Tutorial

This tutorial will help you get started with Docker, Docker Compose and Redis. During the tutorial a small sample project is used to get you started. The sample project will use a Symfony PHP application which can be accessed through a Nginx webserver. The application will be accessible via http://localhost:80. This sample project will load 1000 car entries into a MySQL database. This sample project also contains a Redis instance for caching data to improve the response times.

When you visit the URL mentioned above 1000 database entries are retrieved from the database and cached in Redis. The data is cached for ten seconds. When you visit the URL within 10 seconds the data is not retrieved from the database but is returned immediately by Redis.

Setting up Docker

Before we start you need to install Docker on your device. You can follow the instructions described on the Docker website:

https://docs.docker.com/install/

Checkout the example code

git clone https://github.com/martijnvankempen/tutorial.git

Your directory now should look like this:

```
├── app/
|   └── tutorial-application/
|       ├── bin/
|       ├── config/
|       ├── public/
|       ├── src/
|       ├── var/
|       ├── vendor/
|       ├── .env
|       └── .composer.json
├── docker/
|   ├── nginx/
|   |   └── default.conf
|   └── php/
|       └── Dockerfile
├── docker-compose.yml
└── README.md
```

Docker

Dockerfile

Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image

The Dockerfile you find in the docker/php directory describes our own implementation of a Docker PHP image. With Docker you can create a Docker image that is created based on your Dockerfile. With Docker or Docker Compose you can run your Docker image which will result in a running Docker container.

The first line of the Dockerfile starts with FROM. The FROM instruction initializes a new build stage and sets the Base Image for subsequent instruction. Available images can be found on the Docker Hub: https://hub.docker.com/.

Other lines have RUN instruction. The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

In this Dockerfile we also set the WORKDIR. The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.

More Dockerfile options can be found here:

https://docs.docker.com/engine/reference/builder/

During this tutorial we will use docker-compose to create our Docker images and start our Docker containers.

Docker Compose

Docker Compose is a tool for defining and running multi-container Docker applications. With Docker Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. All Docker Compose options can be found here:

https://docs.docker.com/compose/compose-file/

When you open the docker-compose.yml file you will see that we have defined four services. As said earlier this tutorial will use a MySQL database, PHP (Symfony), Nginx (Webserver) and Redis for caching the data. In this Docker Compose file we list which Docker images we want to use and what names the Docker containers should have. You can also open specific ports, set environment variables and map volumes to your Docker containers.

The directories on your device will be mounted to the Docker container. Windows will ask you to share your drive when you are using Docker Compose on Windows.

We will use standard Docker images for MySQL, Nginx and Redis. We created our own PHP docker image because we need some extra options in our PHP docker container.

Building using Docker Compose

Open a terminal on your device and type the command below in the directory where the docker-compose.yml file is saved.

docker-compose build --force-rm

This will build our own PHP Dockerfile and create a Docker image. You can list the Docker images via the command below. This will return two Docker images. The default PHP image and our own PHP image. Our own image uses layers from the standard PHP image.

docker images

Running using Docker Compose

When executing the command below Docker Compose will start all four Docker containers. It does not have the MySQL, Nginx and Redis image yet because we didn't build it yet. Docker Compose will automatically download the images from the Docker Hub.

docker-compose up -d --force-recreate

Docker Compose will use the four Docker images to start four Docker Containers. The downloaded Docker images can be found using the command that was used earlier. We can verify if our Docker containers are running by executing the command below. When you add the -a option exited containers are also added to the list.

docker ps -a

Exited container

It is possible that your container didn't start for some reason. You can view the logs of an exited container via the following command:

docker logs <name_of_container_or_container_id>

Entering a container

You can only enter a container when it is running. If a Docker container exists all data that isn't mounted via a volume is lost. You will only be able to view the logs with the command mentioned above.

You can enter a running container via the following command:

docker exec -it <name_of_container_or_container_id> bash

Setting up the database

Because this tutorial needs sample data and a sample database table we will enter the PHP docker container and migrate the database so it will have 1000 records in the database.

Enter the PHP docker container via the following command

docker exec -it tutorial-php bash

You will enter the container at /usr/src/app because this was the WORKDIR defined in our Dockerfile. Move to the tutorial directory via the following below. This is the root directory of our sample project.

cd tutorial-application

Now we will create our database table and load the sample data into the database. You can do so by running the following command:

php bin/console d:m:migrate -n

Database connection

The configured host, username and password can be found here:

```
├── app/
|   └── tutorial-application/
|       └── .env
```

Database migration

The database migrations including the init script can be found here:

```
├── app/
|   └── tutorial-application/
|       └── src
|           └── Migrations/
```

Getting the data

Because the webserver was automatically setup by Docker Compose we can open our browser and go to:

http://localhost:80

The webserver will accept the request on port 80 and pass it to our running PHP container. When you open the URL you will see a JSON output of all 1000 car database entries and the time of the request (in UTC+0 timezone).

We also have a running Redis container that is used by our PHP container. Please have a look at our DefaultController that handles the request and returned the data. This file can be found at:

```
├── app/
|   └── tutorial-application/
|       └── src/
|           └── Controller/
|               └── DefaultController.php
```

At the start of the indexAction method a Redis client is created. The host is Redis which is mapped by Docker to the ip address of the Redis container. The code will look for a Redis cache entry. When it cannot find the cache entry in Redis it will retrieve all car database records from the database. All the database records are added to an array and converted into JSON so that it can be returned via

our endpoint. The JSON is converted into a base64 string and is added to the Redis cache. The expire time for the entry is set to 10 seconds.

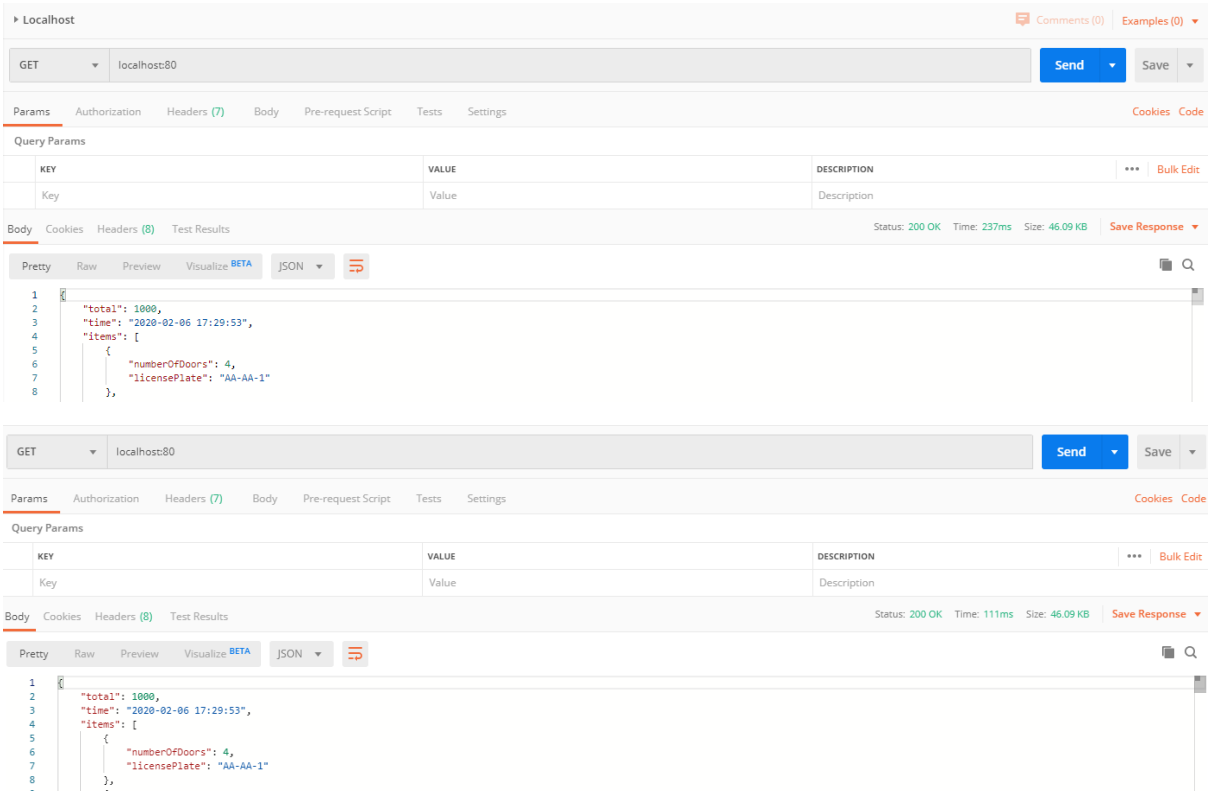
If you retrieve the same data within 10 seconds the code will find a cache entry and will return the data instead of retrieving it via the database. You can see that the data is cached because the time in the response is cached for 10 seconds as well.

Performance

You can measure the performance impact of the Redis caching when using an application like Postman. You can download Postman via:

https://www.postman.com/downloads/

When you execute a GET request to localhost:80 via Postman you can see the response time. The first GET request will take += 250ms. The second time that you access the GET call (within 10 seconds) you can see that the response time decreased to +=110 ms. This difference will only get bigger when your database has more than 1000 records.



Updating the TTL of the cache entry

Open the DefaultController and go to line 57. The second parameter of the method expire is the ttl of the cache entry in seconds.

```
├── app/
|   └── tutorial-application/
|       └── src/
|           └── Controller/
|               └── DefaultController.php
```

Using Redis CLI

Display the cached keys in Redis container

You can view the cached keys inside the Redis container.

1) Enter the Redis container via: docker exec -it tutorial-redis bash

2) Type: redis-cli and hit enter

List available keys

You can list all keys in Redis that match a certain pattern. You can list the keys used by the sample project by executing:

KEYS cache-entry. This will list all keys matching the cache-entry pattern.

Get value of key

You can list the value of our key by executing: get cache-entry

This will output a base64 string of our JSON response.

Troubleshoot

Fatal error

When opening localhost:80 in your browser or when migrating the database data you get one of the errors displayed below. This could be happing for two reasons:

1) Your dependencies were not downloaded correctly

1.1) Enter the docker-php container by executing the following command: docker exec -it tutorial-php bash

1.2) After entering the command go to the tutorial-application directory using the following command: cd tutorial-application/

1.3) Download the dependencies using: composer update

1.4) Now the dependencies are downloaded and the error should be fixed

2) Your volume was not mounted correctly

2.1) When you use Docker Desktop for Windows you should make sure that your drives are shared. The following link will explain how to do so: https://token2shell.com/howto/docker/sharing-windows-folders-with-containers/

Fatal error: require(): Failed opening required '/usr/src/app/tutorial-application/vendor/autoload.php' (include_path='.:/usr/local/lib/php') in /usr/src/app/tutorial-application/config/bootstrap.php on line 5

Warning: require(/usr/src/app/tutorial-application/vendor/autoload.php): failed to open stream: No such file or directory in /usr/src/app/tutorial-application/config/bootstrap.php on line 5

Redis CLI get cache-entry returns (nil)

After you entered your Redis container and executed redis-cli you execute get cache-entry but it returned (nil). This happens because the TTL of the cache entry has passed. You can go to localhost:80 so a new cache entry is created or you update the TTL in the DefaultController. The get command will return (nil) when no data is cached.