

The Learning Experiences Developer Manual

Contents

Introduction

- [What is a Duckietown LX?](#)
- [LX Features and Activities](#)

How To - Complete an LX

- [Step 1: Environment Setup](#)
- [Step 2: Using the `dt` workflow](#)
- [Step 3: What's Next?](#)

How To - Create an LX

- [Step 1: Review the LX Structure](#)
- [Step 2: Create a New LX](#)
- [Step 3: Develop Your Activities](#)
- [Step 4: Publish Your LX](#)

Behind the Scenes

- [Digging Deeper: The LX Build Process](#)
- [Digging Deeper: The `dt` Workflow](#)

A Duckietown Learning Experience provides a structured template for using the Duckietown infrastructure to create and work through high-quality learning activities.

The goal of this developer manual is to provide all the tools you need to work through Learning Experiences and then use the skills you develop to create your own custom Duckiebot demos and LX.

What is a Duckietown LX?

Definition

A learning experience is a stand-alone block of activities that is self-contained and ideally can plug and play into a full course of materials.

Learners can complete activities from the Duckietown LX library. This will allow you to dive into the fundamental topics of robotics and then immediately implement behaviors on your Duckiebot using your new skills. See [How To - Complete an LX](#) to jump right in.

Instructors can create additional custom learning experiences using the Duckietown development tools to present course content as interactive notebooks. To cement topics in practice, you can seamlessly integrate additional robot activities and simulated challenges in a preconfigured environment. This prevents students from getting lost in debugging before their project even begins by eliminating extra setup steps and smoothing over complex implementation details. You can see all available features on the [following page](#) and the LX creation tutorial in [How To - Create an LX](#).

Outcomes

Outcomes after completing a learning experience should generally be actionable skills to demonstrate new knowledge.

These skills are verified through the culminating a demo activity in each LX that can be run on a simulated or real world Duckiebot and evaluated in a challenge.

Learning Goals

LX developers should answer the following questions in the LX description contained in each README. Anyone working through an LX should use this information to direct their efforts as they choose and work through LX:

Question	Example
What will the LX learner gain from this experience?	Learners will be able to describe and implement a PID controller for a differential drive robot.
What are the output activities of the experience?	Learners will tune and test their PID controller on a Simulated Duckiebot to achieve a distance traveled goal.
What is the prerequisite knowledge and where can learners find it?	This LX depends on students having completed the Duckietown ROS LX for the fundamentals of using ROS messages.

LX Features and Activities

Let's start by understanding each of the learning experience activities available and how they might be used.

The `duckietown-lx` repository on GitHub contains the learning experiences developed by the Duckietown team - we'll break down the [Object Detection](#) LX as the main example here.

Note

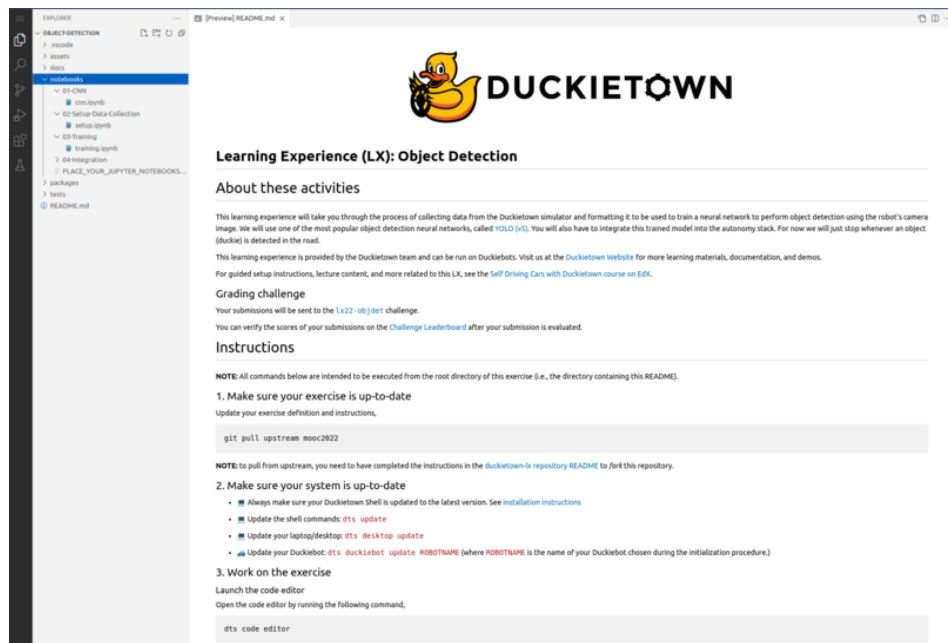
Learning Experiences are run using the `dtc code` workflow as described in [How To - Progress Through an LX](#). This command set gives students a streamlined environment and powerful tools to complete activities.

The following activity types can be implemented with the Duckietown Learning Experience infrastructure:

- [Notebook](#)
- [Workbench Tool](#)
- [Simulated Agent](#)
- [Duckiebot Agent](#)
- [Evaluation](#)

Activity: Notebooks

Learners are immediately presented with the goals and workflow instructions for a learning experience when they use `dtc code editor` to spin up the preconfigured VSCode editor. Installing a local editor is not necessary, and everyone begins with a uniform environment to complete the learning experience. The `notebooks` directory will always contain the first activity.

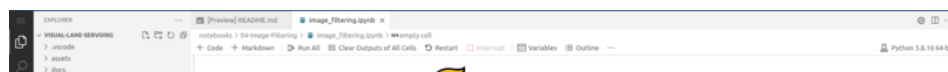


A *Notebook Activity* introduces key concepts within a Jupyter notebook that learners can work through to cement, visualize, and implement their understanding. Tab through the gallery of notebooks below for a few examples of notebook features.

[Image Filtering](#)

[Object Detection](#)

[Hello World](#)



Students should be given instruction within the notebooks on how to progress through the LX activities in order. Every learning experience should also revolve around a main *Learning Goal* (or set of learning goals), documented at the beginning of the [README](#) file.

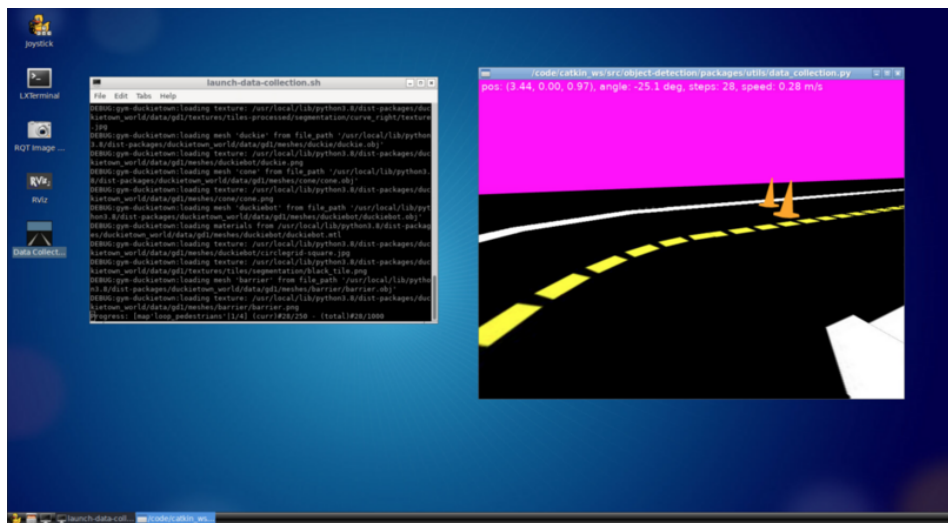
i Example Learning Goal

The Object Detection learning experience will take you through the process of collecting data from the Duckietown simulator and formatting it to be used to train a neural network to perform object detection using the robot's camera image. We will use one of the most popular object detection neural networks, called YOLO (v5). Finally you will integrate this trained model into the autonomy stack to create a Duckiebot agent that stops whenever an object (duckie) is detected in the road.

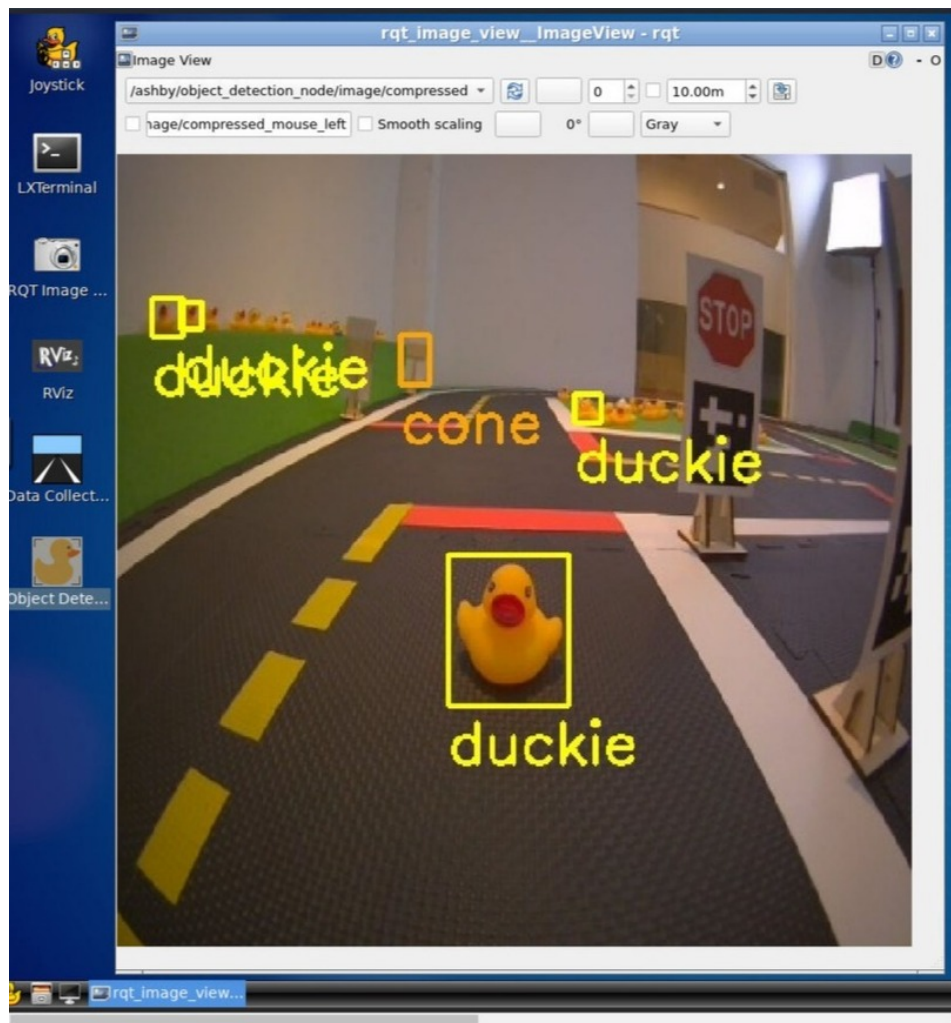
Activity: Workbench Tool

A *Workbench Activity* provides a VNC that is used for running tool, simulation, and agent based activities. This is a fully functional Desktop environment with the Duckietown and ROS dependencies installed and can be started by simply running `dts code workbench`. Instructors can develop custom tools or incorporate any standard ROS tool into the LX activity.

The Object Detection LX uses the workbench environment to run a dataset augmentation tool for learners.

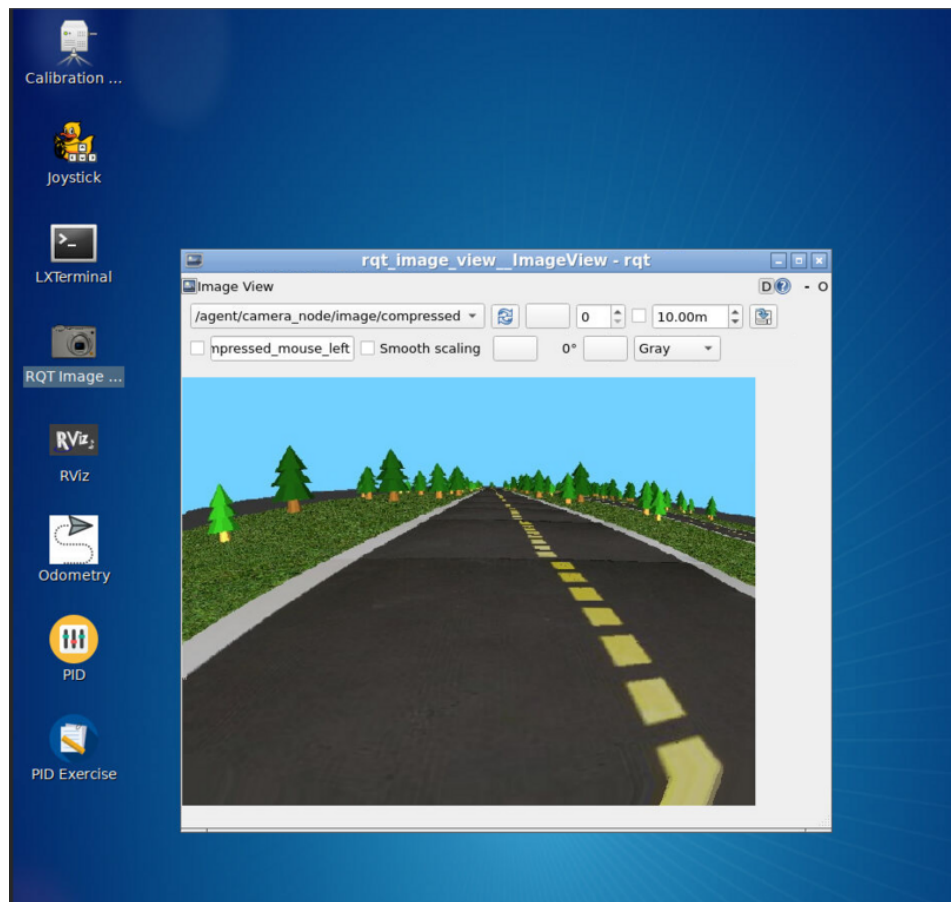


It can also be used to display the Object Detection model results as applied to an image stream from the Duckiebot for visual analysis.



Activity: Simulated Agent

The *Workbench* can also run simulated Duckiebot agents, allowing learners to test their robot behaviors in a virtual environment.

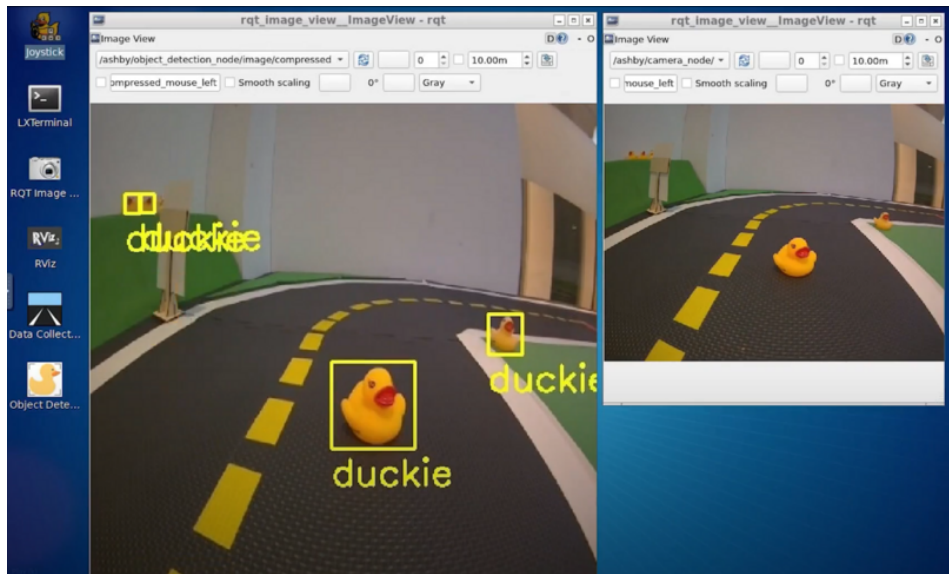


Activity: Duckiebot Agent

Once their solution works in simulation, learners may wish to run their solution on a real-world Duckiebot in a Duckietown environment like the one shown below.



The *Workbench* can interface with the Duckiebot using the ROS network and run connected tools such as keyboard control or *rviz*. Tab through the gallery below to see examples of a variety of tools for interacting with Duckiebot agents.



Activity: Evaluation

Learners can *evaluate* their solutions to Learning Experience challenges locally and submit them to the uploads your agent to the [Duckietown Challenges Server](#) for evaluation on the cloud.

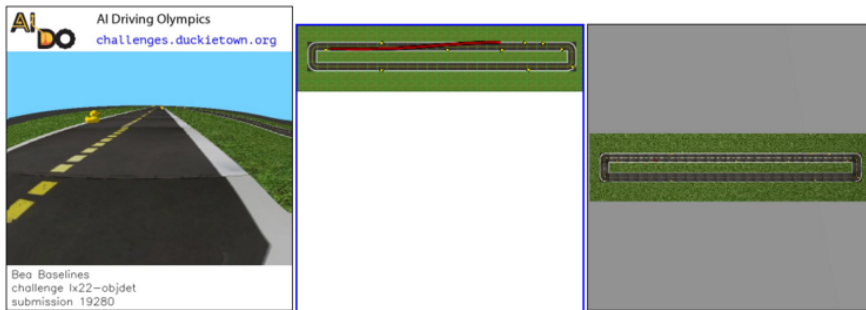
Submission 19280

Submission	19280
Competing	yes
Challenge	lx22-objdet
User	Bea Baselines
Date submitted	3 months, 1 week
Last status update	3 months, 1 week
Complete	complete
Details	Evaluation is complete.
Sisters	
Result	🏆
Jobs	sim: 140851
Next	
User label	minimal-agent
Admin priority	50
Blessing	n/a
User priority	50

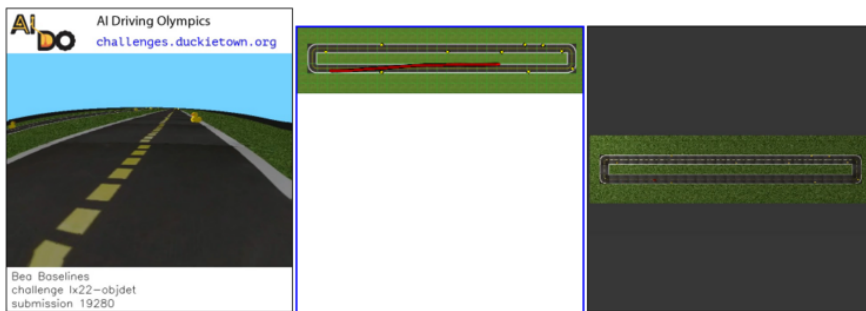
🏆 140851

Click the images to see detailed statistics about the episode.

LF-long-loop-with-duckies-000



LF-long-loop-with-duckies-001



Step 1: Environment Setup

Before working through any Duckietown Learning Experience, you first need to set up your development environment.

Important

- Complete the following setup steps carefully to prevent running into bugs later on. ☐

1 - Requirements

We assume in this manual that you have already set up your Duckietown development environment following the steps in the [Setup - Laptop](#) and [Setup - Account](#) sections of the Duckiebot operation manual.

Install the following dependency libraries,

Open a terminal and run the following command,

```
sudo apt install libnss3-tools
```

We then update the Duckietown shell and the shell commands,

```
pip3 install -U duckietown-shell  
dts update
```

2 - Docker Configuration

After completing Duckietown development setup instructions, add your `docker.io` credentials to the Duckietown shell by running the following command,

```
dts challenges config --docker-username <USERNAME> --docker-password <PASSWORD>
```

Note

These are the `<USERNAME>` and `<PASSWORD>` that you use to log in to DockerHub (hub.docker.io) when setting up Docker in the Duckiebot operation manual.

3 - SSL certificate

We use SSL certificates and TLS encryption to guarantee the highest standard of safety and privacy. Let us set up a local SSL certificate needed to run the learning experience editor inside your browser,

```
dts setup mkcert
```

4 - The `hello-world` LX

We will be walking through the **hello-world** LX in following pages. Fork and clone the [duckietown-lx](#) repository to follow along and complete the tutorial activities. This will give you access to the full library of Duckietown Learning Experiences.

1. To store your own code, while also keeping the ability to pull updates from our version of this repo, create your own fork. Start by pressing "Fork" in the top right corner of [the duckietown-lx repository page on GitHub](#). Your new repository fork that will appear in your GitHub repository list as:

```
<your_username>/duckietown-lx
```

Then clone your new repository, replacing your GitHub username in the command below,

```
git clone -b mooc2022 git@github.com:<your_username>/duckietown-lx
```

2. Now we will configure the Duckietown version of this repository as the upstream repository to sync with your fork. List the current remote repository for your fork,

```
git remote -v
```

Specify a new remote upstream repository,

```
git remote add upstream https://github.com/duckietown/duckietown-lx
```

Confirm that the new upstream repository was added to the list,

```
git remote -v
```

You can now push your work to your own repository using the standard GitHub workflow, and the beginning of every learning experience will prompt you to pull from the upstream repository - updating your exercises to the latest Duckietown version,

```
git pull upstream mooc2022
```

And that's it! You are ready to move on to the next section and start your development journey with the **dt**s **code** workflow.

Step 2: Using the **dt**s **code** workflow

The **dt**s **code** workflow is a set of simple but powerful Duckietown shell commands that you can use to edit, test, and run Duckietown Learning Experiences (LX). These tools can

- spin up a development environment
- run new robot behaviors in the simulator and on a Duckiebot
- submit your results to Duckietown challenges
- and much more

We will gain familiarity with this workflow by walking through the **hello-world** learning experience from the [duckietown-lx repository](#) as an example - instructions on how to fork and clone this repository are located in [Step 1: Environment Setup](#).

Getting started

Complete the following steps to start your development journey with the **dt**s **code** workflow:

□ Step 1

Open a terminal and navigating to the **duckietown-lx/hello-world-lx** directory.

□ Step 2

Glance over the following command list for a preview of your toolkit.

□ Step 3

Continue on to the next page to start your first learning experience!

The **dt**s **code** commands set

dts **code** **build**

Builds a learning experience into a Docker image that can then be run.

dts **code** **edit**

Spins up a browser-based development environment that can be used to work through the Learning Experience (LX) using **VSCode**.

dts **code** **workbench**

Creates a virtual environment with Desktop icons that will allow you to easily run activities, simulate a Duckiebot directed through a virtual world by your control algorithms, or execute a demo on your real world Duckiebot - all with debugging and visualization tools to help you along the way.

dts **code** **evaluate**

Evaluates your solutions to the learning experience activities on your local machine to quickly inform you of your progress.

dts **code** **submit**

Submits your work to the [Duckietown Challenges Server](#) so that you can monitor your results and view the work of other developers around the world.

TODO Update the URL to the challenges server once we move to [duckietown.com](#).

💡 Tip

Remember that in addition to the `dts code` workflow, you also have the complete set of Duckietown development tools at your disposal for building and running the projects within each learning experience.

Check out the [Duckiebot](#) and [DTPProject](#) development pages for more helpful Duckietown shell commands.

dts code build

What does it do?

The `dts code build` command builds the learning experience package into a Docker image that will be used by each of the other commands in the workflow.

The first step to working through any learning experience is to build it.

💡 Hint

Strengthen your iterative development habits by beginning every work session with a fresh build of your LX. This will help ensure that you don't continue development on top of any previous errors.

Prerequisites

Always make sure that your system is up-to-date before starting a new learning experience. If you haven't yet installed the Duckietown shell and configured your development environment, return to the [Step 1: Environment Setup](#) before continuing.

- Update the shell commands:

```
dts update
```

- Update your laptop/desktop:

```
dts desktop update
```

- Update your Duckiebot:

```
dts duckiebot update [ROBOT_NAME]
```

How do I run it?

First, navigate into the directory containing the **hello-world** learning experience (or the root directory of the LX you are working on completing).

`duckietown-lx/hello-world-lx`

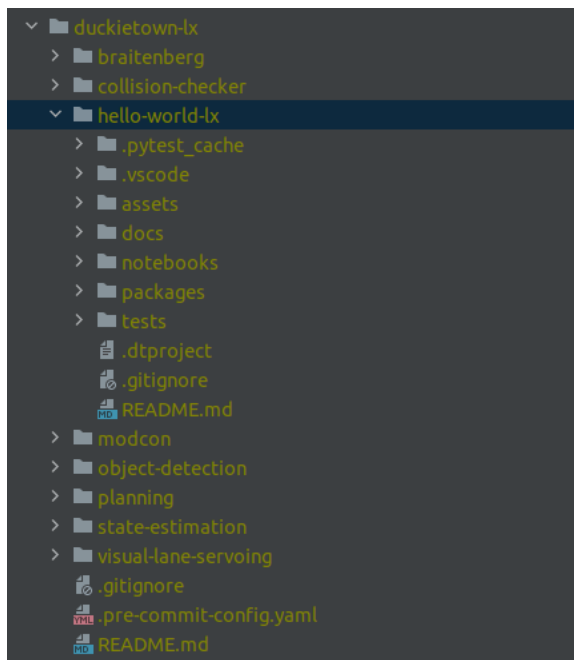


Fig. 1 List of LX directories with the hello-world-lx directory highlighted.

Important

All `dts code` commands should be executed inside the root directory of the learning experience.

Then run

```
dts code build
```

You will see the following message once your LX has built successfully,

```
INFO:dts:Project packaged successfully!

=====
Docker Build Analyzer
Version: 1.1.0
=====

Legend:  [ ] EMPTY LAYER  [ ] BASE LAYER  [ ] < 20.0 MB  [ ] < 75.0 MB  [ ] > 75.0 MB

=====
Final image name: registry-stage2.duckietown.org/kathryn/hello-world-lx:latest-amd64
Base image size: 0.00 B
Final image size: 0.00 B
Your image added 0.00 B to the base image.
-----
Layers total: 0
- Built: 0
- Cached: 0
-----
Time: 13 seconds
Documentation: Skipped
=====

IMPORTANT: Always ask yourself, can I do better than that?
```

Fig. 2 Success message that indicates a successful LX build.

For more information about what is happening during your build process, you can run any `dts code` command in debug mode using the `--debug` flag.

```
dts --debug code build
```

Troubleshooting

If you run into any issues while building the image, you can search the troubleshooting symptoms below or reference the [How to get help](#) section of this manual.

Troubleshooting

SYMPTOM	<code>dts</code> : The path <code>'/home/myuser/not_an_lx_directory'</code> does not appear to be a Duckietown project. : The metadata file <code>'.dtproject'</code> is missing.
RESOLUTION	You need to be in the root directory of the LX in order to run the <code>dts code</code> commands.

What's Next?

Now that you've built the **hello-world** learning experience, continue on to the next page to open the editor and complete your first notebook activities.

Extra Options

⚠ Warning

If this is your first time using the `dts code` workflow, don't worry about the following section just yet. Continue on to the next page to open your first LX activity.

Once you are comfortable with the `dts code` workflow, you may want to use some additional control provided over each command. This section documents each of the flags available to extend the `dts code build` command.

You can also explore the Behind the Scenes - `dts code build` chapter for more details on what happens in the background when you run the `dts code build` command.

Command options

```
usage: dts [-h] [-C WORKDIR] [-H MACHINE] [-u USERNAME] [--no-pull] [--no-cache] [--push]
[--recipe RECIPE] [--registry REGISTRY] [-L LAUNCHER] [-b BASE_TAG] [-v] [--quiet]

optional arguments:
  -h, --help                show this help message and exit
  -C WORKDIR, --workdir WORKDIR
                           Directory containing the project to be built
  -H MACHINE, --machine MACHINE
                           Docker socket or hostname to use
  -u USERNAME, --username USERNAME
                           The docker registry username to use
  --no-pull                 Skip updating the base image from the registry
  --no-cache               Ignore the Docker cache
  --push                   Push the resulting Docker image to the registry
  --recipe RECIPE          Path to use if specifying a custom recipe
  --registry REGISTRY      Docker registry to use
  -L LAUNCHER, --launcher LAUNCHER
                           The launcher to use as entrypoint to the built container
  -b BASE_TAG, --base-tag BASE_TAG
                           Docker tag for the base image. Use when the base image is
                           also a development version
  -v, --verbose             Be verbose
  --quiet                  Be quiet
```

`dts code editor`

What does it do?

The `dts code editor` command provides a local code editor that you will use to work through learning experience notebooks and develop agents to run on your Duckiebot - right in your browser.

How do I run it?

Open the code editor by running the following command

```
dtS code editor
```

Wait for a URL to appear on the terminal, then click on it or copy-paste it in the address bar of your browser to access the **VSCode** powered code editor.

Note

If your Operating System supports it, the page should be opened automatically for you in a new browser tab as soon as it is ready to be opened.

```
INFO:dtS:
You can open VSCode in your browser by visiting the URL:

> https://localhost:32768

VSCode might take a few seconds to be ready...
-----
INFO:dtS:Use Ctrl-C in this terminal to stop VSCode.
```

Fig. 3 Url to access your VSCode editor in the browser.

The first thing you will see is the README document, which should contain the learning objectives for the LX.

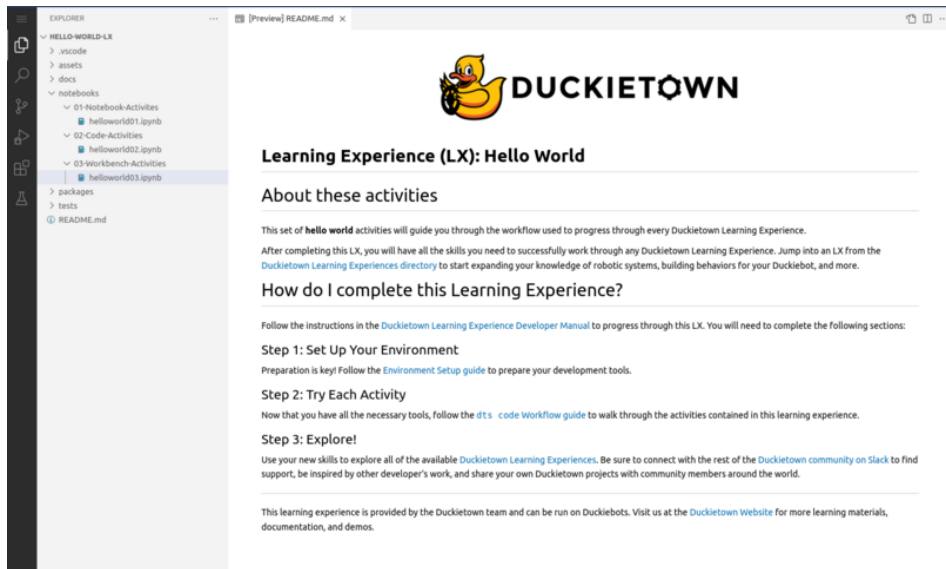


Fig. 4 The VSCode Learning Experience editor.

Once you have read about the learning experience goals in the README document, you can open the **notebooks** directory using the file navigation on the left side of the editor. The activities in the **notebooks** directory contain the main guidance and content of a learning experience.

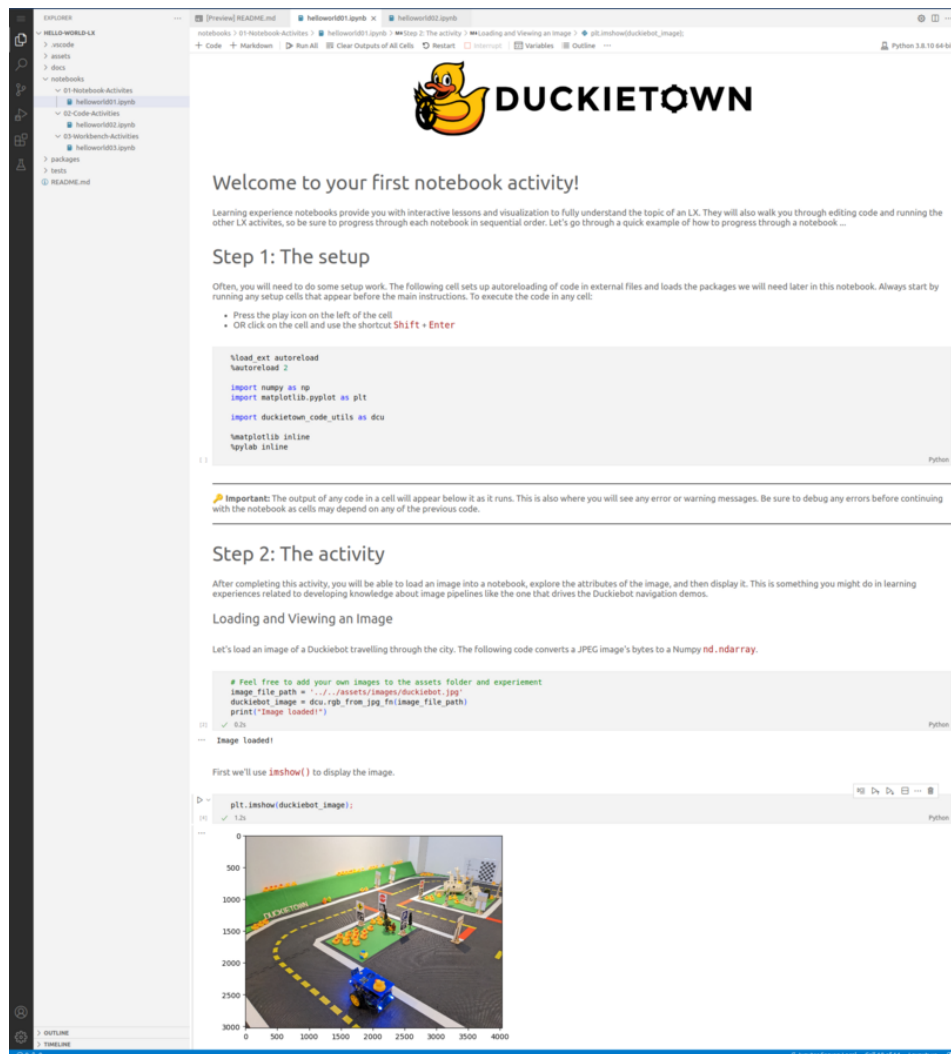


Fig. 5 The first Hello World notebook will guide you through editing and running the activity.

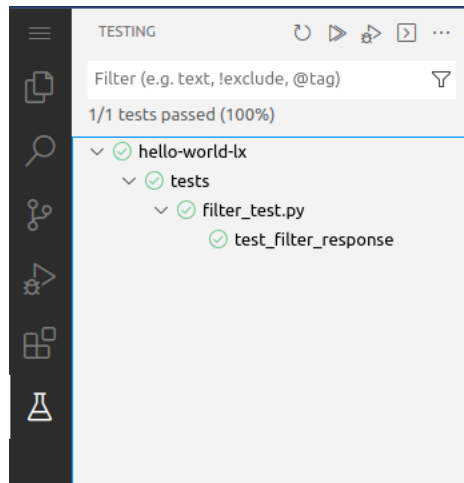
Follow the instructions to complete each notebook in sequence. If you are working through the **hello-world LX**, complete the following notebooks to create an image filter and explore the editor features:

- 01-Notebook-Activities
- 02-Code-Activities

Then return to this page and continue on to the **ds workbench** command.

Hint

Strengthen your test-driven development (TDD) habits by using the Testing interface in the **VSCode** editor to run the provided unit tests for each function you complete in an LX. This will confirm that your solution performs as expected before you run it in simulation or on your Duckiebot. Note that the beaker symbol to open the Testing interface may not appear in the sidebar until after you've opened one of the Python files in the **packages** directory.



Troubleshooting

If you run into any issues using this command, you can search the troubleshooting symptoms below or reference the [How to get help](#) section of this manual.

Troubleshooting	
SYMPTOM	<code>dts</code> : The path <code>'/home/myuser/not_an_lx_directory'</code> does not appear to be a Duckietown project. : The metadata file <code>'dtproject'</code> is missing.
RESOLUTION	You need to be in the root directory of the LX in order to run the <code>dts</code> code commands.

What's Next?

Once you've completed the first two notebooks in the **hello-world** learning experience, continue on to the next page to use the workbench tools and drive a Duckiebot in the Duckietown simulator.

Extra Options

Warning

If this is your first time using the `dts` code workflow, don't worry about the following section just yet. Continue on to the next page to run your first LX activity.

Once you are comfortable with the `dts` code workflow, you may want to use some additional control provided over each command. This section documents each of the flags available to extend the `dts code editor` command.

You can also explore the Behind the Scenes - `dts` code editor chapter for more details on what happens in the background when you run the `dts code editor` command.

Command options

```
usage: dts [-h] [-C WORKDIR] [-u USERNAME] [--distro DISTRO] [--bind BIND] [--no-build]
[--build-only] [--recipe RECIPE] [--image IMAGE] [--plain] [--no-pull] [--keep]
[--impersonate IMPERSONATE] [-v]

optional arguments:
  -h, --help                show this help message and exit
  -C WORKDIR, --workdir WORKDIR
                           Directory containing the project to open the editor on
  -u USERNAME, --username USERNAME
                           The docker registry username to use
  --distro DISTRO           Custom distribution to use VSCode from
  --bind BIND               Address to bind to
  --no-build                Whether to skip building VSCode for this project, reuse last
build instead
  --build-only              Whether to build VSCode for this project without running it
  --recipe RECIPE           Path to a custom recipe to use
  --image IMAGE             Docker image to use as editor (advanced use only)
  --plain                   Whether to skip building VSCode for this project, use plain
VSCode instead
  --no-pull                 Whether to skip updating the base VSCode image from the
registry
  --keep                    Whether to keep the VSCode once done (useful for debugging)
  --impersonate IMPERSONATE
                           Username or UID of the user to impersonate inside VSCode
  -v, --verbose             Be verbose
```

dts code workbench

What does it do?

The `dts code workbench` command runs the workbench portion of a learning experience, which provides a virtual desktop tool (the VNC) with three different purposes,

1. **VNC Visualization Tools:** In any LX notebook activity, you may be directed to use the `dts code workbench` command to open the VNC and run a visualization or calibration tool. This provides a more advanced interface to complement the notebook experience;
2. **Duckietown Simulator:** The simulator interface runs agents developed in the LX on a Duckiebot in a virtual world. Keep an eye out for any duckies that may be wandering around the simulated road;
3. **Duckiebot Agent Interface:** The `dts code workbench` command is also your interface for running agents on your real world Duckiebot.

The focus of the workbench is to provide a streamlined experience for testing and deploying the activities and solutions that you develop while working through a learning experience.

How do I run it?

1. To run an activity visualization or calibration in the VNC, run

```
dts code workbench --sim
```

and follow the instructions in the notebook to select the correct desktop icon and run the tool.

2. To test in simulation, use the command

```
dts code workbench --sim
```

There will be two URLs popping up to open in your browser: one is the direct view of the simulated environment. The other is VNC and only useful for some exercises, follow the instructions in the notebooks to see if you need to access VNC.

This simulation test is just that, a test. Don't trust it fully. If you want a more accurate metric of performance, use the `dts code evaluate` command described on the next page.

3. You can test your agent on the robot using the command,

```
dts code workbench --duckiebot [ROBOT_NAME]
```

This is the modality "all software runs on the robot".

You can also test using

```
dts code workbench --duckiebot [ROBOT_NAME] --local
```

This is the modality “drivers running on the robot, agent running on the laptop.”

If you run into any issues using this command, you can search the troubleshooting symptoms below or reference the [How to get help](#) section of this manual.

Troubleshooting

Troubleshooting	
SYMPTOM	<code>dts</code> : The path '/home/myuser/not_an_lx_directory' does not appear to be a Duckietown project. : The metadata file '.dtproject' is missing.
RESOLUTION	You need to be in the root directory of the LX in order to run the <code>dts code</code> commands.

Troubleshooting	
SYMPTOM	These errors appear: <code>requests.exceptions.HTTPError: 500 Server Error: Internal Server Error for url: http+docker://localhost/v1.43/containers/84ce.../start</code>
	and it is complained that certain ports are in conflict and could not be used.
RESOLUTION	Please check your running docker containers and ports with: <code>docker ps --format "table {{.Names}}\t{{.Image}}\t{{.ID}}\t{{.Ports}}"</code> And stop the ones unnecessary, that occupy the mentioned conflicted ports.

Extra Options

Warning

If this is your first time using the `dts code workflow`, don't worry about the following section just yet. Continue on to the next page to evaluate the solution to your first LX activity.

Once you are comfortable with the `dts code` workflow, you may want to use some additional control provided over each command. This section documents each of the flags available to extend the `dts code workbench` command.

You can also explore the Behind the Scenes - `dts code workbench` chapter for more details on what happens in the background when you run the `dts code workbench` command.

Command options

Usage:

```
$ dts code workbench --sim
$ dts code workbench --duckiebot [ROBOT_NAME]
```

optional arguments:

```
-h, --help            show this help message and exit
-C WORKDIR, --workdir WORKDIR
                        Directory containing the project to bring up
--duckiebot DUCKIEBOT, -b DUCKIEBOT
                        Name of the Duckiebot on which to run the exercise
-s, --simulation, --sim, --simulator
                        Should we run it in the simulator instead of the real robot?
--stop                Just stop all the containers
--local, -l            Should we run the agent locally (i.e. on this machine)?
Important Note: this is not expected to work on MacOSX
--recipe RECIPE        Path to use if specifying a custom recipe
--pull                Should we pull all of the images
--no-cache             Ignore the Docker cache
--bind BIND            Address to bind to (VNC)
--logs LOGS            Use --logs NAME:LEVEL to set up levels. The container names
and their defaults are [agent:Levels.LEVEL_DEBUG
                        manager:Levels.LEVEL_NONE simulator:Levels.LEVEL_NONE
                        bridge:Levels.LEVEL_NONE vnc:Levels.LEVEL_NONE]. The levels are
                        none, debug, info, warning, error.
--log_dir LOG_DIR      Logging directory
-L LAUNCHER, --launcher LAUNCHER
                        Launcher to invoke inside the exercise container (advanced
users only)
--registry REGISTRY    Docker registry to use (advanced users only)
--interactive, -i      Will run the agent in interactive mode with the code mounted
--keep                Do not auto-remove containers once done. Produces garbage
containers but it is very useful for debugging.
--sync                RSync code between this computer and the agent
--challenge CHALLENGE
                        Run in the environment of this challenge.
--scenarios SCENARIOS
                        Uses the scenarios in the given directory.
--step STEP            Run this step of the challenge
--nvidia              Use the NVIDIA runtime (experimental).
```

dts code evaluate

What does it do?

The `dts code evaluate` command runs your code to implement a robot agent at the end of each LX and evaluates it against some set of performance metrics for the Duckiebot. This might be distance travelled in an obstacle avoidance challenge, intersections successful and lawfully handled in the Duckietown city, or any other hurdle that the LX creator may have defined for you.

Details of the evaluation metrics will be outlined in the LX [README](#) file.

How do I run it?

We suggest you evaluate your work locally before submitting your solution. You can do so by running the following command,

```
dts code evaluate
```

This should take a few minutes.

Wait for a URL to appear on the terminal, then click on it or copy-paste it in the address bar of your browser to access the real-time visualization of your evaluation simulation and statistics.

If you run into any issues using this command, you can search the troubleshooting symptoms below or reference the [How to get help](#) section of this manual.

Troubleshooting

Troubleshooting

SYMPTOM	<code>dts</code> : The path <code>'/home/myuser/not_an_lx_directory'</code> does not appear to be a Duckietown project. : The metadata file
---------	---

RESOLUTION

`'dtproject' is missing.`

You need to be in the root directory of the LX in order to run the `dt` code commands.

Extra Options

⚠ Warning

If this is your first time using the `dt` code workflow, don't worry about the following section just yet. Continue on to the next page to submit the solution to your first LX activity.

Once you are comfortable with the `dt` code workflow, you may want to use some additional control provided over each command. This section documents each of the flags available to extend the `dt` code `evaluate` command.

You can also explore the Behind the Scenes - `dt` code `evaluate` chapter for more details on what happens in the background when you run the `dt` code `evaluate` command.

Command options

```
usage: dt [-h] [-C WORKDIR] [-H MACHINE] [-a ARCH] [-u USERNAME] [--recipe RECIPE]
[--no-pull] [--no-cache] [--impersonate IMPERSONATE]
        [-c CHALLENGE] [-L LAUNCHER] [-v]

optional arguments:
  -h, --help                show this help message and exit
  -C WORKDIR, --workdir WORKDIR
                           Directory containing the project to submit
  -H MACHINE, --machine MACHINE
                           Docker socket or hostname where to build the image
  -a ARCH, --arch ARCH      Target architecture for the image to build
  -u USERNAME, --username USERNAME
                           The docker registry username to use
  --recipe RECIPE           Path to use if specifying a custom recipe
  --no-pull                 Skip pulling the base image from the registry (useful when
you have a local BASE image)
  --no-cache                Ignore the Docker cache
  --impersonate IMPERSONATE
                           Duckietown UID of the user to impersonate
  -c CHALLENGE, --challenge CHALLENGE
                           Challenge to evaluate against
  -L LAUNCHER, --launcher LAUNCHER
                           The launcher to use as entrypoint to the submission
  container
  -v, --verbose             Be verbose
```

dt code submit

What does it do?

The `dt` code `submit` command is very similar to the `dt` code `evaluate` command, but instead of evaluating your agent's performance on your local machine, it uploads your agent to the [Duckietown Challenges Server](#) for evaluation on the cloud.

Duckietown challenges server

This is the Duckietown challenges server. For more information, look at duckietown.org.

Challenges

Mooc 2022 challenges Validation challenges Variations on AI-DO challenges "X" AI-DO (full state)

These are the challenges for the 2022 mooc.

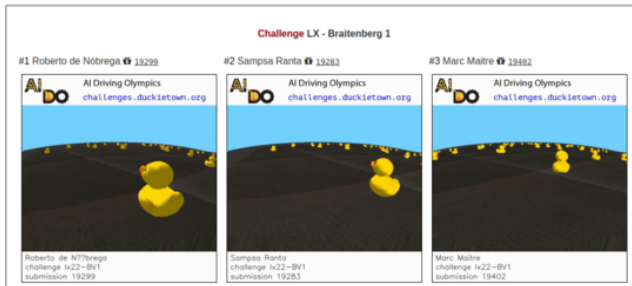


Fig. 6 The Duckietown challenges server displaying results for submission simulations.

How do I run it?

When you are ready to submit your solution to the challenge for your LX, use the following command,

```
dts code submit
```

This will package all of your code and send it to the Duckietown Challenges Server for evaluation. The command will output a URL that you can use to follow your submission and compare your agent with other developers' solutions from all over the world.

If you run into any issues using this command, you can search the troubleshooting symptoms below or reference the [How to get help](#) section of this manual.

Troubleshooting

Troubleshooting

SYMPTOM	<code>dts</code> : The path <code>'/home/myuser/not_an_lx_directory'</code> does not appear to be a Duckietown project. : The metadata file <code>'.dtproject'</code> is missing.
RESOLUTION	You need to be in the root directory of the LX in order to run the <code>dts code</code> commands.

Extra Options

⚠ Warning

If this is your first time using the `dts code` workflow, don't worry about the following section just yet. You now have all the tools to complete your first learning experience - go for it!

Once you are comfortable with the `dts code` workflow, you may want to use some additional control provided over each command. This section documents each of the flags available to extend the `dts code evaluate` command.

You can also explore the Behind the Scenes - `dts code submit` chapter for more details on what happens in the background when you run the `dts code submit` command.

Command options

```
usage: dts [-h] [-C WORKDIR] [-H MACHINE] [-a ARCH] [-u USERNAME] [--recipe RECIPE]
[--no-pull] [--no-cache] [--impersonate IMPERSONATE]
        [-c CHALLENGE] [-L LAUNCHER] [-v]

optional arguments:
  -h, --help                show this help message and exit
  -C WORKDIR, --workdir WORKDIR
                           Directory containing the project to submit
  -H MACHINE, --machine MACHINE
                           Docker socket or hostname where to build the image
  -a ARCH, --arch ARCH      Target architecture for the image to build
  -u USERNAME, --username USERNAME
                           The docker registry username to use
  --recipe RECIPE           Path to use if specifying a custom recipe
  --no-pull                 Skip pulling the base image from the registry (useful when
you have a local BASE image)
  --no-cache                Ignore the Docker cache
  --impersonate IMPERSONATE
                           Duckietown UID of the user to impersonate
  -c CHALLENGE, --challenge CHALLENGE
                           Challenge to evaluate against
  -L LAUNCHER, --launcher LAUNCHER
                           The launcher to use as entrypoint to the submission
container
  -v, --verbose             Be verbose
```

Step 3: What's Next?

We have seen how a few Duckietown shell commands is all we need to learn and develop robotic systems that will perform in the simulator or on a real world robot.

Find more Duckietown Learning Experiences

You can find example LX packages to walk through in the [Duckietown Learning Experiences repository](#).

TODO Link Duckietown LX library page when created

Worried about forgetting the workflow?

Here are a few resources

1. Bookmark this development guide to come back to any time;
2. Run `dts code` in the terminal to display the list of available workflow commands. You can also use the help flag `-h` after any command to get more details and a list of available options;
3. Download the Duckietown `dts code` workflow cheatsheet here to hang up around your workspace: [dts code Workflow Cheatsheet](#);

How to get help

If you run into any issues that can't be solved using the troubleshooting sections in this development manual, you can join the [Duckietown community on Slack](#). There you can request an invitation to the Duckietown Stack Overflow team and find other developers using Duckietown for a wide variety of projects and learning experiences.

Create your own LX

You can begin developing your own custom learning experiences to teach others by continuing on to the next section of this development guide: [How To - Create an LX](#).

Step 1: Review the LX Structure

A learning experience is structured in three parts, each implemented by a different directory. These directories are created for you when running the command `dts lx create` (more on this in the following sections of this manual). When distributed, they will reside in three different Git repositories.

If you are reading this as a learner, you will only need to use the single LX directory provided to you by your instructor or class instructions. All the necessary behind the scenes components to drive the LX activities will be handled automatically by `dts`.

Three parts make a single LX

A Learning Experience (LX) is broken down into three parts:

- A *learner's workspace*;
- A *back-end*;
- A *solution*;

The *learner's workspace*

The *learner's workspace* (also referred to as **the meat**) is the only part that a learner will ever need to interact with. It contains instructions, interactive learning materials (e.g., jupyter notebooks) and boilerplate code that the learner will have to work on, complete, and in some cases submit.

The *back-end*

The *back-end* (also referred to as **the recipe**) is what defines the rules of a learning experience. It contains all those files that are necessary for the *learner's workspace* to function but must be protected from the learner's edits. The **recipe** constitutes the immutable part of a learning experience, and it is shared by all the learners' workspaces and solutions of said LX;

For example, in a robot localization exercise, the back-end might have access to a map of the environment and a trajectory taken by a robot within it. The back-end would use this data to generate simulated sensors' readings and pass those on to the *learner's workspace*. The learner is required to implement a system that can consume these readings and reconstruct the original trajectory taken by the robot. In this case, the back-end must protect the ground-truth trajectory of the robot or else the solution would be exposed to the learner, effectively invalidating the usefulness of any grading.

The *solution*

The *solution* part is simply a copy of the *learner's workspace* but with all the solutions to the activities and exercises already implemented. This is usually used for two main reasons: (i) to make sure that the boilerplate provided in the *learner's workspace* is enough for a learner to implement a correct solution to the problems presented; (ii) to serve as example solutions to problems that are optional or not graded;

Tip

The reason why the **back-end** the **learner's workspace** are also respectively referred to as **the recipe** and **the meat** comes from an analogy with a cooking scenario. The back-end defines the rules, so it is a **recipe** for how the learning experience works, the learner brings the content, the solution, the substance, hence the **meat** that together with the recipe makes a cooked meal, in this case a solved learning experience. Moreover, similarly to how better cuts of meat lead to better tasting meals, better solutions lead to better scores for the learner. Solving a learning experience just became a hunt for the best meat.

Development Workspace

An LX development workspace is set up on running `dts lx create`. This command generates a workspace directory that is named using a filesafe version of the LX name. [Fig. 7](#) shows an example of how an LX development workspace is structured.

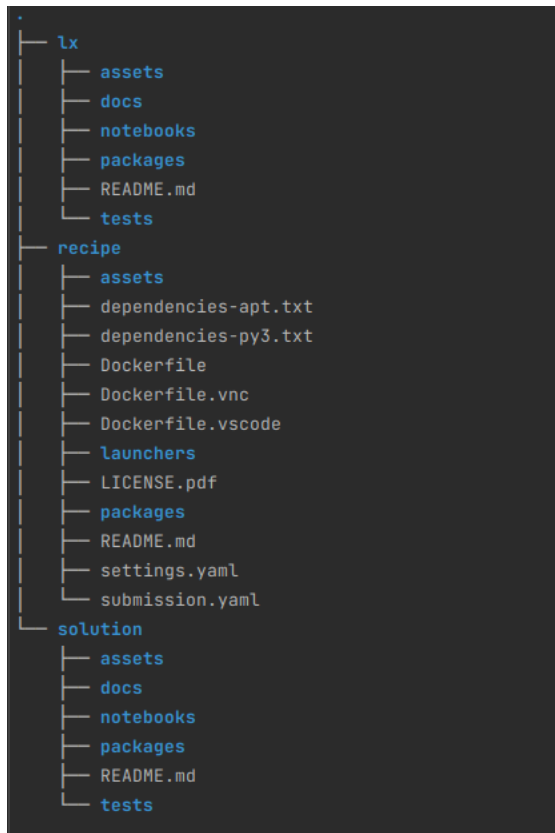


Fig. 7 Directory structure of a LX development project named *How To Robot!*

generated by `dts lx create`.

The learner's workspace (the meat)

The *learner's workspace* is stored inside the directory `lx` (`how-to-robot/lx` in [Fig. 7](#)). [Fig. 8](#) shows an example of what a learner's workspace looks like.

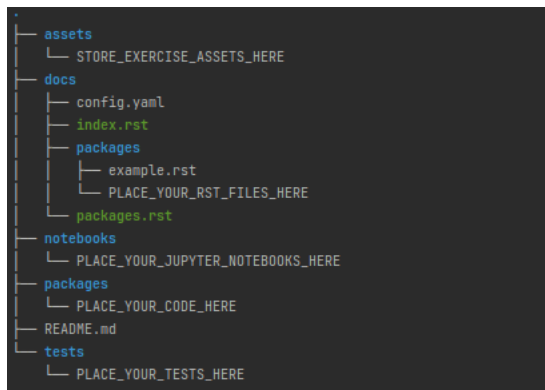


Fig. 8 Directory structure of the LX template generated by `dts lx create`.

The back-end (the recipe)

The *back-end* is stored inside the directory `recipe` (`how-to-robot/recipe` in [Fig. 7](#)). [Fig. 9](#) shows an example of what a back-end directory looks like.

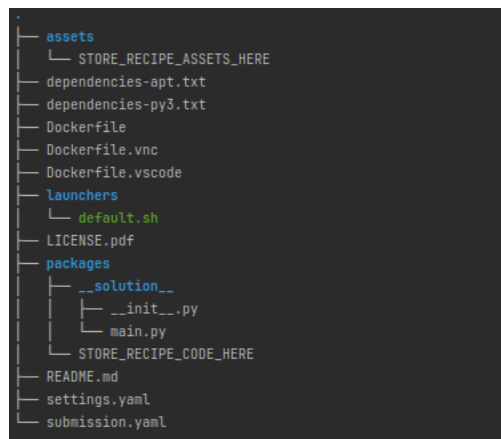


Fig. 9 Directory structure of the LX recipe template generated by `dtls lx create`.

The solution

When a new LX is created, the *learner's workspace* and the *solution* located in the `solution` directory are identical. The instructor is then responsible for adding a fully developed and tested solution, with example agents and activity solutions.

💡 Tip

While developing a new LX, it is good practice to start working on the solution first. Once the solution is in place, parts of the solution with relative pedagogical value can be stripped out and replaced with placeholders effectively producing a boilerplate code that can populate the learner's workspace. This procedure guarantees that the resulting boilerplate is (by construction) enough for the learner to achieve a valid solution.

Step 2: Create a New LX

New learning experiences are developed within the [Development Workspace](#). This structure provides a working directory containing all three of the required directories

- `lx`, `recipe`, and `solution` - for your LX in one place.

What do I need to know ahead of time?

Some information will be required to initialize your LX:

Information	Description
Title	This is a human readable name that will also be converted into a safe file format to name your LX directories. For example, <code>How to Robot!</code> is used as the title and then converted to <code>how-to-robot</code> in file and directory names.
Description	This should contain a learning goal for the LX and short summary of what students will accomplish.
Dependencies	The <code>dtls lx create</code> interface allows you to list initial <code>apt</code> and <code>python3</code> dependencies so that you can ensure your base build works prior to development.

Table 1 Development LX requirements

Note

You can always go back and change these values within the appropriate project files later, but beginning your work with a title, learning goal, and list of general dependencies will help focus your activity development.

Creating an LX development project

Create your LX development project by running

```
dts lx create
```

Wait for the form UI to appear or click on the URL provided in the terminal to access the following form:

Create New Learning Experience
Populate the fields below to create a new Learning Experience

Title *
This will also be used to generate a file safe workspace name e.g. How to Robot! -> how-to-robot.

Description *
What is this Learning Experience about?

Base *
Choose the foundation for your Learning Experience.
☐ Duckietown Baseline (challenge-aido_if-baseline-duckietown)
☐ Machine Learning Baseline (challenge-aido_if-baseline-duckietown-ml)

Supported Robot Versions
Which robots will your Learning Experience support?
☐ DB18
☐ DB19
☐ DB20
☐ DB21M
☐ DB21J
☐ DBR4

Maintainer Name *
Your name and email will be listed as the maintainer in the project files.

Maintainer Email *

APT Dependencies
List the packages you would normally install with 'apt install'.

Python3 Dependencies
List the packages you would normally install with 'pip3 install'.

* Required field

Generate LX

DUCKIETOWN

Fig. 10 The LX create tool configuration interface.

Once you have filled in each of the initial configuration details, click on **Generate LX**. You should receive the following message.

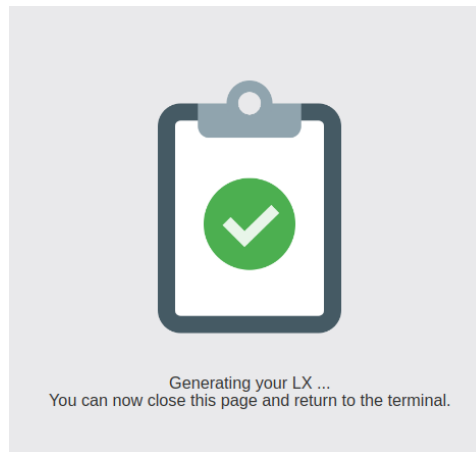


Fig. 11 The message indicating your new LX development project was successfully generated.

Troubleshooting

SYMPTOM	The form does not close or give the success message when I select Create . What now?
RESOLUTION	If you have any issues with the form submission, check your command line for any error messages associated with the data you provided. Then resubmit the form.

Return to the command line to ensure your project was created successfully.

Verify your template

When the create process is finished, verify that your LX development project structure matches the following

```
.
├── lx
│   ├── assets
│   ├── docs
│   ├── notebooks
│   ├── packages
│   ├── README.md
│   └── tests
├── recipe
│   ├── assets
│   ├── dependencies-apt.txt
│   ├── dependencies-py3.txt
│   ├── Dockerfile
│   ├── Dockerfile.vnc
│   ├── Dockerfile.vscode
│   ├── launchers
│   ├── LICENSE.pdf
│   ├── packages
│   ├── README.md
│   ├── settings.yaml
│   └── submission.yaml
└── solution
    ├── assets
    ├── docs
    ├── notebooks
    ├── packages
    ├── README.md
    └── tests
```

Fig. 12 Directory structure of the LX development project generated by `dts lx`

`create`.

The first step after creating a new development project should always be to run `dts code build --recipe ../recipe` in the `<your-lx-workspace>/lx` directory to ensure that the template was initialized properly.

Attention

When working in the development workspace, always append the `--recipe` flag to any `dts code` commands in order to specify your local development version of the recipe.

Jumping into to development ...

The next section of this development manual focuses on providing context and tutorials for each type of learning experience activity that you can add to your new LX template. An LX may have one, several, or all activity types included, so read through the educational purposes associated with each one to determine what is best for your desired learning outcome.

Step 3: Develop Your Activities

Notebook Activities

Notebooks offer a clean interface to provide lesson content, visualizations, and instructions for proceeding through an LX.

At the core, a Duckietown Learning Experience notebook has all the possibility of a [classic Jupyter notebook](#) but with every library and functionality from the Duckietown ecosystem added in.

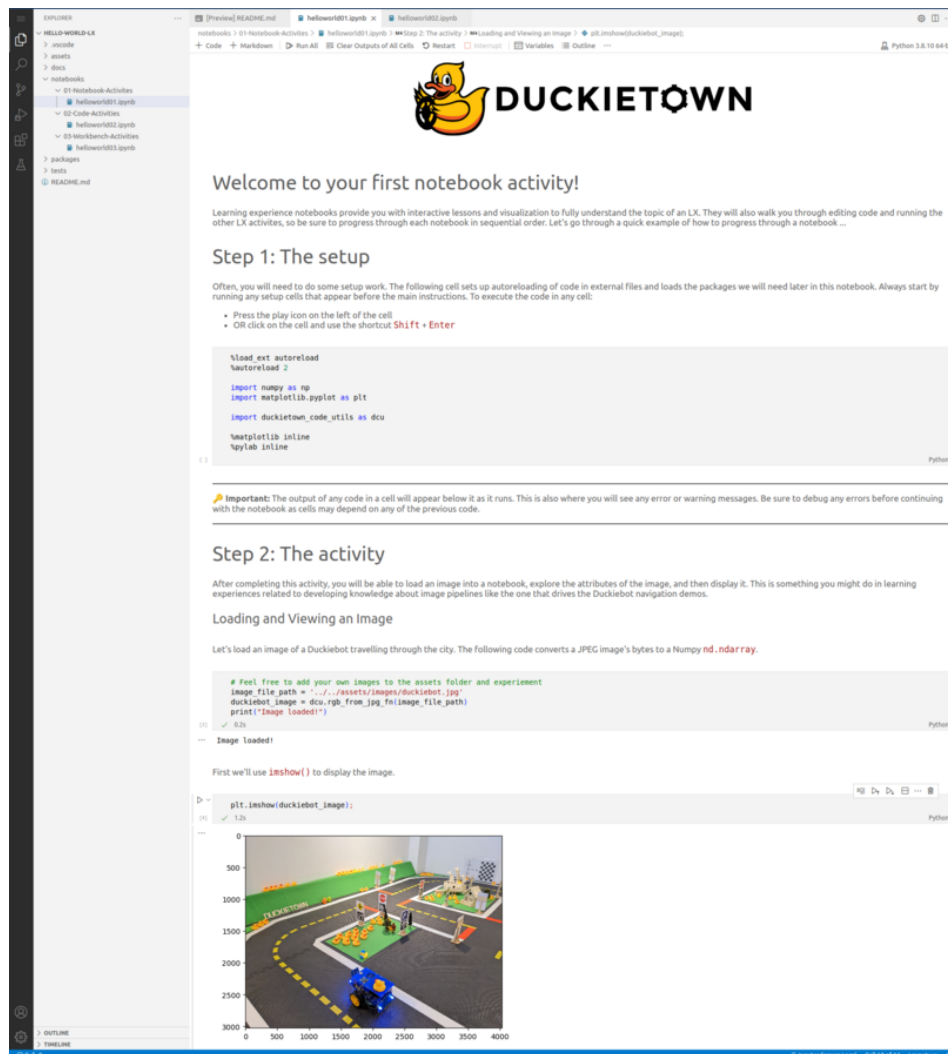


Fig. 13 Each challenge and linked learner submissions can be found on the Duckietown Challenges Server.

Take a moment to explore the notebooks in the [Demo Learning Experience](#) to see a few of the integration options.

Edit zones

The following table contains a list of the files and directories that you may need to update to implement this type of LX activity. If you would like a full walkthrough showing how to implement notebooks, skip to the next section.

Feature	File Location	Purpose
Jupyter Notebook	<code>lx/notebooks/01_first_notebook.ipynb</code>	Notebook files contain the knowledge portion of an LX, walking students through activities, visualizations, and development.
Learner Solution Code	<code>lx/packages/solution_module.py</code>	Python files for learner implementations should remain in the <code>packages</code> directory, with learners filling in TODOs as instructed by the notebooks.
Notebook Dependencies	<code>recipe/dependencies-apt.txt</code> , <code>recipe/dependencies-py3.txt</code>	Required libraries are built into the VSCode editor environment by including them in the <code>recipe</code> dependency files.

Table 2 Edit zones

Development Guidelines

Notebooks vs packages:

Notebooks vs. Packages: As a general rule, example code and visualizations should be developed by students directly in the notebooks.

Code for agents and other packages should be placed in the `packages/solutions` directory for students to edit in the respective Python files, then imported into the notebooks and unit tests for visualization and testing of their results.

A note on solutions:

Notebooks should generally walk through the knowledge required to implement some behavior on the Duckiebots, providing solutions to every notebook activity along way until the culminating activity in the final notebook.

Students should then be editing package files to implement their work as directed by the notebook instructions, and the solution should be hidden in the separate solutions repository to enable evaluation.

Notebook length:

The format of a Duckietown Learning Experience paired with the `dtsc` workflow enables students to iteratively edit and run solutions as they walk through notebooks and develop their understanding of a topic.

Within this framework, shorter notebooks with concrete goals that build learner skills up to a final agent challenge are more effective than one long, content-heavy notebook.

Tutorial: Adding Content to Notebooks

Step 0: Confirm your LX workspace setup

The first step after creating a new development project should always be to run `dtc code build --recipe ../recipe` in the `<your-lx-workspace>/lx` directory to ensure that the template was initialized properly.

Step 1: Determine the learning and development goals

Defining the learning goal will inform the content and visualization that you add to the notebook.

Defining a development goal will allow you to set learners up with a clear purpose and provide tests that ensure they successfully completed any coding exercises included in the notebook or a linked solution script.

Step 2: Determine dependencies

If possible, determine the required Duckietown and external libraries and add them to a setup cell at the beginning of the notebook. This will confirm a lack of later import errors.

You can install external libraries by adding to the `recipe/dependencies-apt.txt` and `recipe/dependencies-py3.txt` files. Any dependencies added here will be available in the VSCode editor environment.

Step 3: Content and code recommendations

Content is added to a Duckietown Learning Experience notebooks in the same Markdown format as any standard Jupyter notebook. For more information, see the [Jupyter notebook docs](#).

As noted above, students should be editing package files to implement their work as directed by the notebook instructions, and the solution should be hidden in the separate solutions repository to enable evaluation.

This means that there are three places you can choose to have learners write solution code.

1. Directly in the notebook cells. This should be used for content examples and practice.
2. In a solution python script in the `lx/packages/solution` directory that is imported into the notebook for visualization. All functions should be predefined with clear **TODOs** marked for learners to complete as directed by the notebook.
3. In a solution python script in the `lx/packages/solution` directory that is then imported into the Duckiebot agent code located in the `recipe` for simulation and workbench activities (more on this in the following sections).

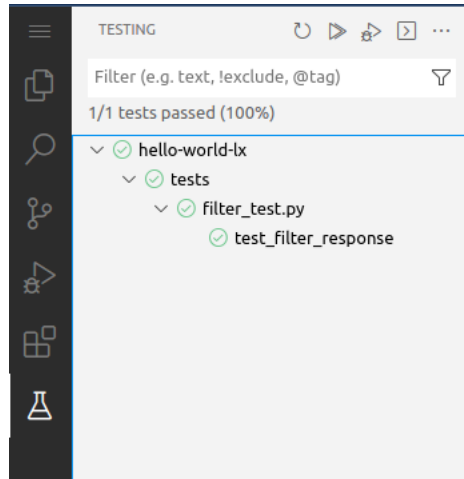
As a general rule, notebooks **do not** have access to the packages in the `recipe/solutions` directory where the base code is placed for Duckiebot agents. This is to prevent learners from editing agent code to a confusing or unusable state.

In summary: Place visualization and content practice code in the notebook. Place learner solution code in linked python scripts housed in the solutions directory.

Hint

Strengthen learner test-driven development (TDD) habits by using the Testing interface in the **VSCode** editor to provide unit tests for each function to be completed in an LX.

This will confirm that their solution performs as expected before any attempts to run it in simulation or on your Duckiebot. Note that the beaker symbol to open the Testing interface may not appear in the sidebar until after one of the Python files in the **packages** directory has been opened.



Step : Clean up and publish

Clear all cells of output to avoid publishing solutions. Then follow the workflow in [Publishing your LX](#) to add your notebook and recipe updates to your Learning Experience repositories.

Workbench Tools

Educational purpose

TODO Empty section here

Edit zones

The following table contains a list of the files and directories that you may need to update to implement this type of LX activity. If you would like a full walkthrough showing how to implement VNC experiences, skip to the next section.

Feature	File Location	Purpose
Desktop Icon	fill	fill
Launch Script	fill	fill
Build Requirements	fill	fill
Experience Scripts	fill	fill

Table 3 Edit zones

TODO Complete the table and remove the **fill** placeholders.

Tutorial

TODO Empty section here

Simulated Agents

Educational purpose

TODO Empty section here

Edit zones

The following table contains a list of the files and directories that you may need to update to implement this type of LX activity. If you would like a full walkthrough showing how to implement VNC experiences, skip to the next section.

Feature	File Location	Purpose
Desktop Icon	fill	fill
Launch Script	fill	fill
Build Requirements	fill	fill
Experience Scripts	fill	fill

Table 4 Edit zones

TODO Complete the table and remove the `fill` placeholders.

Tutorial

TODO Empty section here

Duckiebot Agents

Educational purpose

TODO Empty section here

Edit zones

The following table contains a list of the files and directories that you may need to update to implement this type of LX activity. If you would like a full walkthrough showing how to implement VNC experiences, skip to the next section.

Feature	File Location	Purpose
Desktop Icon	fill	fill
Launch Script	fill	fill
Build Requirements	fill	fill
Experience Scripts	fill	fill

Table 5 Edit zones

TODO Complete the table and remove the `fill` placeholders.

Tutorial

TODO Empty section here

Evaluated Challenges

The agents developed in Duckietown Learning Experiences can be evaluated against a set of benchmarks defined as a *Duckietown Challenge*.

Learners may evaluate their agent locally via

```
dts code evaluate
```

or on the *Duckietown Challenges Server* via

```
dts code submit
```

In each of these cases, a report is created against these benchmark metrics along with visualizations of the agent's behavior in simulation. Server submissions are appended to a running leaderboard, allowing learners to compare their solutions with previous work.

You can explore previous challenge definitions and the related student submissions on the [Duckietown Challenges Server](https://duckietown.org/challenges-server/).

Duckietown Challenges [Home](#) [Challenges](#) [Submissions](#) [My submissions](#)


Duckietown challenges server
This is the Duckietown challenges server. For more information, look at duckietown.org.


Challenges

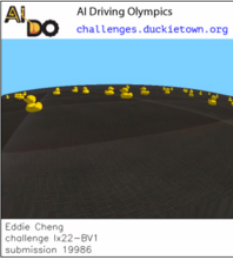
Mooc 2022 challenges Validation challenges Variations on AI-DO challenges AI-DO (full state)

These are the challenges for the 2022 mooc.

Challenge LX - Braitenberg 1

#1 Roberto de Nóbrega 19299
AI Driving Olympics
challenges.duckietown.org

Roberto de Nóbrega
challenge lx22-BV1
submission 19299
[copy link](#)

#2 Sampa Ranta 19283
AI Driving Olympics
challenges.duckietown.org

Sampa Ranta
challenge lx22-BV1
submission 19283
[copy link](#)

#3 Eddie Cheng 19986
AI Driving Olympics
challenges.duckietown.org

Eddie Cheng
challenge lx22-BV1
submission 19986
[copy link](#)

Rank (user)	User	Submission	complete	User label	Distance from starting point.1
1	Roberto de Nóbrega	19299	1/5	exercises_braitenberg	5.3
2	Sampa Ranta	19283	1/5	exercises_braitenberg	5.2
3	Eddie Cheng	19986	1/5	exercises_braitenberg	5.1
4	Marc Maitre	19482	1/5	exercises_braitenberg	5.05
5	James Hofer	19287	1/5	exercises_braitenberg	5.05
6	Daniel Penchev	19645	1/5	exercises_braitenberg	4.95
7	Ron Saad	19481	1/5	exercises_braitenberg	4.65
8	Laido Valdyee	19539	1/5	exercises_braitenberg	4.5
9	Thomas Steinle	19480	1/5	exercises_braitenberg	4.45
10	Panagioti Moraiti	19641	1/5	exercises_braitenberg	4.3

See also [the extended leaderboard](#).

This is a validation challenge: the output is visible to everybody. This challenge is OPEN for submissions.

Fig. 14 Each challenge and linked learner submissions can be found on the Duckietown Challenges Server.

Development Guidelines

Communicating benchmarks:

It is recommended that the evaluation metrics for a challenge are clearly defined in the *Grading* section of the Learning Experience `README.md` file to give learners clear performance goals for their agent.

Encouraging frequent commits:

Students should also be encouraged to bookmark their development at each submission attempt with a git commit referencing the submission number. This allows them to easily track and revert to prior attempts.

Tutorial: Creating and Linking a Duckietown Challenge

Learning Experiences can be associated with a predefined challenge by updating the fields in the file `recipe/submission.yaml`.

ⓘ Attention

The ability to create custom challenges for a Learning Experience has not yet been released. Please check back later for future updates.

The following chapters outline the implementation steps and build requirements for each type of activity that a learning experience may contain.

Use the activity descriptions to determine which type is right for the learning goal you are adding to your LX.

Then jump right into the list of **Edit zones** for free form content development or walk through the **Development Tutorial** for step-by-step examples.

We will be using the `demo-1x` Learning Experience as a running example in this section. You can clone it from the [Duckietown Learning Experiences GitHub repository](#) to follow along.

💡 Tip

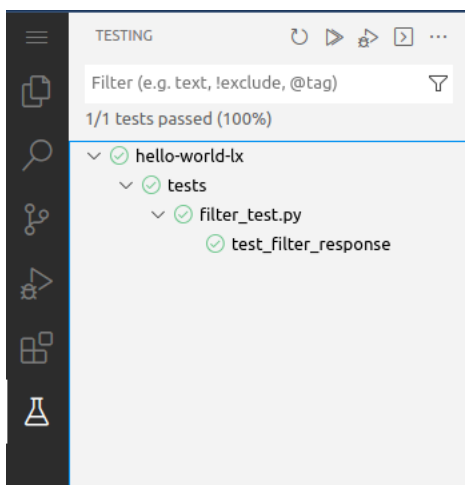
While developing a new LX, it is good practice to start working on the solution first. Once the solution is in place, parts of the solution with relative pedagogical value can be stripped out and replaced with placeholders effectively producing a boilerplate code that can populate the learner's workspace. This procedure guarantees that the resulting boilerplate is (by construction) enough for the learner to achieve a valid solution.

Step 4: Publish Your LX

Testing Your LX

TODO

In progress: We will be developing a testing interface into the tools provided by the `dtls lx` command set soon. In the meantime, be sure to provide unit tests for students in the `tests` directory. They can be easily run in VSCode using the Testing plugin.



Publishing Your LX

The three directories making up an LX can be published to their respective repositories using the `dtls code publish` command.

This provides a streamlined interface for managing repositories and branches so that your LX directories will never be out of sync with each other across the development project.

What do I need to know ahead of time?

Some information will be required to publish your LX:

Information	Description
Repository / Branch for each LX portion	Each of the three LX directories (lx , recipe , solution), should be published to a different repo. Students should have access to the LX and optionally the solution, but the recipe should remain private to avoid complicating the dts code learner workflow.
Version	The version description will be used as the commit message when you publish to a set of GitHub repositories.

Table 6 LX publishing requirements

Publishing an LX development project

Publish your LX development project by entering the main project directory (one level above the **lx**, **recipe**, and **solution** directories, **not** within them) and running

```
dts lx publish
```

Wait for the form UI to appear or click on the URL provided in the terminal to access the following form:

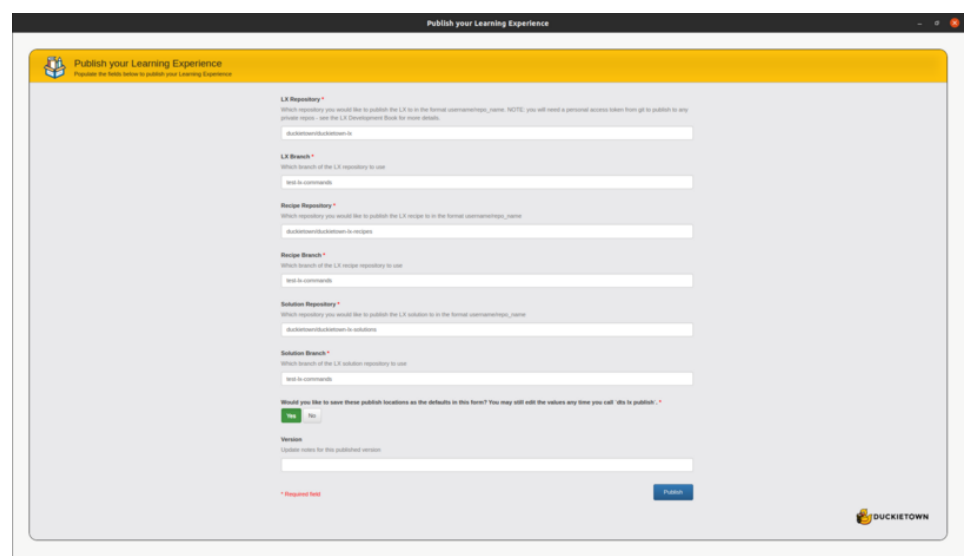


Fig. 15 The LX publish tool configuration interface.

Then fill in the required information. First, the repository and branch for each of the three LX portions:

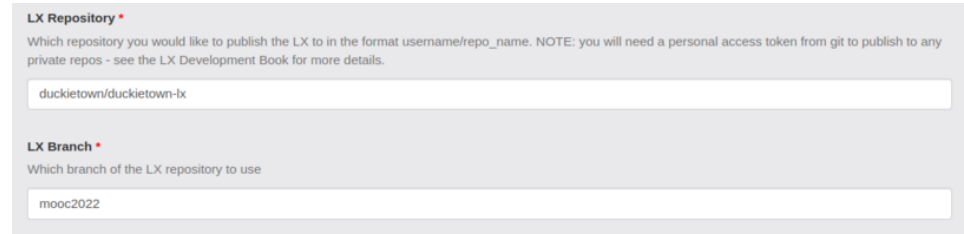
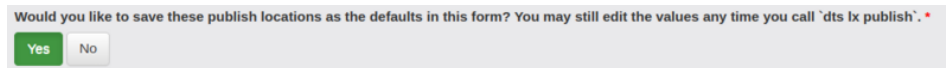


Fig. 16 The repository and branch that each directory will be published to.

! Important

Each of the three LX directories (`lx`, `recipe`, `solution`), should be published to a different repo. Students should have access to the LX and optionally the solution, but the recipe should remain private to avoid complicating the `dtls code` learner workflow.

The information you enter will automatically save, so that you can conveniently publish frequently. You may update these values during any future publish as the form will appear every time.



Would you like to save these publish locations as the defaults in this form? You may still edit the values any time you call 'dtls lx publish'. *

Yes No

Fig. 17 The default values will be saved for convenient iterative publishing.

The version description you provide will be used as the commit message when pushing to the repositories.



Version

Update notes for this published version

* Required field

Publish

Fig. 18 The publish commit message.

Select `Publish` and return to the terminal to confirm that your artifacts were pushed successfully.

TODO Goal: Define testing best practices and where to publish LX materials

Digging Deeper: The LX Build Process

Digging Deeper: The `dtls code` Workflow