Duckietown Architecture and Development
PART A

Duckietown Shell
Duckietown Shell (dt-shell) is a pure Python utility for Duckietown. The idea is that most of the functionality is implemented as Docker containers, and dt-shell provides a nice interface for that, so that user should not type a very long docker run command line.

1.1. Dependencies
Installing dt-shell via pip will also install the following packages
- GitPython - A python library used to interact with Git repositories
- texttable - A module to generate a formatted text table, using ASCII characters.

1.2. Installation
You can install dt-shell by running

```bash
$ pip install duckietown-shell
```

1.3. Run the shell
You can launch the shell by running the command

```bash
$ dt
```
You should see the dt-shell prompt

```
dt>
```
dt-shell is modular, and the pip package does not contain commands. The first time you launch the shell, the most recent version of the commands will be automatically downloaded and made available to you. A barebones set of commands will be automatically installed, others will be available but not installed.

### 2.1. List commands

You can list all the commands available in dt-shell by running

```bash
$ dt> commands [--core] [--installed] [--installable]
```

You should see something like the following

```
dt> commands
Core commands:
  commands
  install
  uninstall
  update
  version
  exit
  help

Installed commands:
  <empty>

Installable commands:
  aidol8
  logs
```

Use the arguments `--core`, `--installed`, and `--installable` to filter the commands.

### 2.2. Update list of commands

Run the command `dt> update` to download the most recent version of the commands available.

**Note:** dt-shell uses git to update the commands. If you get an error, please make sure that git is installed.

### 2.3. Install a new command

Run `dt> commands --installable` to see the list of commands you can install.
Run `dt> install command_name` to install a new command.

### 2.4. Remove a command

Run `dt> commands --installed` to see the list of commands already installed. Run `dt> uninstall command_name` to remove a command.
Custom commands

dt-shell downloads all the commands from the git repository duckietown/duckietown-shell-commands. If you want to contribute to the shell and create new commands, please follow the instructions in the section Section 3.3 - Create a new command.

3.1. Downloading the commands

You can download the source code of the commands by forking the duckietown-shell-commands repository and pulling your fork locally. The repository can be found at this link.

3.2. Understanding the structure of the repository

Move to the directory where you pulled the commands repository. Run

$ tree ./

You should be able to see a hierarchy of Python files that looks like the following,
Let us focus on the directories first. The directory `lib` is reserved to third-party libraries needed by the commands. Each directory (except for `lib`) defines a command. You will recognize some of the core commands (e.g., install, uninstall, version). The name of the directory defines the name of the command.

**Note:** A valid command name can contain only alphanumeric characters plus `_` and `-`.

Let us focus on files now. `dt-shell` determines whether a command `mycommand` is installed by looking for the file `./mycommand.installed.flag`.

All the command directories contain a default `__init__.py` file. Such file is the same for all the commands. A standard template file `__init__.py.template` that you can copy if you wish to implement your own command is available in the main level of the repository.

All the magic happens in the file `command.py`, where the logic of the command is implemented. If you want to learn more about how to create your own command, read the section `Section 3.3 - Create a new command`.

This is all you need to know about the structure of the repository.

### 3.3. Create a new command

**Remark:** Before you can create a new command, you have to pull the source code locally as described in `Section 3.1 - Downloading the commands`. It is also important that you are familiar with the structure of the repository (described in `Section 3.2 - Understanding the structure of the repository`).

You can create a command `mycommand` by creating a directory in the main level of the repository with the following structure

```
./
  ...
  └── mycommand
      ├── __init__.py
      └── command.py
```

You can copy the files `__init__.py.template` and `command.py.template` that you find on the main level of the repo into your new command directory renaming them as `__init__.py` and `command.py` respectively.

This is enough to get your new command in `dt-shell`. Launch `dt-shell` and run

```
$ dt> mycommand
```

You should be able to see something like the following

```
dt> mycommand
You called the "mycommand" command, level 0, with arguments []
```

You can pass arguments to your command. For example, by running
the shell will return

```
dt> mycommand
You called the "mycommand" command, level 0, with arguments ['--arg1', 'value1']
```

When the user types in the command `mycommand` and presses Enter, the file `./my-command/command.py` will be used to serve the request. A valid file `command.py` will have the following structure
from __future__ import print_function
from dt_shell import DTCommandAbs

class DTCommand(DTCommandAbs):
    help = 'Brief description of the command'  # please redefine this
    help message
    # name = <read-only> a string with the name of the command
    # level = <read-only> an integer indicating the level of this com-
    # mand. Follows the directory hierarchy
    # commands = <read-only> a dictionary of subcommands

    @staticmethod
    def command(shell, args):
        # this function will be invoked when the user presses the [Re-
        # turn] key and submits the command
        #
        # shell is the instance of DTShell hosting this command
        # args is a list of arguments passed to the command
        #
        # PUT YOUR CODE HERE
        print(
            'You called the "%s" command, level %d, with arguments %r'
            % (
                DTCommand.name,
                DTCommand.level,
                args
            )
        )

    @staticmethod
    def complete(shell, word, line):
        # this function will be invoked when the user presses the [Tab]
        # key for auto completion.
        #
        # shell is the instance of DTShell hosting this command
        # word is the right-most word typed in the terminal (usual-
        # ly the string the user is trying to auto-complete)
        # line is the entire command
        #
        # PUT YOUR CODE HERE
        return ['suggestion_1', 'suggestion_2']

You can find the same template in the file command.py.template in the main level of the repository. You can recognize the default message printed above by the method command(shell, args) of the command mycommand.
The method `command(shell, args)` is invoked by the object `shell` when the user presses `Enter` and submits the argument `args` to the command. The argument `shell` is the instance of DTShell hosting this command, while `args` is the list of arguments passed to the command.

The method `complete(shell, word, line)` is invoked by the object `shell` when the user presses `Tab` and requests auto-complete. This method should return a list of strings, with each string being a suggestion for completing the command.

### 3.4. Update an existing command

**Remark:** Before you can create a new command, you have to pull the source code locally as described in Section 3.1 - Downloading the commands. It is also important that you are familiar with the structure of the repository (described in Section 3.2 - Understanding the structure of the repository).

You can follow the instructions in the section Section 3.3 - Create a new command to learn how the shell reacts to the input of the user and update your commands accordingly.

### 3.5. Install third-party libraries

You can add third-party libraries to the `./lib/` directory using `git submodule` if you have access to a public `git` repository that you can pull. Alternatively (but not suggested) you can simply copy your library in the `./lib/` directory and push it together with the commands.

`dt-shell` prepends the `./lib/` path to the `$PYTHON_PATH` environment variable before calling your command function. If you have your library in `./lib/my_lib/foo.py`, you can add the line `from my_lib import foo` at the very top of your `command.py` file.

**Note:** `dt-shell` will take care of updating your library when the user runs `dt> update` if you use `git submodule`. 
4.1. Issue with TAB auto-complete

**Symptom:** Pressing TAB does not auto-complete the command but indents the line.

**Resolution:** Update Python to at least 2.7.15 on Mac.
This part describes the Duckietown algorithms and system architecture. We do not go in the software details. The implementation details have been already talked about at length in - Duckietown Software development guide. We do give links to the ROS packages implementing the functionality.
This document describes how to use the tools for querying and downloading logs.

1.1. Installation

1) Option 1: Native
In the Software repo (branch master18) run the following:

```
$ (cd catkin_ws/src/00-infrastructure/ && python setup.py develop --user)
```

2) Option 2: Docker
You can run everything using Docker; see Section 1.5 - Containerized version of log utils.

1.2. Utilities summary

1) `dt-logs-summary`: Querying the DB
You can query the cloud DB using the program `dt-logs-summary`:

```
$ dt-logs-summary
```

You can also include a query string:

```
$ dt-logs-summary query
```

where `query` is a query expressed in the selector language (Section 1.3 - The selector language).

Here is an example output:
$ dt-logs-summary "*dp3tele*"

<table>
<thead>
<tr>
<th>#</th>
<th>Log name</th>
<th>date</th>
<th>length</th>
<th>vehicle name</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20160504-dp3tele1-thing</td>
<td>2016-05-04</td>
<td>294 s</td>
<td>thing</td>
<td>Yes.</td>
</tr>
<tr>
<td>1</td>
<td>20160506-dp3tele1-nikola</td>
<td>2016-05-06</td>
<td>321 s</td>
<td>Nikola</td>
<td>Yes.</td>
</tr>
<tr>
<td>2</td>
<td>2016-04-29-dp3tele-neptunus-0</td>
<td>2016-04-29</td>
<td>119 s</td>
<td>Neptunus</td>
<td>Yes.</td>
</tr>
<tr>
<td>3</td>
<td>20160503-dp3tele1-pipquack</td>
<td>2016-05-03</td>
<td>352 s</td>
<td>Pipquack</td>
<td>Yes.</td>
</tr>
<tr>
<td>4</td>
<td>20160503-dp3tele1-redrover</td>
<td>2016-05-03</td>
<td>384 s</td>
<td>Redrover</td>
<td>Yes.</td>
</tr>
<tr>
<td>5</td>
<td>20160504-dp3tele1-penguin</td>
<td>None</td>
<td>None</td>
<td>Penguin</td>
<td>None.</td>
</tr>
<tr>
<td>6</td>
<td>20160430-dp3tele-3-quackmobile</td>
<td>None</td>
<td>None</td>
<td>Quackmobile</td>
<td>Yes.</td>
</tr>
</tbody>
</table>

2) **dt-logs-download**: Downloading the logs

You can use the command `dt-logs-download` to download the logs:

```
$ dt-logs-download 20160503164104_ayrton
```

```
$ dt-logs-download 20160503164104_ayrton
DT| download.py:109 - download_url_to_file| Download from http://ipfs.duckietown.org:8080/ipfs/QmPrn3V1v6RkkwYf4eUM9Uvnp- perNB9G6JhGkm5a73Mg5x using urllib
...0%, 0.01 MB of 918.2MB, 144.99 MB/s, 0 seconds passed
...14%, 129.55 MB of 918.2MB, 25.54 MB/s, 5 seconds passed
...28%, 263.76 MB of 918.2MB, 26.18 MB/s, 10 seconds passed
...42%, 391.12 MB of 918.2MB, 25.95 MB/s, 15 seconds passed
...56%, 520.31 MB of 918.2MB, 25.91 MB/s, 20 seconds passed
...70%, 649.75 MB of 918.2MB, 25.91 MB/s, 25 seconds passed
...85%, 781.15 MB of 918.2MB, 25.97 MB/s, 30 seconds passed
...98%, 900.80 MB of 918.2MB, 25.68 MB/s, 35 seconds passed
...100%, 918.17 MB of 918.2MB, 25.69 MB/s, 36 seconds passed
```

DT| download.py:115 - download_url_to_file| -> $({DUCKIETOWN_TMP})/downloads/20160503-dp3tele1-ayrton.bag

3) **dt-logs-find**: Get the local filename of a log

To get the location of the bag file, use the command `find`:

```
$ dt-logs-find 20160503164104_ayrton
/mnt/additional/duckietown-tmp/downloads/20160503-dp3tele1-ayrton.bag
```

A typical use case would be the following, in which a script needs a log with which to
work.

Using the `dt-logs-download` program we declare that we need the log. Then, we use `dt-logs-find` to find the path.

```bash
#!/bin/bash
set -ex

# We need the log to proceed
dt-logs-download 20160429223659_neptunus

# Here, we know that we have the log. We use `find` to get the filename.
filename=`dt-logs-find 20160429223659_neptunus`

# We can now use the log
vdir ${filename}
```

4) **dt-logs-copy**: Copy the logs to a local directory

To copy the logs to a local directory, use the command `dt-logs-copy`:

```
$ dt-logs-copy --outdir destination 20160503164104_ayrton
```

5) **dt-logs-details**: More details about the logs

Use `dt-logs-details` to have more details about a log:
6) **dt-logs-thumbnails**: Create thumbnails

Use `dt-logs-thumbnails` to create thumbnails:

```
$ dt-logs-thumbnails [logs query]
```

For example:

```
$ dt-logs-thumbnails 20160503164104_ayrton
Created out-dt-logs-thumbnails/20160503164104_ayrton.thumbnails.jpg
```

Use the option `--write_frames` to write each frame separately.

```
$ dt-logs-thumbnails --write_frames 20160503164104_ayrton
Created out-dt-logs-thumbnails/20160503164104_ayrton/ayrton_camera_node_image_compressed/image-00000.jpg
Created out-dt-logs-thumbnails/20160503164104_ayrton/ayrton_camera_node_image_compressed/image-00001.jpg
Created out-dt-logs-thumbnails/20160503164104_ayrton.thumbnails.jpg
```
Use the option `--all_topics` to do this for all topics, in addition to the camera image.

7) `dt-logs-videos`: Create videos

Use `dt-logs-videos` to create videos:

```
$ dt-logs-videos logs query
```

For example:

```
$ dt-logs-videos 20160503164104_ayrton
Created: out dt-logs-videos/20160503164104_ayrtonvideo.mp4
```

### 1.3. The selector language

Here are some examples for the query language (Table 1.2).

Show all the Ferrari logs:

```
$ dt-logs-summary vehicle:ferrari
```

All the logs of length less than 45 s:

```
$ dt-logs-summary "length:<45"
```

All the invalid logs:

```
$ dt-logs-summary "length:<45,valid:False"
```

All the invalid logs of length less than 45 s:

```
$ dt-logs-summary "length:<45,valid:False"
```

<table>
<thead>
<tr>
<th>expression</th>
<th>example</th>
<th>explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>attribute : expr</code></td>
<td>vehicle:ferrari</td>
<td>Checks that the attribute <code>attribute</code> of the object satisfies the expression in <code>expr</code></td>
</tr>
<tr>
<td><code>&gt; lower bound</code></td>
<td>&gt;10</td>
<td>Lower bound</td>
</tr>
<tr>
<td><code>&lt; upper bound</code></td>
<td>&lt;1</td>
<td>Upper bound</td>
</tr>
<tr>
<td><code>expr1 , expr2</code></td>
<td>&gt;10,&lt;20</td>
<td>And between two expressions</td>
</tr>
<tr>
<td><code>expr1 + expr2</code></td>
<td>&lt;5+&gt;10</td>
<td>Or between two expressions</td>
</tr>
<tr>
<td><code>pattern</code></td>
<td><em>ferrari</em></td>
<td>Other strings are interpreted as wildcard patterns.</td>
</tr>
</tbody>
</table>

### 1.4. Advanced log indexing and generation
1) Shuffle

`expr /shuffle` shuffles the order of the logs in `expr`.

Give me all the `oreo` logs, in random order:

```
$ dt-logs-summary vehicle:oreo/shuffle
```

2) Simple indexing

`expr /[i]` takes the i-th entry.

Give me the first log:

```
$ dt-logs-summary "vehicle:oreo/[0]"
```

Give me a random log; i.e. the first of a random list.

```
$ dt-logs-summary "vehicle:oreo/shuffle/[0]"
```

3) Complex indexing

You can use the exact Python syntax for indexing, including `[a:]`, `[a:b]`, `[a:b:c]`, etc.

Give me three random logs:

```
$ dt-logs-summary "all/shuffle/[0:3]"
```

4) Time indexing

You can ask for only a part of a log using the syntax:

```
expr /{ start : stop }
expr /{ start : }
expr /{: stop }
```

where `start` and `stop` are in time relative to the start of the log.

For example, “give me all the first 1-second intervals of the logs” is

```
all/{:1}
```

Cut the first 3 seconds of all the logs:

```
all/{3:}
```

Give me the interval between 30 s and 35 s:

```
all/{30:35}
```
1.5. Containerized version of log utils

You can run the commands using the Docker container `duckietown/logs`, by using:

```
$ docker run -it duckietown/logs dt-logs-commands arguments
```

1) Recipe: see the logs

For example, see the logs DB:

```
$ docker run -it duckietown/logs dt-logs-summary
```

2) Recipe: copy cloud logs to the current directory

Copy some logs to the current directory.

```
$ docker run -it -v $PWD:/mylogs duckietown/logs dt-logs-copy --outdir /mylogs 20171124170042_yaf
```

Note here we mount the current directory as `/mylogs` using the Docker options:

```
-v $PWD:/mylogs
```

and then we specify it as the output dir for `dt-logs-copy`.

3) Recipe: create thumbnails

Create thumbnails:

```
$ docker run -it -v $PWD:/outdir duckietown/logs dt-logs-thumbnails --outdir /outdir 20171124170042_yaf
```

4) Recipe: create video

Create videos:

```
$ docker run -it -v $PWD:/outdir duckietown/logs dt-logs-videos --outdir /outdir 20171124170042_yaf
```

5) Adding persistence

Mount the directory `/dt-data/DUCKIETOWN_TMP` to have the tmp data persist across invocations:

```
$ mkdir -p dt-tmp-data && docker run -it -v $PWD/dt-tmp-data:/dt-data/ DUCKIETOWN_TMP duckietown/logs dt-logs-videos 20171124170042_yaf
```
UNIT B-2

Configuration

This chapter explains what are the assumptions about the configuration. While the “Setup” parts are “imperative” (do this, do that); this is the “declarative” part, which explains what are the properties of a correct configuration (but it does not explain how to get there).

The tool what-the-duck checks some of these conditions. If you make a change from the existing conditions, make sure that it gets implemented in what-the-duck by filing an issue.

2.1. Environment variables (updated Sept 12)

You need to have set up the variables in Table 2.2.

**Note:** The way to set these up is to add them in the file `~/.bashrc` (export `var="value"`). Do not modify the `environment.sh` script.

<table>
<thead>
<tr>
<th>variable</th>
<th>reasonable value</th>
<th>contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUCKIETOWN_ROOT</td>
<td>~/duckietown</td>
<td>Software repository</td>
</tr>
<tr>
<td>DUCKIEFLEET_ROOT</td>
<td>~/duckiefleet</td>
<td>Where to look for class-specific info</td>
</tr>
<tr>
<td>DUCKIETOWN_DATA</td>
<td>~/duckietown-data</td>
<td>The place where to look for logs.</td>
</tr>
<tr>
<td>DUCKIETOWN_TMP</td>
<td></td>
<td>If set, directory to use for temporary files.</td>
</tr>
<tr>
<td>DUCKIETOWN_CONFIG_SEQUENCE</td>
<td>defaults:baseline:vehicle:user</td>
<td>The configuration sequence for EasyN</td>
</tr>
</tbody>
</table>

1) Duckietown root directory **DUCKIETOWN_ROOT**

2) Duckiefleet directory **DUCKIEFLEET_ROOT**

For Fall 2017, this is the the repository duckiefleet.
For self-guided learners, this is an arbitrary repository to create.
2.2. The “scuderia” (vehicle database)

The system needs to know certain details about the robots, such as their host names, the name of the users, etc.

This data is contained in the ${DUCKIEFLEET_ROOT}/robots/{your_branch} directory, in files with the pattern robot name .robot.yaml.

The file must contain YAML entries of the type:

```
owner:  ID of owner
username:  username on the machine
hostname:  host name
description:  generic description
log:
    date :  comment
    date :  comment
```

A minimal example is in Listing 2.2.

```
owner:  censi
hostname:  emma
username:  andrea
description:  Andrea's car.
log:
    2017-08-01:  >  
        Switched RPI2 with RPI3.
    2017-08-20:  >  
        There is something wrong with the PWM hat and the LEDs.
```

Listing 2.2. Minimal scuderia file emma.robot.yaml

Explanations of the fields:

- **hostname**: the host name. This is normally the same as the robot name.
- **username**: the name of the Linux user on the robot, from which to run programs.
- **owner**: the owner's globally-unique Duckietown ID.

2.3. The machines file

Make sure you already set up ROS (unknown ref opmanual_duckiebot/build-repo)
Activate ROS:

```
$ cd ~/duckietown
$ source environment.sh
```

The `machines` file is created from the scuderia data using this command:

```
$ rosrunduckieteamcreate-machines
```

### 2.4. People database

1. The globally-unique Duckietown ID

This is a globally-unique ID for people in the Duckietown project. It is equal to the Slack username.

   comment
   There is no Slack username anymore, so we should change this to some other convention. -AC

### 2.5. Modes of operation

There are 3 modes of operation:

1. **MODE-normal**: Everything runs on the robot.
2. **MODE-offload**: Drivers run on the robot, but heavy computation runs on the laptop.
3. **MODE-bag**: The data is provided from a bag file, and computation runs on the laptop.

<table>
<thead>
<tr>
<th>mode name</th>
<th>who is the ROS master</th>
<th>where data comes from</th>
<th>where heavy computation happen</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE-normal</td>
<td>duckiebot</td>
<td>Drivers on Duckiebot</td>
<td>duckiebot</td>
</tr>
<tr>
<td>MODE-offload</td>
<td>duckiebot</td>
<td>Drivers on Duckiebot</td>
<td>laptop</td>
</tr>
<tr>
<td>MODE-bag</td>
<td>laptop</td>
<td>Bag file</td>
<td>laptop</td>
</tr>
</tbody>
</table>
We use Makefiles to describe frequently-used commands.

### 3.1. User guide

The command

```
$ make all
```

displays the help for each command.

### 3.2. Makefile organization

There is one Makefile at the root of the Software repository, which includes other makefiles in the directory Makefiles/:

```
Makefile
Makefiles/
    Makefile.build.mk
    Makefile.demos.mk
    Makefile.docker.mk
    Makefile.generate.mk
    Makefile.hw_test.mk
    Makefile.maintenance.mk
    Makefile.openhouse.mk
    Makefile.stats.mk
    Makefile.test.mk
```

Each child Makefile is called `Makefile. section.mk` and it should contain only targets of the form `section - name`.

For example, `Makefile.stats.mk` contains the targets “stats, stats-easy_node, stats-easy_logs“.

The target called `section` should provide an help for the section.

For example, when you run `make build`, you see:
### Building commands

Commands to build the software.

- `make build-machines` : Builds the machines file.
- `make build-machines-clean` : Removes the machines file.
- `make build-clean` : Clean everything.

The output should be valid Markdown, so that it can be included in this documentation.
4.1. Installation
Pull the branch 1710-place-recognition

$ cd duckietown
$ git pull

Source environment:

$ source environment.sh

Remove previous installations:

$ sudo apt remove ipython ipython-notebook

Then run:

$ pip install --user jupyter IPython==5.0

Check the versions are correct:

$ which ipython
/home/andrea/.local/bin/ipython

Check the version is correct:

$ ipython --version
5.0.0

4.2. Configuration
Set a password:

$ jupyter notebook password

4.3. Running it
4.4. Extra configuration for virtual machines

Create a configuration file:

```bash
$ jupyter notebook --generate-config
```

Edit the file `~/.jupyter/jupyter_notebook_config.py`. Uncomment and change the line:

```
#c.NotebookApp.ip = 'localhost'
```

Into:

```
c.NotebookApp.ip = '*'
```
This chapter describes formally what makes a conforming ROS package in the Duckietown software architecture.

5.1. Naming
☐ For exercises packages, the name of the package must be `package_handle`.

5.2. `package.xml`
☐ There is a `package.xml` file.
✔ Checked by `what-the-duck`.

5.3. Messages
☐ The messages are called ....

5.4. Readme file
☐ There is a `README.md` file
✔ Checked by `what-the-duck`.

5.5. Launch files
☐ there is the first launch file

5.6. Test files
UNIT B-6

Road Release Process

KNOWLEDGE AND ACTIVITY GRAPH

**Requires:** You have implemented a new feature or improved an existing feature in your branch `devel-` *project_name*

**Requires:** A robot that is configured and able to follow lanes in Duckietown according to instructions in Unit A-23 - Checkoff: Navigation

**Results:** Your branch gets merged into `master` and doesn’t break anything

This page is about what to do once you have developed and tested your new or improved feature and you want to contribute it back to the `master` branch.

**Note:** If your branch is not merged into `master` and passes all the tests it will be lost forever.

6.1. **Merge Master into your branch**

$ git merge origin master

6.2. **Unit Tests**

6.3. **Continuous Integration**

6.4. **Regression Tests**

If you have developed a perception module you should define a set of regression tests that ensure that it is working properly.

6.5. **Simulation Tests**

6.6. **Hardware-in-the-loop (HWIL)**

If you have been doing your development on your laptop or some other computer, the time is now to put the code on the RasPI and test.

6.7. **Road Test**

Verify that the previous functionality of the robot is preserved. For now repeat the instructions in Unit A-23 - Checkoff: Navigation, and ensure that the basic lane following and indefinite navigation abilities are preserved.
6.8. Make a Pull Request

Once all of the tests are passed and you are sufficiently convinced that your code is not going to break the system, you should make a Pull Request by going [here](#). For the base branch put `master` and for the compare branch put your branch.
UNIT B-7

Duckietown ROS Guidelines

7.1. Node and Topics
In the source code, a node must only publish/subscribe to private topics.
In rospecpp, this means that the topic argument of rospecpp.Publisher and rospecpp.Subscriber should always have a leading ~. ex: ~wheels_cmd, ~mode.
In rospy, this means that the node handle should always be initialized as a private node handle by supplying with a "~" agrument at initialization. Note that the leading "~" must then be obmitted in the topic names of. ex:

```cpp
ros::NodeHandle nh_("~");
sub_lineseglist_ = nh_.subscribe("lineseglist_in", 1, &GroundProjection::lineseglist_cb, this);
pub_lineseglist_ = nh_.advertise<duckietown_msgs::SegmentList> ("lineseglist_out", 1);
```

7.2. Parameters
All the parameters of a node must be private parameters to that node.
All the nodes must write the value of the parameters being used to the parameter server at initialization. This ensures transparency of the parameters. Note that the `get_param(name,default_value)` does not write the default value to the parameter server automatically.
The default parameter of pkg_name/node_name should be put in ~/duckietown/catkin_ws/src/duckietown/config/baseline/pkg_name/node_name/default.yaml. The elemental launch file of this node should load the parameter using <rosparam>.
| Note: The above is deprecated. The configuration is handled differently.

7.3. Launch file
Each node must have a launch file with the same name in the launch folder of the package. ex: joy_mapper.py must have a joy_mapper.launch. These are referred to as the elemental launch files.
Each elemental launch file must only launch one node.
The elemental launch file should put the node under the correct namespace through the veh arg, load the correct configuration and parameter file throught config and param_file_name args respectively. veh must not have a default value. This is to ensure the user to always provide the veh arg. config must be default to baseline and param_file_name must be default to default.
When a node can be run on the vehicle or on a laptop, the elemental launch file should provide a local arg. When set to true, the node must be launch on the launching machine, when set to false, the node must be launch on a vehicle through the machine attribute.
A node should always be launched by calling its corresponding launch file instead of using `rosrun`. This ensures that the node is put under the correct namespace and all the necessary parameters are provided.

Do not use `<remap>` in the elemental launch files.

Do not use `<param>` in the elemental launch files.

### 7.4. Downloading resources

Use this recipe if you need to download things:

```python
from duckietown_utils.download import download_if_not_exist
url = 'https://www.dropbox.com/s/bzezpwlufu4b0/frame0002.jpg?dl=0'
f = 'local.jpg'
download_if_not_exist(url, f)
```

(Do not commit JPGs and other binary data to the Software repository.)
8.1. Implementation

Drivers:

(unknown ref code_docs/adafruit_drivers)

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/adafruit_drivers'.

Location not known more precisely.

Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

(unknown ref code_docs/pi_camera)

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/pi_camera'.

Location not known more precisely.

Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

Operator interface:

(unknown ref code_docs/joy_mapper)

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/joy_mapper'.

Location not known more precisely.

Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

8.2. Camera
8.3. Actuators

8.4. IMU
UNIT B-9
Parallel autonomy
UNIT B-10
Lane control

10.1. Implementation
Perception:

(unknown ref code_docs/anti_instagram)

previous warning next (6 of 25) index

warning

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/anti_instagram'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

(unknown ref code_docs/ground_projection)

previous warning next (7 of 25) index

warning

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/ground_projection'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

(unknown ref code_docs/line_detector)

previous warning next (8 of 25) index

warning

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/line_detector'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

(unknown ref code_docs/line_detector2)

previous warning next (9 of 25) index

warning

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/line_detector2'.

Location not known more precisely.
I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/line_detector2'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/lane_filter'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

Control:

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/lane_control'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/dagu_car'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.
UNIT B-11
Indefinite navigation

11.1. Implementation
The packages involved in this functionality are:

I will ignore this because it is an external link.
> I do not know what is indicated by the link '#code_docs/apriltags_ros'.
Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

I will ignore this because it is an external link.
> I do not know what is indicated by the link '#code_docs/fsm'.
Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

I will ignore this because it is an external link.
> I do not know what is indicated by the link '#code_docs/indefinite_navigation'.
Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

I will ignore this because it is an external link.
> I do not know what is indicated by the link '#code_docs/intersection_control'.
I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/intersection_control'.

Location not known more precisely.
Created by function `check_if_any_href_is_invalid` in module `mcdp_docs.check_missing_links`.

Note: we don't discuss the details of the packages here; we just give pointers to them.
UNIT B-12
Planning

12.1. Implementation
The packages involved in this functionality are:

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/localization'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

Note: we don’t discuss the details of the packages here; we just give pointers to them.
UNIT B-13
Coordination

13.1. Implementation

previous warning next (20 of 25) index

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/led_detection'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

previous warning next (21 of 25) index

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/led_emitter'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

previous warning next (22 of 25) index

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/led_interpreter'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.

previous warning next (23 of 25) index

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/led_joy_mapper'.

Location not known more precisely.
Created by function check_if_any_href_is_invalid in module mcdp_docs.check_missing_links.
I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/led_joy_mapper'.

Location not known more precisely.
Created by function `check_if_any_href_is_invalid` in module `mcdp_docs.check_missing_links`.

(unknown ref code_docs/traffic_light)

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/traffic_light'.

Location not known more precisely.
Created by function `check_if_any_href_is_invalid` in module `mcdp_docs.check_missing_links`.

(unknown ref code_docs/rgb_led)

I will ignore this because it is an external link.

> I do not know what is indicated by the link '#code_docs/rgb_led'.

Location not known more precisely.
Created by function `check_if_any_href_is_invalid` in module `mcdp_docs.check_missing_links`.