Modelit
Elisabethdreef 5
4101 KN Culemborg

info@modelit.nl
www.modelit.nl

Deploying Matlab applications with Docker

Date: March 3, 2022

Content

1 Introduction 1
1.1 About this document 1
1.2 In short 1
1.3 Background 1
1.4 Docker containers 2
1.5 Contents of this guide 2
2 Method 1: Use the documented Matlab method 3
2.1 Basic procedure 3
2.2 Limitations 3
3 Method 2: use Docker command line 4
3.1 Install Docker 4
3.2 Graphical overview 4
3.3 Step by step approach 5
4 Example: remote evaluation 8
4.1 Introduction 8
4.2 Test back-end program in the Matlab IDE (developer PC) 8
4.2.1 Standalone test in Matlab IDE 9
4.2.2 Web service test in Matlab IDE 9
4.3 Verify application in a Linux environment, in the Matlab IDE ... 10
4.4 Compile the application (Linux) 11
4.5 Verify the application (Linux) 11
4.6 Package the application in a Docker image 12
4.7 Run the Docker image 13
5 Multi container Docker images 15
5.1 Introduction 15
5.2 Example application 16
5.2.1 Organization of the application 16
5.2.2 Front-end 17
5.3 Steps for creating a multi-container application 18
5.3.1 yml script and build command 20
6 Appendix: createDockerfile, example use 21
6.1 Usage of createDockerfile 21
6.2 Source code 21
6.3 Example input 21
6.4 Example output 21
7 Useful links 23

1 Introduction

1.1 About this document

This document is aimed at Matlab developers who need to deploy algorithms written in Matlab code in a typical cloud or computer center setting, with or without a webservice interface.

1.2 In short

To enable robust deployment of Matlab programs as microservices in the cloud or computer centers Modelit has created a toolbox that integrates a webserver into Matlab applications and has documented a procedure to package these programs in so called Docker images. This document describes this procedure. A separate document describes the webserver toolbox.

1.3 Background

Between initial idea and final deployment of any data driven method lies a path of analysis, experimenting, implementation and testing. At each stage of development, different requirements apply to development tools and programming environment(s) that are used. The table below outlines the typical requirements that apply during the research phase versus the deployment phase.

Research phase	**Deployment phase**
• Rely on specialized researchers	• Should not require specialized knowledge
• Develop the optimal algorithm	• Guarantee security and availability
• Improvise and explore options	• Use standardized deployment procedure
• Work with an interactive environment, adhoc scripts, and visual/graphic feedback	• Must run unattended, preferably as a webservice • Processes must integrate in an environment that is scalable and robust
• Run on a personal computer	• Deploy in the cloud, computer center or Linux platform
• One license per developer is acceptable	• Royalty free distribution is preferred

These differences often result in the decision to re-program all functionality after the proof-of-concept phase of a project is complete. This leads to duplication of work, additional costs and time required for implementation, and the risk of introducing errors. And maybe more important this makes it very hard to realize improvements by means of iterations.

Out of the box, Matlab is a great tool to develop and test algorithms. However, this only takes care of the requirements in the left column. In order to take care of the right column, further requirements apply:

- The application must be able to communicate with the outside world through http;
- The application must be packaged in such a way that it can run in the cloud or computer center.

1.4 Docker containers

"A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings" (quoted from [1])

A Docker container image of a Matlab application typically contains the compiled Matlab code, an installation of the Matlab Component Runtime (MCR) library, and any data files needed to run the application. A container image can be run on a Docker engine, which in turn runs on the host operating system.

Unlike with a Virtual Machine, the Docker engine does not emulate the entire operating system. "Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space" (quoted from [1]). Because of this, a program running with Docker requires much less resources than the same program running on a Virtual Machine.

The popularity of Docker has sparked many useful third-party developments, like ready-for-use images that implement building blocks like a database or a load balancer.

Productivity and short development cycle times are often a reason for selecting Matlab as a development tool. In many cases Matlab developers value an iterative development process. Docker complements this by greatly simplifying the DevOps cycle, allowing for inclusion of deployment in consecutive iterations. This is a great advantage if time to market is key.

1.5 Contents of this guide

This guide describes how to deploy a Matlab application as a Docker image. The guide describes two methods.

Method 1 uses the Matlab-documented procedure
"Package MATLAB Standalone Applications into Docker Images" as a point of departure. This procedure is available for Matlab versions 2020b and later and is restricted to Linux.

Method 2 entirely relies on the tools provided by Docker and can be used with any Matlab version. Also, it should be applicable to Windows too, and it allows more control over the final Docker image.

2 Method 1: Use the documented Matlab method

2.1 Basic procedure

As a point of departure, we use the procedure "Package MATLAB Standalone Applications into Docker Images" described in the Matlab documentation. To access this documentation from within the Matlab IDE type:

```
>> doc docker
```

And then look for the hyperlink as shown below.

See Also

compiler.build.Results | compiler.build.standaloneApplication | compiler.package.DockerOptions

Topics

Package MATLAB Standalone Applications into Docker Images

Introduced in R2020b

The information can also be found online. Its last known location is:
https://nl.mathworks.com/help/compiler/package-matlab-standalone-applications-into-docker-images.html

The information contains all that is required to build and deploy your Docker image based on a compiled Matlab application.

2.2 Limitations

The basic procedure is a good starting point, but has a few limitations:

- the procedure is not available for Matlab versions prior to version 2020b;
- the procedure only works with Linux;
- the procedure does not support multi-container applications;
- the procedure uses a dedicated syntax that allows the user to control a subset of the properties that can be controlled with the equivalent "docker build" command (see method 2), and does not benefit from the public documentation that is available for "docker build" (see https://docs.docker.com/engine/reference/commandline/build).

Chapter 3 describes a method that will probably result in a more shallow learning curve initially, but is sometimes required to address all requirements of the project.

3 Method 2: use Docker command line

This chapter describes how to create and deploy a Docker image that encapsulates a compiled Matlab application without using the Docker-specific commands packaged with Matlab version 2020b and beyond.

Follow these steps for packaging compiled Matlab applications preceding version 2020b, to deploy on Windows, or when additional control over the final product is required.

The steps below assume the Linux/Ubuntu platform. But deployment with Windows is possible in an analogous way.

3.1 Install Docker

The Docker engine is the program that runs Docker images. For install instructions, see:

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

or (for the Windows case)

https://hub.docker.com/editions/community/docker-ce-desktop-windows

3.2 Graphical overview

The image below outlines the deployment process. In this image rectangles represent product stages and rounded rectangles represent actions.

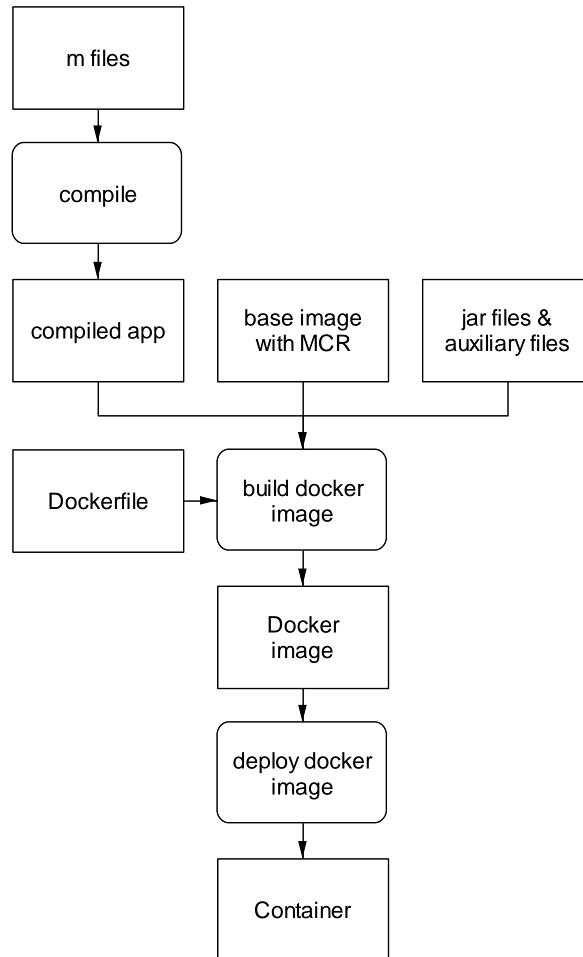


Figure 1: *Outline of deployment of a Matlab application in a Docker container*

3.3 Step by step approach

The objective is to create a Docker image and deploy it on a private server or with a cloud hosting service. Once Docker and Matlab are properly installed on the Linux computer, this should be a simple task that can be completed in a few minutes, provided one exactly knows which steps are required.

For a person without Docker experience however, the challenge is to execute the process without introducing errors or skipping a step. Despite all the benefits attributed to the Docker development tools, for novices, this process can be hard to setup and debug.

This manual aims to present a script with a number of verifiable progressions that will help the typical Matlab user execute the process. After becoming familiar with the process, the intermediate verifications can be skipped to save time.

The table below outlines the steps. Analogous to these steps chapter 4 presents a practical example. This chapter will also discuss the various steps in more detail.

Verification step	Notes
Verify the application in the primary development environment	As a point of departure, we assume m-files of the application run well in the development environment, usually this is the Matlab integrated development environment (IDE) on a personal computer.
Move sources and data to Linux environment.	Move m-files and required auxiliary files to the computer that is used to create the Docker image. Extracting all required files from the code base in the development machine can be a struggle. The Matlab utilities “depfun” (2013b and before) and “requiredFilesAndProducts” (2014a and beyond) automatize this task but in general will select a (much) wider set of m-files than required. An alternative is usage of Matlab drive. If the primary development environment relies on a customized static java class path, the path must be setup on the Linux machine through a dedicated javaclasspath.txt file in Linux startup folder (see Matlab documentation).
Verify the application in the Linux environment, in the Matlab IDE (optional)	Because (auxiliary) files have been moved verification of the application is recommended. If the OS of the primary development is non-Linux, any mex files that are used must be recompiled for the Linux platform. Other migration issues to check for might be calls to the OS like “dos” and the usage of hard-coded file separators “\”.
Compile the application	Because the target OS is Linux, the application must be compiled on a Linux machine. The typical compile command is: `mcc('-m', '-d', targetdir, 'target.m');` where “target.m” is the entrypoint of the application and “targetdir” is a subfolder of the current directory.
Verify the compiled application on Linux (optional)	Do not forget to setup a proper javaclasspath.txt file.
Package the application in a Docker image	Move the compiled application and all auxiliary files to a separate folder, or re-use the earlier folder “targetdir”. Important! Note that within the docker image this folder is referred to as ‘/’

	(root). Any file references must take this into account. Create a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble a Docker image. Again, update the javaclasspath.txt file to reflect the location of the custom jar files (relative to the root of the image!). Build the docker image. A typical build command is: sudo docker build -t <myProject> . (do not forget the dot) with <myProject> a projectname of choice
Verify the Docker image	If the built is successful a container can be started with the command: sudo docker run -it <myProject> or: sudo docker run -it -p ePort:iPort <myProject> (tcp) port ePort will be forwarded to iPort inside the container.
Deploy the Docker image	The container image contains all data, executables and libraries required to run the application. It can be deployed on any computer with Docker installed. As long as the application does not use any kernel specific features, this even applies for computers with a different OS.

4 Example: remote evaluation

4.1 Introduction

As an example, we use the remote evaluation tool, as included in the embedded webserver toolbox. The example creates a webservice that executes function calls from a Matlab client. The client-side syntax for these calls is near-identical to the "feval" syntax.

There might be different reasons for doing this, for example:

- offload computational tasks to a more powerful CPU;
- make requests to a server that takes care of real time data acquisition;
- perform calculations based on privileged data;
- establish a centralized storage for a multi platform application

The embedded webserver toolbox allows asynchronous function calls. And if required, multiple instances of the remote evaluation utility can run simultenously. Combined with asynchronous function calls this allows parallel execution of CPU intensive tasks.

One could also install the server on a central computer and share it with a group of users. In this setup each user is provided with a high peak performance at moderate additional costs.

4.2 Test back-end program in the Matlab IDE (developer PC)

Before we compile and package our server-side function, it is a good idea to test in the Matlab IDE. The function that will be ran on the server is "simpleMailbox" as seen below. This function allows posting and retrieving messages.

```
function messages = simpleMailbox(username,message)

% CALL:
%      simpleMailbox(username) retrieve messages for use "username"
%      simpleMailbox(username,message) send message to user "username"
%
% INPUT:
%     username:
%        identity of sender or reciever
%     message:
%         data to be sent
%
% OUTPUT:
%     messages:
%         cell array with all undelivered messages for "username"
%

persistent allMessages
assert(ischar(username),'First argument must be string identifying user')
messages={};
if nargin==2
    %send a message
    this.username = username;  %must be char string
    this.message  = {message}; %can be any type, use cell to avoid conflicts
    allMessages=cat(1,allMessages,this);
else
    %fetch all messages for username
    if isempty(allMessages)
        return  %no messages at all
    end
    tf=ismember({allMessages.username},username);
```

```
    messages=[allMessages(tf).message];  %deliver mail for username
    allMessages=allMessages(~tf);        %clear delivered messages
end
```

4.2.1 Standalone test in Matlab IDE

First, we check the function in standalone mode with the commands below:

```
simpleMailbox('userA','A1');
simpleMailbox('userA','A2');
simpleMailbox('userB','B1');

rC=simpleMailbox('userC')
rA=simpleMailbox('userA')
rB=simpleMailbox('userB')
```

The output should be:

```
rC =  []
rA =  1×2 cell array {'A1'}    {'A2'}
rB =  1×1 cell array {'B1'}
```

4.2.2 Web service test in Matlab IDE

Next, we setup "simpleMailbox" for remote execution, but still in the Matlab IDE. For this we need two Matlab sessions. In the first Matlab session we execute:

```
function startSimpleMailbox
server = modelit.web.server.Server('0.0.0.0',4444);
server.addEvaluation('allowedFunctions','simpleMailbox',...
    'username','usr1',...
    'password','pw1')
start(server);
```

- The first command instantiates a "blank" webservice, that listens to port 4444;
- The method "addEvaluation" installs a specific callback function that will process incoming requests. The name/value pairs used in this call are optional and explained below

Name/Value pair (optional)	Impact
'allowedFunctions', 'simpleMailbox'	Only the function "simpleMailbox" is accepted for execution. "allowedFunctions" can be a single string, or an array of strings packaged in a cell or string array. If "allowedFunctions" is not specified, any function is accepted for execution.
'username','usr1' 'password','pw1'	Restrict access. Refuse requests without these credentials. If the argument "username" is not specified, credentials will not be required.

- The command "start(server)" activates the server.

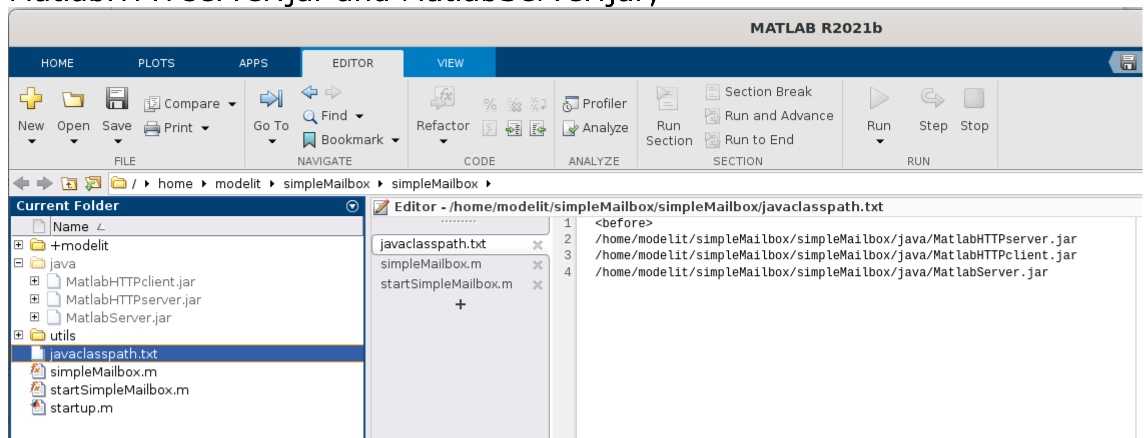
Once the server is running, open a second Matlab session to test the server.

```
h = modelit.web.client.HttpRequest('GET','http://localhost:4444')
h = setCredentials(h,'usr1','pw1')

feval(h,@simpleMailbox,'userA','A1');
feval(h,@simpleMailbox,'userA','A2');
feval(h,@simpleMailbox,'userB','B1');

rC = feval(h,@simpleMailbox,'userC')
rA = feval(h,@simpleMailbox,'userA')
rB = feval(h,@simpleMailbox,'userB')
```

Verify that the output is consistent with the output observed in section 4.2.1.

4.3 Verify application in a Linux environment, in the Matlab IDE

- collect all sources in a .zip file;
- create a folder "simpleMailbox" on the deployment machine (Ubuntu);
- move the zip file to this folder and select "extract here", a subfolder is created;
- move to this subfolder (assuming this is where the "startup.m" is located);
- open a terminal, start a Matlab session with the command "matlab";
- create or update the file "javaclasspath.txt" to include references to MatlabHTTPserver.jar and MatlabServer.jar;



- restart Matlab, verify the java classpath, by typing "javaclasspath" in the Matlab console;
- start the server the command "startSimpleMailbox";
- start a second Matlab session locally, and run the script "testSimpleMailbox". This script basically contains the commands shown in section 4.2.2
- An optional additional test is to invoke the webservice from another computer in the Local Area Network. This requires that port "4444" is accessible for other computers on the network. The Linux command for this is:

  sudo ufw allow 4444/tcp

4.4 Compile the application (Linux)

Start the Matlab IDE and move to the previous folder “simpleMailbox”

```
target='startSimpleMailbox';
targetdir= fullfile(pwd,['sh_',target]);
mcc('-m', '-d', targetdir, target);
```

This creates the shell script “run_startSimpleMailbox.sh” and the runnable “startSimpleMailbox” in subfolder “sh_startSimpleMailbox”.

4.5 Verify the application (Linux)

To run the example, two jar files belonging to the embedded webserver toolbox must be copied to the folder “sh_startSimpleMailbox” and a javaclasspath.txt file must be created that refers to these.

```
<before>
MatlabHTTPserver.jar
MatlabServer.jar
```

Note that the path to the .jar files has been omitted here. This simplification is permitted if one places the .jar files in the startup folder of the project.

Then open a terminal in the folder “sh_startSimpleMailbox” and issue the command:

```
./run_startSimpleMailbox.sh /usr/local/MATLAB/MATLAB_Runtime/v911
```

Note that the location of the Matlab runtime depends on the Matlab version. v911 corresponds to R2021b.

Now complete the verification by running “testSimpleMailbox” in a separate Matlab session. Your terminal should look like:

```
modelit@xeon1220:~/simpleMailbox/sh_startSimpleMailbox$ ./run_startSimpleMailbox
.sh /usr/local/MATLAB/MATLAB_Runtime/v911
------------------------------------------
Setting up environment variables
---
LD_LIBRARY_PATH is .:/usr/local/MATLAB/MATLAB_Runtime/v911/runtime/glnxa64:/usr/
local/MATLAB/MATLAB_Runtime/v911/bin/glnxa64:/usr/local/MATLAB/MATLAB_Runtime/v9
11/sys/os/glnxa64:/usr/local/MATLAB/MATLAB_Runtime/v911/sys/opengl/lib/glnxa64
Start in folder /home/modelit/simpleMailbox/sh_startSimpleMailbox
Use hostname: 0.0.0.0
Use port 4444
24-Feb-2022 13:35:01: Server is ready and listening to port 4444.
message recieved
message recieved
message recieved
message recieved
message recieved
message recieved
24-Feb-2022 13:36:01: Server is ready and listening to port 4444.
24-Feb-2022 13:37:01: Server is ready and listening to port 4444.
24-Feb-2022 13:38:01: Server is ready and listening to port 4444.
24-Feb-2022 13:39:01: Server is ready and listening to port 4444.
24-Feb-2022 13:40:01: Server is ready and listening to port 4444.
```

4.6 Package the application in a Docker image

The key to creating a Docker image is a so called Dockerfile
For the current project this file looks like:

```
FROM matlabruntime/r2021b/release/update0/c000000000000000
############################################
LABEL Description="MATLAB R2021b startSimpleMailbox"
LABEL Vendor="Modelit"
LABEL Web="http://www.modelit.nl"
LABEL Version="R2021b"
############################################
COPY run_startSimpleMailbox.sh .
COPY startSimpleMailbox .
COPY javaclasspath.txt .
COPY MatlabServer.jar .
COPY MatlabHTTPserver.jar .
############################################
RUN chmod a+x ./startSimpleMailbox
RUN chmod a+x ./run_startSimpleMailbox.sh
############################################
CMD ["sh", "-c", "./run_startSimpleMailbox.sh /opt/matlabruntime/v911"]
```

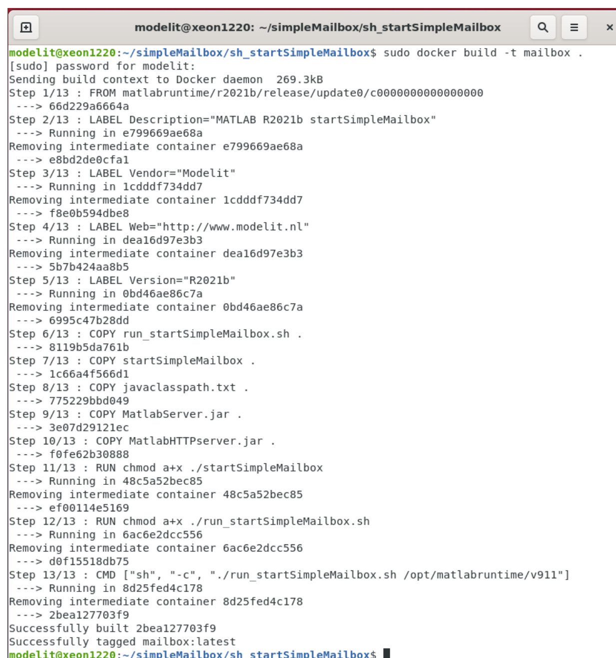
Each line in Dockerfile is labeled with a capitalized initial word. The table below explains briefly what the lines in the script above are for. See docker.com for the full documentation of the Dockerfile format.

Label	Purpose of line
FROM	This line refers to the point of departure for the image. In this case it refers to another Docker image that contains the Matlab runtime library and is necessary to execute compiled Matlab applications.
LABEL	Store a label in the image for administrative purposes (optional)
COPY	Copy file to relative location in image. "." implies that the file is copied to the root of the image.
RUN	Execute this command inside the Docker image during the build stage.
CMD	Execute this command inside the Docker image during the run stage.

Once the dockerfile is in place in the folder sh_startSimpleMailbox, the command:

sudo docker build -t mailbox .

creates the image. The option "-t mailbox" is optional, but tags the image as "mailbox" which makes it easier to refer to it when running the image. The command typically produces this output:



4.7 Run the Docker image

Now that the image has been built. It can be run with the command:

sudo docker run -it -p 4444:4444 mailbox

The command is created from the following parts:

sudo	Excute with administrative privileges
docker run	command for running a container

-it	show the output in the current terminal
-p 4444:4444	forward port 4444 to the container (as 4444)
mailbox	tag of image to use

The output should look like:

5 Multi container Docker images

5.1 Introduction

Until now we have been dealing with Docker images based on a single compiled Matlab application. Applications often consist of multiple services working together for functionality or scaling purposes.

For this type of applications, the deployment process is even more complex, and operators should look for a deployment method that is efficient and reproducible.

Like Docker is used to package and deploy single container applications (see Figure 1 on page 5), docker-compose is used for managing multi-container applications. Figure 2 illustrates the build process. The starting point for the process is the docker images, like discussed in chapters 0 and 4.

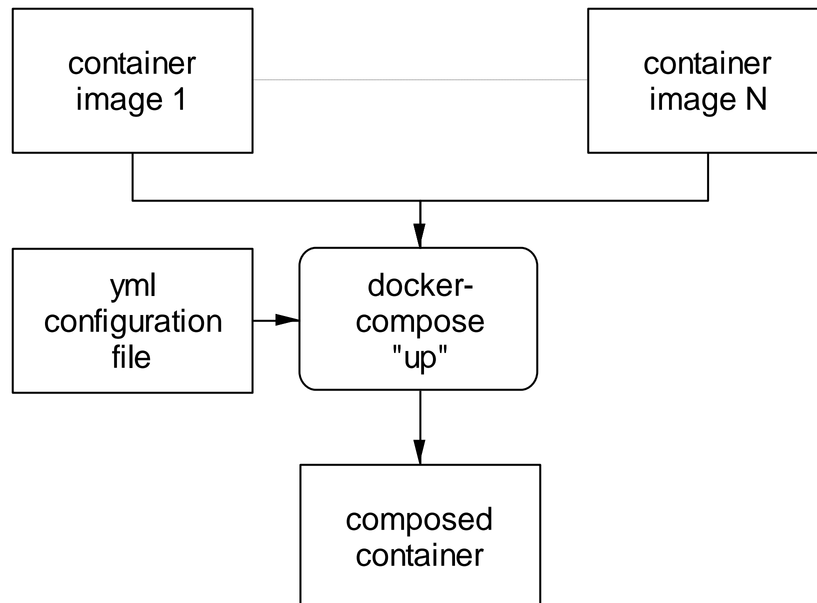


Figure 2: *multi-container application, created from multiple container images, as specified in .yml file*

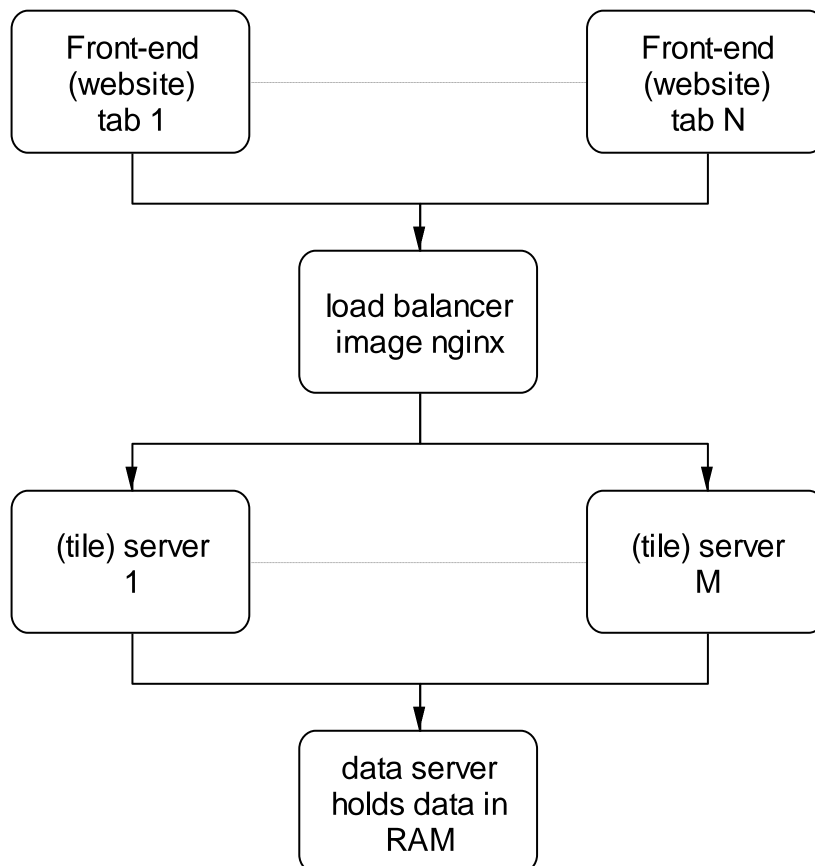
With Docker compose, docker images created from Matlab programs can be combined with “off the shelf” images that deliver valuable functionality, like logging, database functionality or load balancing, just to mention a few.

5.2 Example application

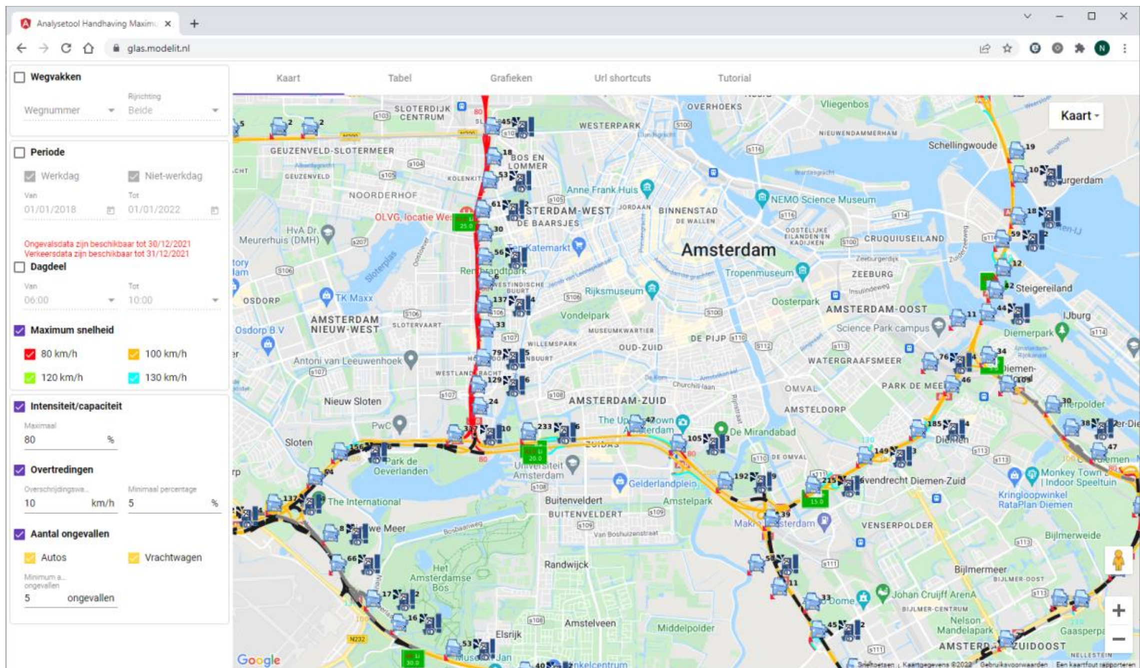
As an example of a multi container application we use a web portal that can be accessed at glas.modelit.nl. The front-end of the application is coded in Angular. The back-end is coded in Matlab and runs in compiled mode in containers on a Linux server.

5.2.1 Organization of the application

The web portal has a front-end that features a number of tabs. The front-end communicates with a back-end that holds all data and does the required computations. The main tab of the portal shows a map that is constructed from multiple 256x256 tiles that depend on the selection settings and are created on the fly. To speed up graphical feedback when panning or zooming on the map multiple instances of the tile server that create these tiles run in parallel. The front-end communicates with a load balancer that distributes requests over these instances.
A single process is running that holds all required data in RAM. The tile server process consults this data through http.



5.2.2 *Front-end*



Analysetool Handhaving Maxim
glas.modelit.nl

Wegvakken
Wegnummer | Rijrichting: Beide

Periode
Werkdag | Niet-werkdag
Van 01/01/2018 | Tot 01/01/2022

Ongevalsdata zijn beschikbaar tot 30/12/2021
Verkeersdata zijn beschikbaar tot 31/12/2021

Dagdeel
Van 06:00 | Tot 10:00

Maximum snelheid
80 km/h | 100 km/h | 120 km/h | 130 km/h

Intensiteit/capaciteit
Maximaal 80 %

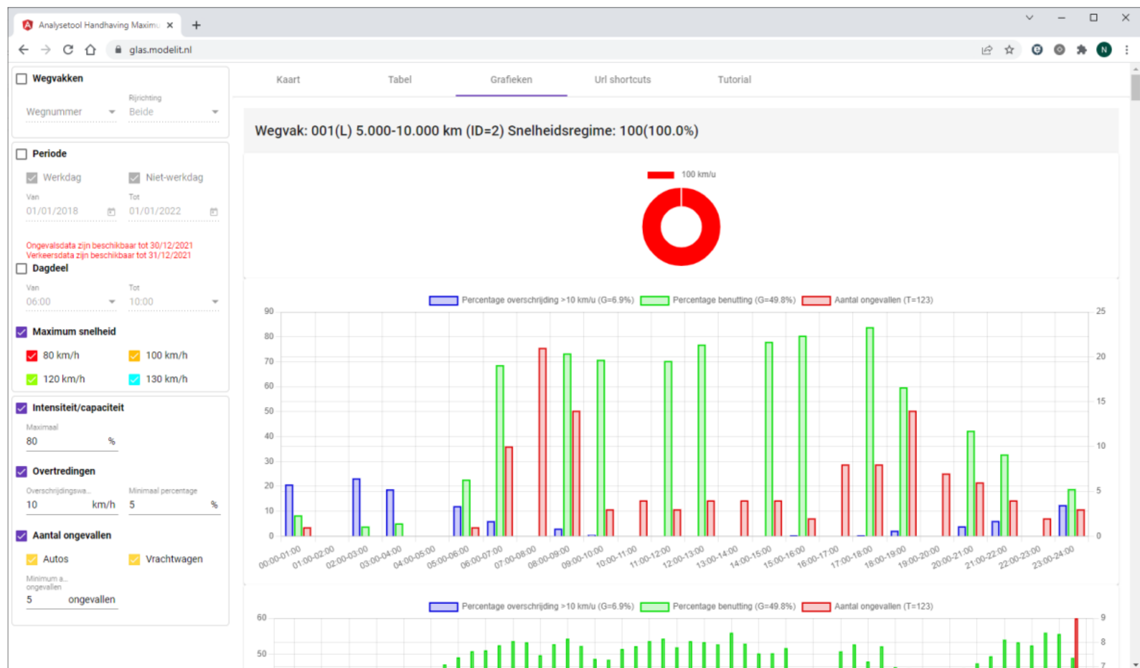
Overtredingen
Overschrijdingswa... 10 km/h | Minimaal percentage 5 %

Aantal ongevallen
Autos | Vrachtwagen
Minimum a... ongevallen 5 ongevallen

Kaart | Tabel | Grafieken | Url shortcuts | Tutorial

Items per page: 100   1 – 100 of 1518

Selectie	ID	Weg	Richting	DVKlet	KM_min	KM_max	Regime_dag	Regime_nacht	Ong_auto	Ong_vracht	Ong_totaal	Perc_10	Benutting
☐	1	001	L	#	3.026	5	70-100	70-100	64	0	64	2.6	48.6
☑	2	001	L	#	5	10	100	100	123	0	123	6.7	50
☑	3	001	L	#	10	15	100-130	100-130	136	4	140	6.3	47.9
☑	4	001	L	#	15	20	50-130	50-130	57	1	58	8.1	46.7
☑	5	001	L	#	20	25	100-130	100-130	96	2	98	6.8	46.8
☑	6	001	L	#	25	30	80-130	80-130	133	8	141	7.8	47.9
☑	7	001	L	#	30	35	80-130	80-130	85	7	92	10.6	47.8
☐	8	001	L	#	35	40	120-130	130	102	2	104	1.5	48.2
☐	9	001	L	#	40	45	100-130	100-130	99	15	114	1.4	51.8
☐	10	001	L	#	45	50	100-130	100-130	119	15	134	3.5	53.6
☐	11	001	L	#	50	55	120-130	120-130	359	9	368	2.4	51.8
☐	12	001	L	#	55	60	50-130	50-130	102	7	109	1.5	51.9
☐	13	001	L	#	60	65	80-130	80-130	57	2	59	4.3	48.6
☐	14	001	L	#	65	70	80-130	80-130	55	3	58	3.9	49.6
☐	15	001	L	#	70	75	130	130	30	4	34	2.7	48.8
☐	16	001	L	#	75	80	130	130	47	4	51	1.7	49.4
☐	17	001	L	#	80	85	80-130	80-130	40	4	44	2.5	45.9
☐	18	001	L	#	85	90	50-130	50-130	83	4	87	1	46.2
☐	19	001	L	#	90	95	120	120	61	7	68	0.6	46.1
☐	20	001	L	#	95	98.991	120	120-130	55	4	59	1.1	45.2
☐	21	001	L	#	104	105	120	130	22	1	23	0.8	45.3
☐	22	001	L	#	105	110	120-130	120-130	61	6	67	1.4	45.9
☐	23	001	L	#	110	115	120-130	130	83	4	87	3.3	46.7
☐	24	001	L	#	115	120	120-130	120-130	67	5	72	3.3	46.6
☐	25	001	L	#	120	125	120-130	120-130	49	13	62	3.1	46.7
☐	26	001	L	#	125	130	130	130	47	2	49	2.4	46.6
☐	27	001	L	#	130	135	130	130	36	5	41	3.9	48.1
☐	28	001	L	#	135	140	130	130	27	6	33	4.2	44.3
☐	29	001	L	#	140	141.692	130	130	7	0	7	2.8	41.7
☐	30	001	L	#	154.76	155	120	120	0	0	0	-1	-1



5.3 Steps for creating a multi-container application

Summary
Some applications require multiple containers to run simultaneously. For example, an application may rely on a compiled Matlab application, a database and a load balancer, each running in a separate container. The command "docker-compose" automates the process of building, starting and stopping multi-container Docker applications.

Steps
Next steps are required to manage a multi-container application.
See also: https://docs.docker.com/compose/

Create a YAML file	See also: https://docs.docker.com/compose/compose-file/ "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." In section 5.3.1 a sample YAML file is shown. This example contains a tileserver (hotspotViewer, accessible externally on port 8081), a data service that is accessible from the tileserver process (hotspotservice, accessible internally on port 6061), and a load balancer (nginx, accessible externally on port 6060)

Create all Docker image(s)	This command creates all docker images that are required for the application, as specified in the YAML file. Navigate to the folder with the YAML file, open a terminal, then execute: `sudo docker-compose build` This command builds the Docker images specified in the YAML file.

Run Docker container(s)	This command starts all containers, using the parameters specified in the YAML file. Execute:
	
	`sudo docker-compose up`
	or:
	`sudo docker-compose up --scale hotspotviewer=4`
	
	The addition "--scale hotspotviewer=4" forces 4 instances of hotspotviewer to be started.

Stop all containers	Execute: `sudo docker-compose stop` or: apply ctrl + C inside the terminal. This with halt the application as well.

5.3.1 *yml script and build command*

```yaml
version: '2'
services:
  hotspotviewer:
    build:
      context: sh_hotspotViewer/.
    image: hotspotviewer
    volumes:
      - /home/modelit/hotspots/data:/data
    environment:
      - hostname="0.0.0.0"
      - port=8081
    expose:
      - "8081"
  hotspotservice:
    build:
      context: sh_hotspotService/.
    image: hotspotservice
    volumes:
      - /home/modelit/hotspots/data:/data
    ports:
      - "6061:6061"
    environment:
      - hostname="0.0.0.0"
      - port=6061
  nginx:
    image: nginx:latest
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - hotspotviewer
      - hotspotservice
    ports:
      - "6060:6060"
```

6 Appendix: createDockerfile, example use

6.1 Usage of createDockerfile

The utility "createDockerfile.m" is included in the embedded webserver toolbox. It is used to create a Dockerfile for a compiled Matlab application. This appendix contains an example of how the utility is used and what output it produces.

As an alternative to using the utility the Dockerfile can also be created manually using "example output" as a template.

6.2 Source code

See createDockerfile.m as included with embedded webserver toolbox.

```
>> help createDockerfile
  Generate and write a Dockerfile

  CALL:
    createDockerfile(targetdir, target, files, varargin)

  INPUT:
    targetdir: <string>
    target: <string>
      the application to run.
    files: <cellstr>
      file to include in the image, such as modelit.jar etc.
    varargin: <any[]>
      input arguments for the target, should appear in the same
      order as the function signature of the target.
```

6.3 Example input

```
createDockerfile(targetdir, 'hotspotViewer',...
    {'javaclasspath.txt','log4j.properties',...
    'modelit.jar','MatlabHTTPserver.jar',...
    'MatlabHTTPclient.jar','xmltoolbox.jar'},...
    'port', '8081',...
    'host', '0.0.0.0');
```

6.4 Example output

The example input creates a file in the "Dockerfile" in the folder "targetdir" with the following content:

```
FROM matlabruntime/r2021b/release/update0/c0000000000000000
##############################################
ENV port=8081
ENV host=0.0.0.0
LABEL Maintainer="Maintainer@company.ext>"
LABEL Description="MATLAB R2021b hotspotViewer"
LABEL Vendor="Modelit"
LABEL Web="http://www.modelit.nl"
```

```
LABEL Version="R2021b"
################################################
COPY run_hotspotViewer.sh .
COPY hotspotViewer .
COPY javaclasspath.txt .
COPY log4j.properties .
COPY modelit.jar .
COPY MatlabHTTPserver.jar .
COPY MatlabHTTPclient.jar .
COPY xmltoolbox.jar .
################################################
RUN chmod a+x ./hotspotViewer
RUN chmod a+x ./run_hotspotViewer.sh
################################################
CMD ["sh", "-c", "./run_hotspotViewer.sh /opt/matlabruntime/v911
${port} ${host}"]
```

7 Useful links

[1]	Docker explained	https://www.docker.com/resources/what-container
[2]	Docker cheat sheet	https://www.pdocker network ls adok.fr/en/blog/do-you-have-all-it-takes-to-use-docker
[3]	Package MATLAB Stan dalone Applications into Docker Images	https://nl.mathworks.com/help/compiler/package-matlab-standalone-applications-into-docker-images.html
[4]	Docker-compose	https://docs.docker.com/compose/
[5]	Docker Desktop for Windows	https://hub.docker.com/editions/community/docker-ce-desktop-windows