I'm not a robot

make is a powerful automation platform that allows you to connect various applications and services to automate workflows and tasks with make, you can automate processes just like a developer, all without any coding expertise what can i do using make? the possibilities with make are nearly endless you can improve how your business works by automating many different tasks here are a few examples of common manual tasks you can automate notify the sales team when a customer fills out a form expressing interest in your products and services automatically send a welcome email to new employees one week before they join the company add new prospects to your crm once the marketing team has qualified them as leads generate quarterly pdf reports of won sales opportunities make is also great at handling complex end to end processes that make your business run more smoothly set up complete employee onboarding that automatically creates it accounts, requests laptops, orders access cards, and sends welcome emails handle customer questions by automatically creating tickets and using ai to sort them based on what they're asking about, their location, and which language they're using process invoices automatically after orders are fulfilled and products are shipped while you focus on growing your business, make takes care of repetitive tasks in the background how does it work? to automate tasks or build complex processes, you need to communicate with different applications and systems for instance, extracting information from a google or microsoft spreadsheet requires connecting to their services these services offer what we call apis what's an api? docid\ tm d9kd1ljjgoun1argdk , you'll learn how make handles these connections for you, opening up automation to everyone I built this guide because I could never quite wrap my head around Makefiles. They seemed awash with hidden rules and esoteric symbols, and asking simple questions didnt yield simple answers. To solve this, I sat down for several weekends and read everything I could about Makefiles. I've condensed the most critical knowledge into this guide. Each topic has a brief description and a self contained example that you can run yourself.If you mostly understand Make, consider checking out the Makefile Cookbook, which has a template for medium sized projects with ample comments about what each part of the Makefile is doing.Good luck, and I hope you are able to slay the confusing world of Makefiles!Getting Started Makefiles are used to help decide which parts of a large program need to be recompiled. In the vast majority of cases, C or C++ files are compiled. Other languages typically have their own tools that serve a similar purpose as Make. Make can also be used beyond compilation too, when you need a series of instructions to run depending on what files have changed. This tutorial will focus on the C/C++ compilation use case.Here's an example dependency graph that you might build with Make. If any file's dependencies changes, then the file will get recompiled: What alternatives are there to Make? Popular C/C++ alternative build systems are SCons, CMake, Bazel, and Ninja. Some code editors like Microsoft Visual Studio have their own built in build tools. For Java, there's Ant, Maven, and Gradle. Other languages like Go, Rust, and TypeScript have their own build tools.Interpreted languages like Python, Ruby, and raw Javascript don't require an analogue to Makefiles. The goal of Makefiles is to compile whatever files need to be compiled, based on what files have changed. But when files in interpreted languages change, nothing needs to get recompiled. When the program runs, the most recent version of the file is used.The versions and types of MakeThere are a variety of implementations of Make, but most of this guide will work on whatever version you're using. However, it's specifically written for GNU Make, which is the standard implementation on Linux and MacOS. All the examples work for Make versions 3 and 4, which are nearly equivalent other than some esoteric differences.Running the ExamplesTo run these examples, you'll need a terminal and "make" installed. For each example, put the contents in a file called Makefile, and in that directory run the command make. Let's start with the simplest of Makefiles:hello: echo "Hello, World"Note: Makefiles must be indented using TABs and not spaces or make will fail.Here is the output of running the above example: makeecho "Hello, World"Hello, WorldThat's it! If you're a bit confused, here's a video that goes through these steps, along with describing the basic structure of Makefiles. A Makefile consists of a set of rules. A rule generally looks like this:targets: prerequisitescommandcommandcommandThe targets are file names, separated by spaces. Typically, there is only one per rule.The commands are a series of steps typically used to make the target(s). These need to start with a tab character, not spaces.The prerequisites are also file names, separated by spaces. These files need to exist before the commands for the target are run. These are also called dependenciesThe essence of MakeLet's start with a hello world example:hello: echo "Hello, World" echo "This line will print if the file hello does not exist."There's already a lot to take in here. Let's break it down:We have one target called helloThis target has two commandsThis target has no prerequisitesWe'll then run make hello. As long as the hello file does not exist, the commands will run. If hello does exist, no commands will run.It's important to realize that I'm talking about hello as both a target and a file. That's because the two are directly tied together. Typically, when a target is run (aka when the commands of a target are run), the commands will create a file with the same name as the target. In this case, the hello target does not create the hello file.Let's create a more typical Makefile - one that compiles a single C file. But before we do, make a file called blah.c that has the following contents:int main() { return 0; }Then create the Makefile (called Makefile, as always):blah:cc blah.c -o blahThis time, try simply running make. Since there's no target supplied as an argument to the make command, the first target is run. In this case, there's only one target (blah). The first time you run this, blah will be created. The second time, you'll see make: 'blah' is up to date. That's because the blah file already exists. But there's a problem: if we modify blah.c and then run make, nothing gets recompiled.We solve this by adding a prerequisite:blah: blah.ccc blah.c -o blahWhen we run make again, the following set of steps happens:The first target is selected, because the first target is the default targetThis has a prerequisite of blah.cMake decides if it should run the blah target. It will only run if blah doesn't exist, or blah.c is newer than blahThis last step is critical, and is the essence of make. What it's attempting to do is decide if the prerequisites of blah have changed since blah was last compiled. That is, if blah.c is modified, running make should recompile the file. And conversely, if blah.c has not changed, then it should not be recompiled.To make this happen, it uses the filesystem timestamps as a proxy to determine if something has changed. This is a reasonable heuristic, because file timestamps typically will only change if the files aremodified. But it's important to realize that this isn't always the case. You could, for example, modify a file, and then change the modified timestamp of that file to something old. If you did, Make would incorrectly guess that the file hadn't changed and thus could be ignored.Whew, what a mouthful. Make sure that you understand this. It's the crux of Makefiles, and might take you a few minutes to properly understand. Play around with the above examples or watch the video above if things are still confusing.More quick examplesThe following Makefile ultimately runs all three targets. When you run make in the terminal, it will build a program called blah in a series of steps:Make selects the target blah, because the first target is the default targetblah requires blah.o, so make searches for the blah.o targetblah.o requires blah.c, so make searches for the blah.c targetblah.c has no dependencies, so the echo command is runThe cc -c command is then run, because all of the blah.o dependencies are finishedThe top cc command is run, because all the blah dependencies are finishedThat's it: blah is a compiled c programblah: blah.o cc blah.o -o blah blah.o: blah.c cc -c blah.c -o blah.o blah.c: echo "int main() { return 0; }" > blah.c If you delete blah.c, all three targets will be rerun. If you edit it (and thus change the timestamp to newer than blah.o), the first two targets will run. If you run touch blah.o (and thus change the timestamp to newer than blah), then only the first target will run. If you change nothing, none of the targets will run. Try it out!This next example doesn't do anything new, but is nontheless a good additional example. It will always run both targets, because some_file depends on other_file, which is never created.some_file: other_file echo "This will always run, and runs second" touch some_file other_file: echo "This will always run, and runs first"Make cleanclean is often used as a target that removes the output of other targets, but it is not a special word in Make. You can run make and make clean on this to create and delete some_file.Note that clean is doing two new things here:It's a target that is not first (the default), and not a prerequisite. That means it'll never run unless you explicitly call make cleanIt's not intended to be a filename. If you happen to have a file named clean, this target won't run, which is not what we want. See .PHONY later in this tutorial on how to fix thissome_file: touch some_file clean:rm -f some_fileVariablesVariables can only be strings. You'll typically want to use :=, but = also works. See Variables Pt 2.Here's an example of using variables:files := file1 file2some_file: $(files) echo "Look at this variable: " $(files) touch some_file file1: touch file1file2: touch file2 clean:rm -f file1 file2 some_fileSingle or double quotes have no meaning to Make. They are simply characters that are assigned to the variable. Quotes are useful to shell/bash, though, and you need them in commands like printf. In this example, the two commands behave the same:a := one twob := 'one two' all:printf '$a'printf $bReference variables using either ${} or $()x := dude all: echo $(x)echo ${x}echo $x Targets Making multiple targets and you want all of them to run? Make an all target.Since this is the first rule listed, it will run by default if make is called without specifying a target.all: one two three one: touch onetwo: touch twothree: touch three clean:rm -f one two threeMultiple targets When there are multiple targets for a rule, the commands will be run for each target. $@ is an automatic variable that contains the target name.all: f1.o f2.o f1.o f2.o: echo $@Automatic Variables and Wildcards Both * and % are called wildcards in Make, but they mean entirely different things. * searches your filesystem for matching filenames. I suggest that you always wrap it in the wildcard function, because otherwise you may fall into a common pitfall described below.print: $(wildcard *.c) ls -la $?* may be used in the target, prerequisites, or in the wildcard function.Danger: * may not be directly used in a variable definitionsDanger: When * matches no files, it is left as it is (unless run in the wildcard function)thing_wrong := *.o thing_right := $(wildcard *.o) all: one two three four one: $(thing_wrong) two: *.o three: $(thing_right) four: $(wildcard *.o)% Wildcard% is really useful, but is somewhat confusing because of the variety of situations it can be used in.When used in "matching" mode, it matches one or more characters in a string. This match is called the stem.When used in "replacing" mode, it takes the stem that was matched and replaces that in a string.% is most often used in rule definitions and in some specific functions.See these sections on examples of it being used:Static Pattern RulesPattern RulesString SubstitutionThe vpath DirectiveAutomatic Variables There are many automatic variables, but often only a few show up:hey: one two echo $@ echo $? echo $^ echo $< touch hey one: touch one two: touch two clean:rm -f hey one twoFancy Rules Make loves c compilation. And every time it expresses its love, things get confusing. Perhaps the most confusing part of Make is the magic/automatic rules that are made. Make calls these "implicit" rules. I don't personally agree with this design decision, and I don't recommend using them, but they're often used and are thus useful to know. Here's a list of implicit rules:Compiling a C program: n.o is made automatically from n.c with a command of the form $(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@Compiling a C++ program: n.o is made automatically from n.cc or n.cpp with a command of the form $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@Linking a single object file: n is made automatically from n.o by running the command $(CC) $(LDFLAGS) $^ $(LOADLIBES) $(LDLIBS) -o $@The important variables used by implicit rules are:CC: Program for compiling C programs; default ccCXX: Program for compiling C++ programs; default g++CFLAGS: Extra flags to give to the C compilerCXXFLAGS: Extra flags to give to the C++ compilerCPPFLAGS: Extra flags to give to the C preprocessorLDFLAGS: Extra flags to give to compilers when they are supposed to invoke the linkerLet's see how we can now build a C program without ever explicitly telling Make how to do the compilation:CC = gcc CFLAGS = -g blah: blah.o blah.c: echo "int main() { return 0; }" > blah.c clean:rm -f blah*Static Pattern Rules Static pattern rules are another way to write less in a Makefile. Here's their syntax:targets...: target-pattern: prereq-patterns ... commandsThe essence is that the given target is matched by the target-pattern (via a % wildcard). Whatever was matched is called the stem. The stem is then substituted into the prereq-pattern, to generate the target's prereqs.A typical use case is to compile .c files into .o files. Here's the manual way:objects = foo.o bar.o all.oall: $(objects) $(CC) $^ -o all foo.o: foo.c $(CC) -c foo.c -o foo.o bar.o: bar.c $(CC) -c bar.c -o bar.o all.o: all.c $(CC) -c all.c -o all.o all.c: echo "int main() { return 0; }" > all.c %.c: touch $@ clean:rm -f *.c *.o allHere's the more efficient way, using a static pattern rule:objects = foo.o bar.o all.oall: $(objects) $(CC) $^ -o all $(objects): %.o: %.c $(CC) -c $^ -o $@ all.c: echo "int main() { return 0; }" > all.c %.c: touch $@ clean:rm -f *.c *.o allStatic Pattern Rules and Filter While I introduce the filter function later on, it's common to use in static pattern rules, so I'll mention that here. The filter function can be used in Static pattern rules to match the correct files. In this example, I made up the .raw and .result extensions.obj_files = foo.result bar.o lose.osrc_files = foo.raw bar.c lose.c all: $(obj_files) $(filter %.o,$(obj_files)): %.o: %.c echo "target: $@ prereq: $

- http://www.jobsincrete.gr/images/_user_na/file/28950458407.pdf
- https://wspaperbag.com/userfiles/file/44209163656.pdf
- jexileki
- sudehi
- katigo
- https://iwistw.com/upload/files/0a5aae82-8b43-4d6e-b154-78addc3131d7.pdf
- magic math tricks app
- onan generator starting procedure