orchestrators by executing the Toskose Packager as follows:

```
$ toskose -p sockshop.csar toskose.yml
```

The archive *sockshop.csar* listed above was a local copy of the CSAR packaging Sock Shop available on GitHub (see above), while *toskose.yml* is the Toskose configuration file shown in Fig. 2. The option *-p* is used to force pushing the automatically generated toskosed images to the Docker Hub.

```
nodes:
   front-end_container:
     alias: front-end
     port: 9001
     user: admin
     password: admin
     log_level: INFO
     docker: { name: giulen/sockshop-front-end_container-toskosed, tag: latest }
   catalogue_container:
     alias: catalogue
     port: 9002
     user: admin
     password: admin
     log_level: INFO
     docker: { name: giulen/sockshop-catalogue_container-toskosed, tag: latest }
   user_container:
     alias: user
     port: 9003
     user: admin
     password: admin
     log_level: INFO
     docker: { name: giulen/sockshop-user_container-toskosed, tag: latest }
   carts_container:
     alias: carts
     port: 9004
     user: admin
     password: admin
     log_level: INFO
     docker: { name: giulen/carts_container-toskosed, tag: latest }
   orders_container:
     alias: orders
     port: 9005
     user: admin
     password: admin
     log_level: INFO
```

```
      docker: { name: giulen/orders_container-toskosed, tag: latest }
    payment_container:
      alias: payment
      port: 9006
      user: admin
      password: admin
      log_level: INFO
      docker: { name: giulen/payment_container-toskosed, tag: latest }
    shipping_container:
      alias: shipping
      port: 9007
      user: admin
      password: admin
      log_level: INFO
      docker: { name: giulen/shipping_container-toskosed, tag: latest }
manager:
  alias: toskose-manager
  port: 10000
  user: admin
  password: admin
  mode: production
  secret_key: secret
  docker: { name: giulen/sockshop-manager, tag: latest }
```

Figure 2. Toskose configuration file used for generating a deployable artifact for Sock Shop.

A snippet of the Docker Compose file automatically generated by the Toskose Packager is shown in Fig. 3. The Toskose Packager introduced and suitably configured each container from the original Sock Shop application, and it also includes an additional container running the Toskose Manager. All such containers are specified as services running on the same overlay network, i.e., *toskose-network*.

```
---
version: '3.7'
services:
  orders-db:
    image: mongo:latest
    init: true
    networks:
      toskose-network:
  front-end_container:
    image: giulen/sockshop-front-end_container-toskosed:latest
    init: true
    networks:
      toskose-network: { aliases: [ front-end ] }
    environment:
    - SUPERVISORD_ALIAS=front-end
    - SUPERVISORD_PORT=9001
    - SUPERVISORD_USER=admin
    - SUPERVISORD_PASSWORD=admin
    - SUPERVISORD_LOG_LEVEL=INFO
    - INPUT_REPO=https://github.com/matteobogo/front-end.git
    - INPUT_CATALOGUE=catalogue
    - INPUT_CARTS=carts
    - INPUT_USER=user
    - INPUT_ORDERS=orders
  catalogue_container:
    image: giulen/sockshop-catalogue_container-toskosed:latest
    init: true
    networks:
      toskose-network: { aliases: [ catalogue ] }
    environment:
    - SUPERVISORD_ALIAS=catalogue
    - SUPERVISORD_PORT=9001
    - SUPERVISORD_USER=admin
    - SUPERVISORD_PASSWORD=admin
    - SUPERVISORD_LOG_LEVEL=INFO
    - INPUT_PORT=80
  ...
  toskose-manager:
    image: giulen/sockshop-manager:latest
    networks:
      toskose-network: { aliases: [ toskose-manager ] }
    init: true
    environment:
    - TOSKOSE_MANAGER_PORT=10000
    - TOSKOSE_APP_MODE=production
    - SECRET_KEY=secret
    ports: [ "10000:10000/tcp" ]
  ...
networks:
  toskose-network: { driver: "overlay", attachable: true
```

Figure 3. A snippet of the Docker Compose file for deploying Sock Shop. The full version of the file is available on GitHub (https://github.com/di-unipi-socc/toskose-packager/blob/master/tests/data/sockshop/docker-compose.yml).

Deploying and Managing Sock Shop on Existing Container Orchestrators

We exploited the Docker Compose file obtained from the Toskose Packager to deploy *Sock Shop* with Docker Compose on both a local cluster of 4 virtual machines created with the Docker Machine tool (https://docs.docker.com/machine), and on a cluster composed by four cloud-hosted virtual machines (i.e., four nodes hosted by Microsoft Azure with the Azure Kubernetes Service), in order to test whether our approach can also effectively work in a real-world cloud scenario. For the sake of brevity, we hereafter only report on the Azure-based deployment and management of *Sock Shop*.

To enact a deployment of *Sock Shop* with the Azure Kubernetes Service, we were first required to setup a cluster on Azure where to run Sock Shop. We hence locally launched a containerised instance of the Azure CLI, which we first used to log into the platform:

```
$ docker run -it -v ${HOME/.ssh:/root/.ssh mcr.microsoft.com/azure-cli
$ az login
```

We then exploited the Azure CLI to create our own resource group, which we used to create a cluster of four virtual machines. The latter was done by directly exploiting Azure Kubernetes Service (i.e., aks):

```
$ az group create --name MyResourceGroup --location eastus
$ az aks create --resource-group myResourceGroup --name AKSCluster \
                --dns-name-prefix AKSCluster-dns --node-count 4 \
                --node-vm-size Standard_DS1_v2 --enable-addons monitoring \
                --generate-ssh-keys --kubernetes-version 1.13.12
```

Once created, we were able to connect to the Kubernetes-managed cluster with the command:

```
$ az aks get-credentials --resource-group myResourceGroup --name AKSCluster
```

We then deployed the containers in *Sock Shop* by executing

```
$ kubectl create -f sockshop
```

where *sockshop* was the directory containing the Kubernetes specification automatically obtained from Kompose (https://kompose.io), to which we fed the Docker Compose file automatically generated by the Toskose Packager.

The above allowed us to deploy the containers of Sock Shop and have them up and running. The software components hosted on such containers were instead not deployed yet, as their actual management was to be orchestrated through Toskose Manager. We hence completed the deployment of Sock Shop by (i) starting Shipping and Carts with

```
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/shipping/shipping-sw/start"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/carts/carts-sw/start"
```

(ii) installing and starting Users, Payment and Catalogue with

```
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/user/user-sw/create"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/user/user-sw/start"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/payment/payment-sw/create"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/payment/payment-sw/start"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/create"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/start"
```

(iii) installing and starting Orders with

```
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/orders/orders-sw/create"
      $ curl -X POST -H "accept: application/json" \
       "http://52.142.38.162:10000/api/v1/node/orders/orders-sw/start"
```

and (iv) installing, configuring and starting Frontend with

```
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/front-end/front-end-sw/create"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/front-end/front-end-
      sw/configure"
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/front-end/front-end-sw/start"
```

As a result, we obtained a running instance of allowing to browse among the catalogue of socks e-sold through its web-based portal (Fig. 4).

The above again showed us that we were able to piggyback on an existing Docker-based container orchestrator for deploying the containers forming an application, and to independently orchestrate the deployment of the components running on such containers by exploiting the Toskose toolset. To further experiment the independent management of components and containers, we stopped the service managing the catalogue of available socks by issuing the following cURL command:

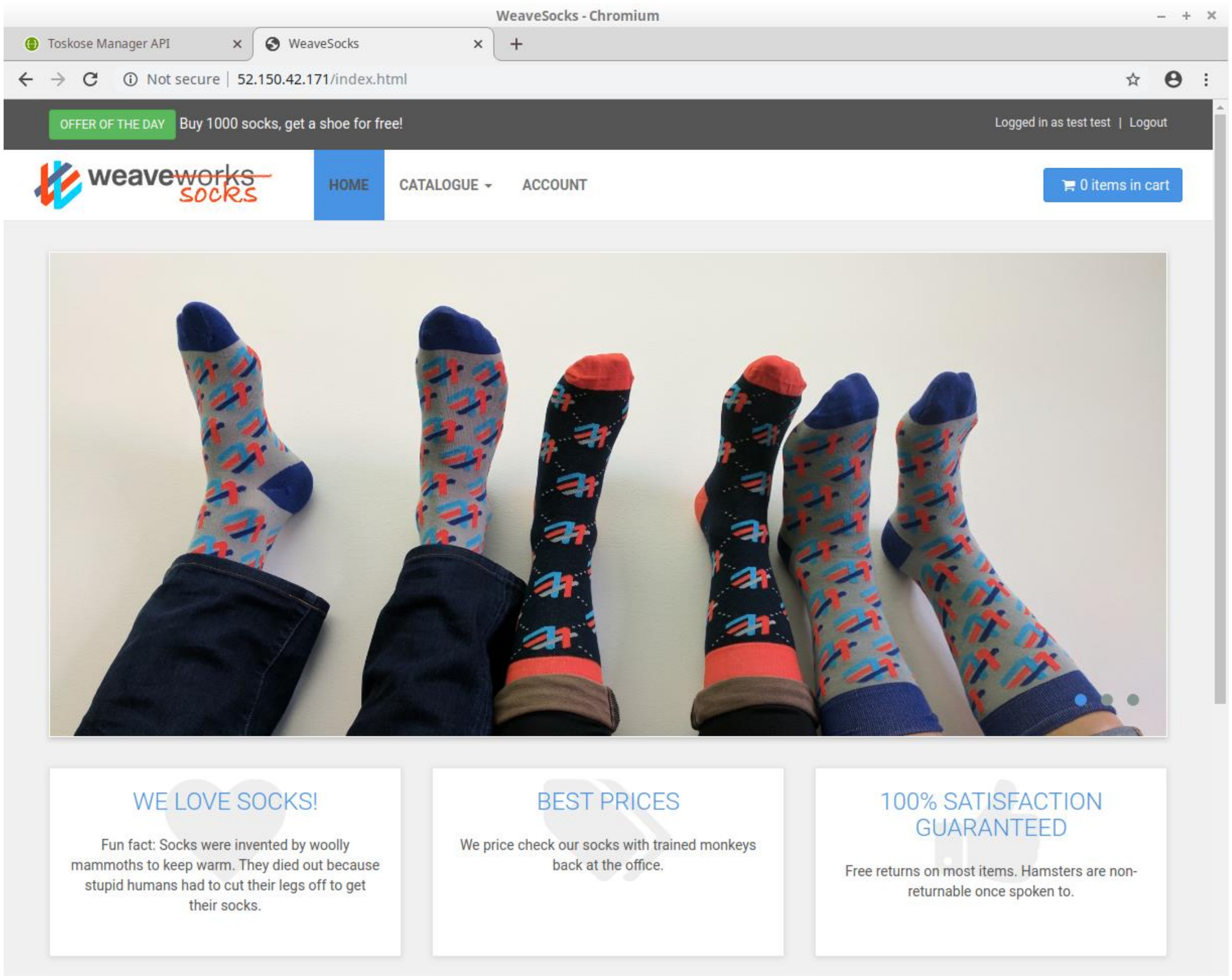
```
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/stop"
```

The above resulted in the web-portal no more being able to show the socks that can be e-shopped, because of the web-based Frontend not being able to connect to Catalogue (Fig. 5). This happened even if the container hosting Catalogue was still working (Fig. 6), hence showing that the lifecycle of Catalogue and of its hosting container were managed independently.

Finally, to return having the instance of Sock Shop properly working, we issued the cURL command

```
      $ curl -X POST -H "accept: application/json" \
            "http://52.142.38.162:10000/api/v1/node/catalogue/catalogue-sw/stop"
```

which (re-)started the Catalogue and brought the application back to offer a fully working web-based portal for e-shopping socks (such as that in Fig. 4).



WE LOVE SOCKS!

Fun fact: Socks were invented by woolly mammoths to keep warm. They died out because stupid humans had to cut their legs off to get their socks.

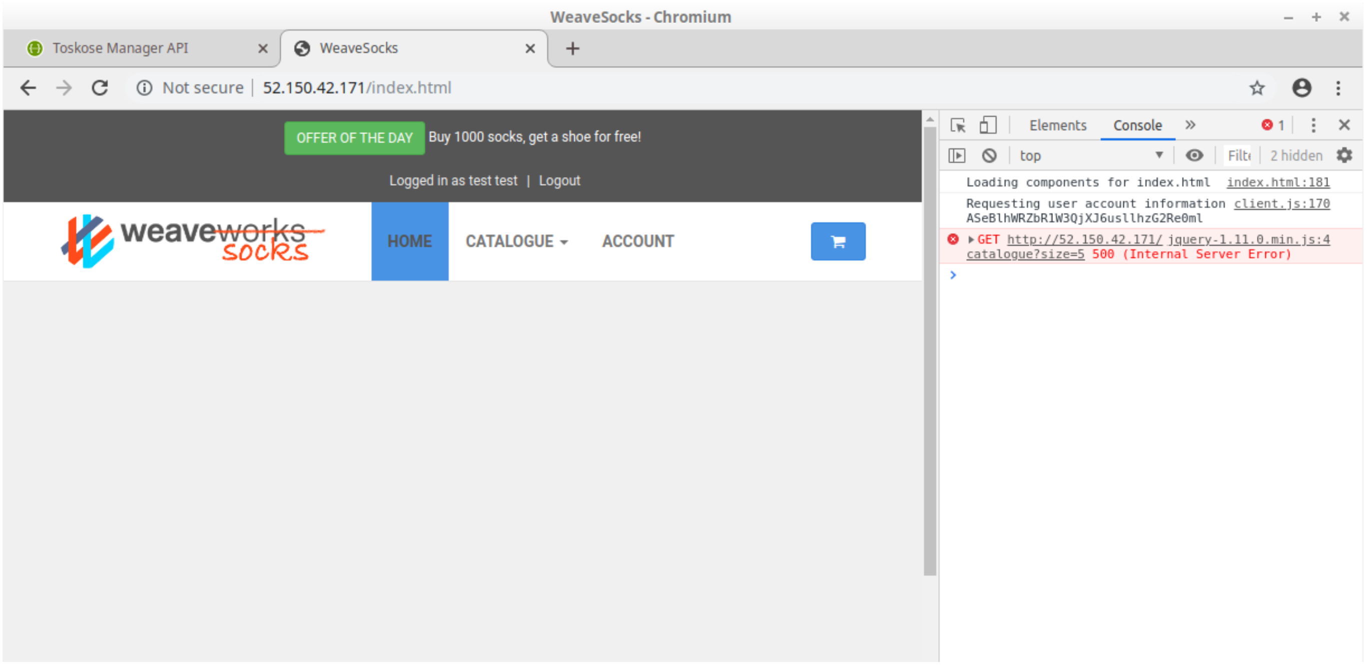
BEST PRICES

We price check our socks with trained monkeys back at the office.

100% SATISFACTION GUARANTEED

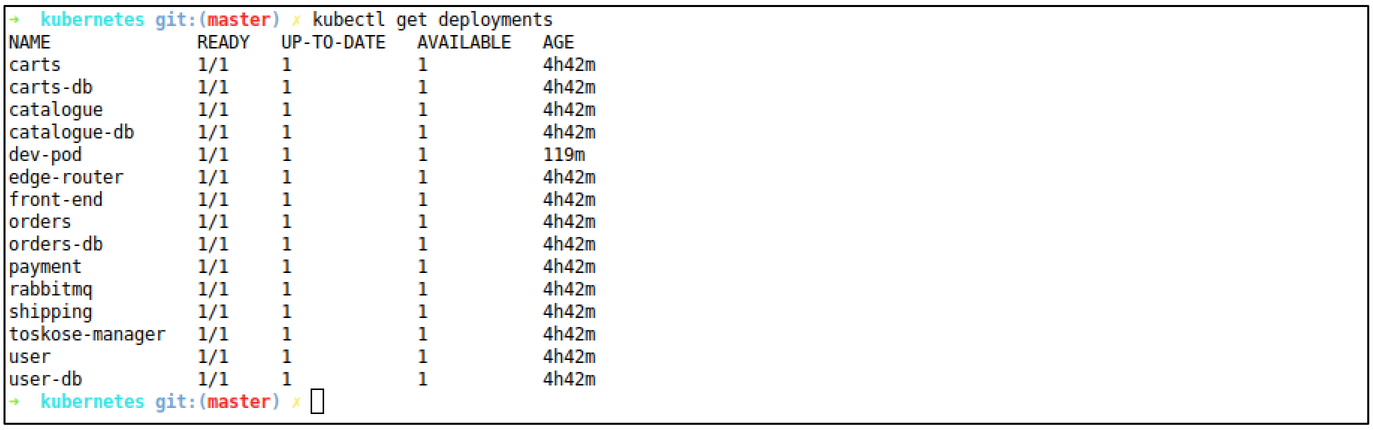
Free returns on most items. Hamsters are non-returnable once spoken to.

Figure 4. Snapshot of the running instance of Sock Shop obtained after completing its deployment.



```
Loading components for index.html  index.html:181
Requesting user account information client.js:170
ASeBlhWRZbR1W3QjXJ6usllhzG2Re0ml
GET http://52.150.42.171/ jquery-1.11.0.min.js:4
catalogue?size=5 500 (Internal Server Error)
```

Figure 5. Snapshot of the instance of Sock Shop after stopping its Catalogue.



```
➜  kubernetes git:(master) ✗ kubectl get deployments
NAME             READY   UP-TO-DATE   AVAILABLE   AGE
carts            1/1     1            1           4h42m
carts-db         1/1     1            1           4h42m
catalogue        1/1     1            1           4h42m
catalogue-db     1/1     1            1           4h42m
dev-pod          1/1     1            1           119m
edge-router      1/1     1            1           4h42m
front-end        1/1     1            1           4h42m
orders           1/1     1            1           4h42m
orders-db        1/1     1            1           4h42m
payment          1/1     1            1           4h42m
rabbitmq         1/1     1            1           4h42m
shipping         1/1     1            1           4h42m
toskose-manager  1/1     1            1           4h42m
user             1/1     1            1           4h42m
user-db          1/1     1            1           4h42m
➜  kubernetes git:(master) ✗ 
```

Figure 6. Snapshot of the containers actually running while the instance of Sock Shop was not properly working.