Web Development and API Design

Lesson 02: Bash, Build Tools and Testing

Dr. Andrea Arcuri

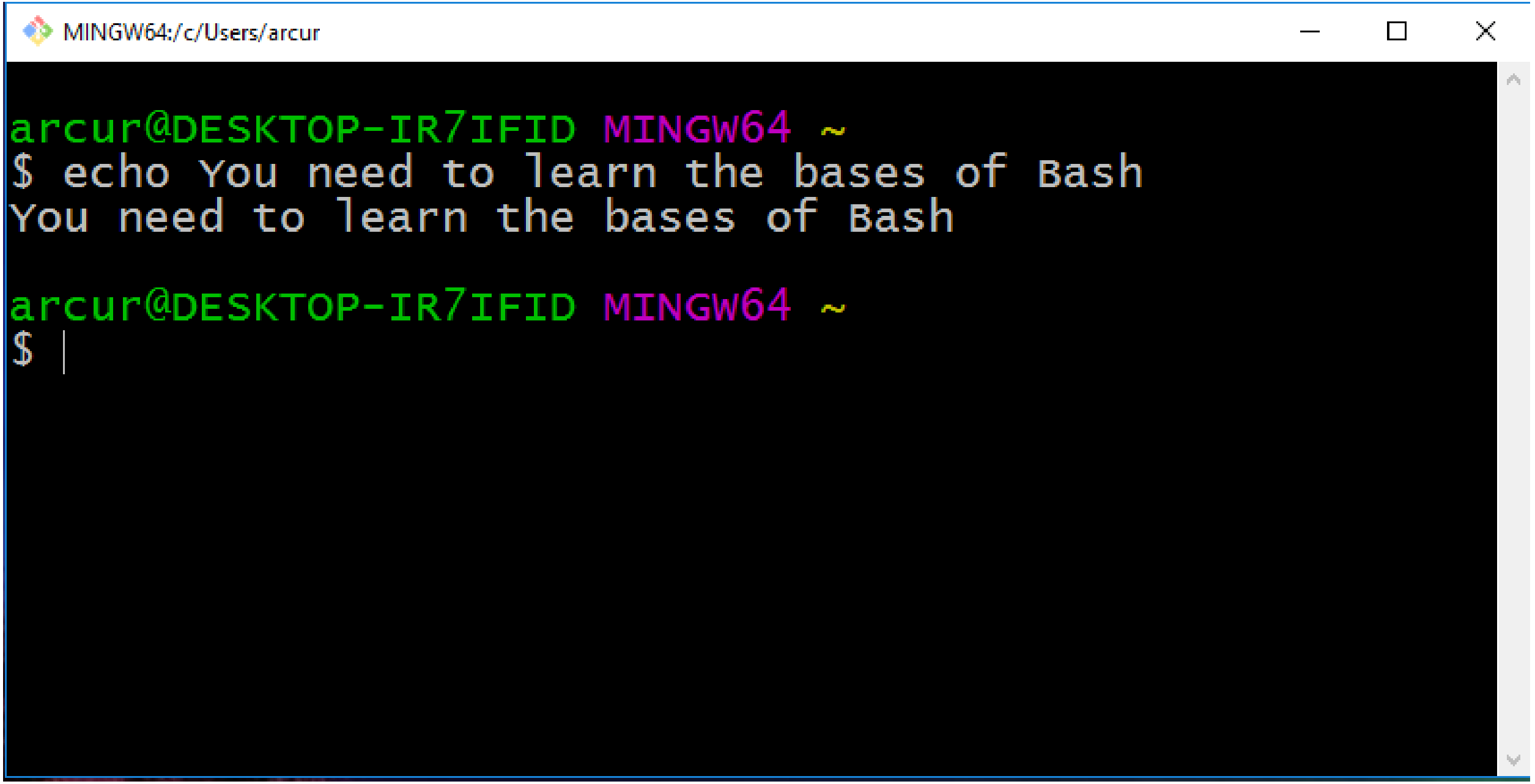
Goals

- Review/intro to Bash Terminal
- Build tools: **YARN**, **WebPack** and **Babel**
- How to write test cases

Bash

Bash

- Bash is a Linux/Mac/Unix shell and command language
- There are also other kinds of shells
  - eg, PowerShell in Windows
- A *shell* is also called: *terminal*, *console*, *command-line*, etc.
- Enable to *type* commands (eg programs), and execute them

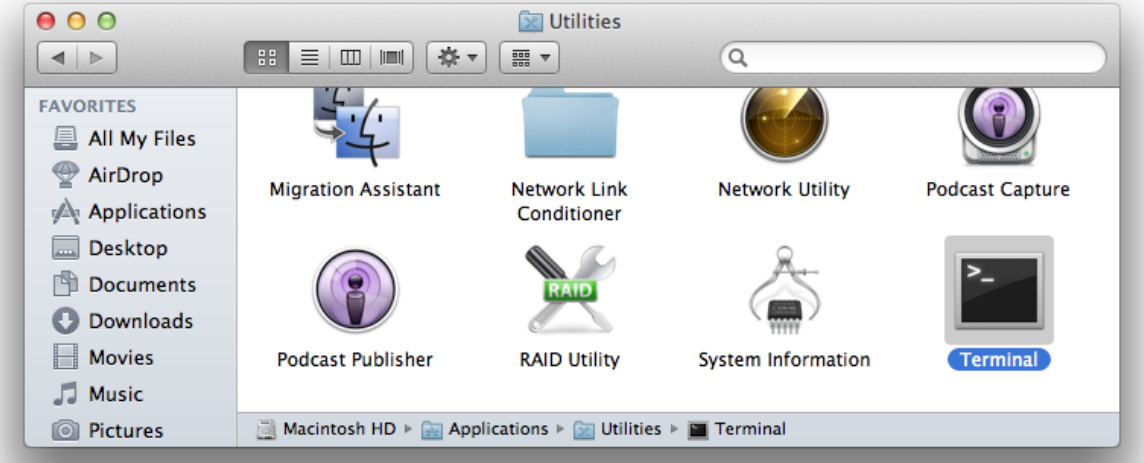

MINGW64:/c/Users/arcur
arcur@DESKTOP-IR7IFID MINGW64 ~
$ echo You need to learn the bases of Bash
You need to learn the bases of Bash
arcur@DESKTOP-IR7IFID MINGW64 ~
$

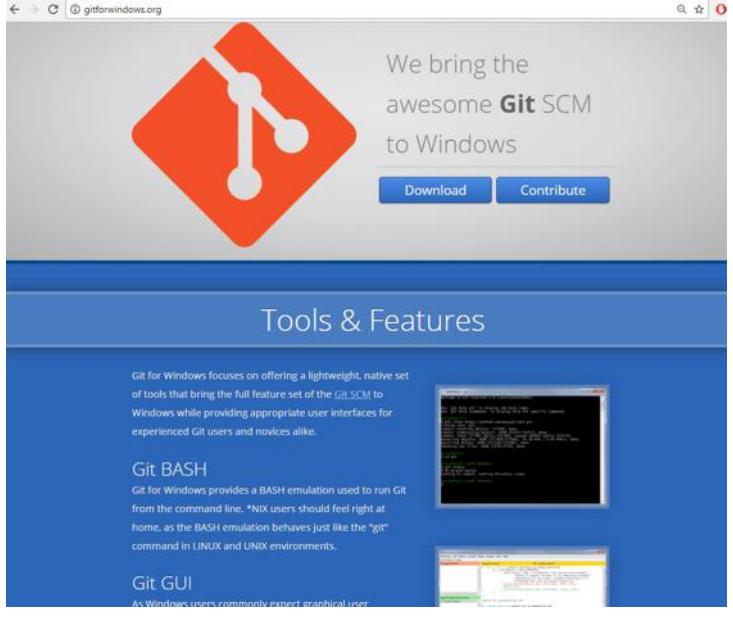
Why?

- Critical skill when you are a programmer
- Help automating several tasks
- When dealing with web/enterprise systems, many servers will NOT have a GUI…
  - … you will access them remotely via SSH using a terminal
  - … this also applies for embedded and IoT devices
- Helpful when commands with specific parameters (eg Git)
- You need to be able to do basic commands
- We will need *Bash* commands in *Docker*

Installing Bash

- If you are using Linux/Mac, it is already installed
  - Mac: Utilities -> Terminal
- If using Windows, strongly recommended to install GitBash
  - which is part of “Git for Windows” at http://gitforwindows.org/





Basic Commands

- “.” the current directory
- “..” the parent directory
- “~” home directory
- “pwd” print working directory
- “cd” change directory
- “mkdir” make directory
- “ls” list directory content
- “cp” copy file
- “mv” move file
- “rm” remove (“-r” for recursive on directories)
- “man” manual for a specific command

Cont.

- “echo” print input text
- “cat” print content of file
- “less” scrollable print of file
- “>” redirect to
- “>>” append to
- “|” pipe commands
- “which” location of program
- “$” resolve variables
- “wc” word count
- “find” files
- “grep” extract based on regular expression
- “touch” modify access time of file, and create it if non-existent

MINGW64:/e/WORK/teaching/bash_examples

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ pwd
/e/WORK/teaching/bash_examples

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ ls

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ echo "ciao" > foo.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ ls
foo.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ cat foo.txt
ciao
```

MINGW64:/e/WORK/teaching/bash_examples/foo

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ mkdir foo

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ ls
foo/  foo.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples
$ cd foo

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ pwd
/e/WORK/teaching/bash_examples/foo

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ ls ..
foo/  foo.txt
```

MINGW64:/e/WORK/teaching/bash_examples/foo

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ cp ../foo.txt ./bar.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ cat bar.txt
ciao

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ ls
bar.txt

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ mv ../foo.txt  .

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ ls
bar.txt  foo.txt
```

MINGW64:/e/WORK/teaching/bash_examples/foo

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ echo $PATH
/c/Users/arcur/bin:/mingw64/bin:/usr/local/bin:/usr/bin:/bin:/mingw64/bin:
/usr/bin:/c/Users/arcur/bin:/c/Program Files/Docker/Docker/Resources/bin:/
c/Users/arcur/bin:/c/Program Files/Java/jdk1.8.0_112/bin:/c/Users/arcur/ap
ache-maven-3.3.9-bin/apache-maven-3.3.9/bin:/c/ProgramData/Oracle/Java/jav
apath:/c/WINDOWS/system32:/c/WINDOWS:/c/WINDOWS/System32/Wbem:/c/WINDOWS/S
ystem32/WindowsPowerShell/v1.0:/cmd:/c/Program Files/MiKTeX 2.9/miktex/bin
/x64:/c/HashiCorp/Vagrant/bin:/c/Program Files/nodejs:/c/Program Files (x8
6)/Skype/Phone:/c/Program Files/PostgreSQL/9.6/bin:/c/Program Files/Micros
oft SQL Server/130/Tools/Binn:/c/Program Files/dotnet:/c/Program Files (x8
6)/GtkSharp/2.12/bin:/c/RailsInstaller/Ruby2.2.0/bin:/c/DevelopmentSuite/c
dk/bin:/c/HashiCorp/Vagrant/bin:/c/DevelopmentSuite/cygwin/bin:/c/Users/ar
cur/AppData/Local/Microsoft/WindowsApps:/c/Users/arcur/AppData/Roaming/npm
:/c/Program Files/Heroku/bin:/c/Users/arcur/AppData/Local/Microsoft/Window
sApps:/usr/bin/vendor_perl:/usr/bin/core_perl

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/teaching/bash_examples/foo
$ which bash
/usr/bin/bash
```

- What if you want to count the number of JavaScript files in your project?

- Or count the total number of lines in all those files?

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/code/teaching/pg6300 (spring-2019)
$ find . -regex '^.*\.jsx?' -not -path */node_modules/* | wc -l
141

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/code/teaching/pg6300 (spring-2019)
$ cat `find . -regex '^.*\.jsx?' -not -path */node_modules/*` | wc -l
12272
```

- "*find*" all the files recursively in the current folder "."
- matching the regular expression for JS/JSX files
  - "**^**" beginning of file name
  - "**.***" any character (.), any number of times (*)
  - "**\.**" escaped any-character to represent the character "."
  - "**\.jsx?**" file ending, where the last "x" is optional (ie "?")
  - "**-not -path */node_modules/***" excludes files of imported dependenices
- "**| wc -l**": pipe file names to line count program
- **cat `x`**: the `` executes the command inside it, and then puts the output on the terminal
  - so, we print all content of all JS/JSX files with **cat**

Useful Tips

- User arrows (up/down) to go through history of commands
- Use “tab” key to complete words, ie commands / file names
- Bash commands can be put in executable scripts
  - Can use “*.sh” as file extension, eg “*foo.sh*”
  - First lines needs to be “*#!<pathToBash>*”, eg “*#!/usr/bin/bash*”
  - Then it can be executed from terminal like any other program

Build Tools

YARN/NPM

- We need to use *external libraries*, typically open-source
  - An important library we are going to use in the rest of the course is for example *React*
- Two main tools in JS: **YARN** and **NPM**
- Both **YARN** and **NPM** access the same dependency repository
- **YARN** tends to be better, with new features coming earlier
- We will use it from terminal
- As **YARN** executes JS code, we need a runtime for it: that is the reason why you also need to install **NodeJS**
  - not going to start a build tool inside a browser…

```
arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/code/teaching/foo/example
$ yarn init -y
yarn init v1.12.3
warning The yes flag has been set. This will automatically answer yes to all questions, which ma
ty implications.
success Saved package.json
Done in 0.05s.

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/code/teaching/foo/example
$ ls
package.json

arcur@DESKTOP-IR7IFID MINGW64 /e/WORK/code/teaching/foo/example
$ yarn install
yarn install v1.12.3
info No lockfile found.
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
[4/4] Building fresh packages...

success Saved lockfile.
Done in 0.12s.
```

- **yarn init –y**
  - create a new *package.json* in current folder, needed when starting new project
- **yarn install**
  - download and install in “*node_modules*” folder all the dependencies declared in *package.json*

package.json

- Main configuration file for the project
- Similar to *pom.xml* in Maven Java projects
- Three main parts you need to care about:
  - **scripts**: executable commands from YARN. Eg, to build or run the app
  - **dependencies**: dependencies used in the project
  - **devDependencies**: dependencies only used during development, but not being part of the final app (eg, we will see *WebPack*)

JSON for Configuration Files

<rant>

JSON as format for configuration files is simply **awful**.

For example, you cannot have comments…

NPM is not better, as uses exactly the same package.json

</rant>

yarn.lock

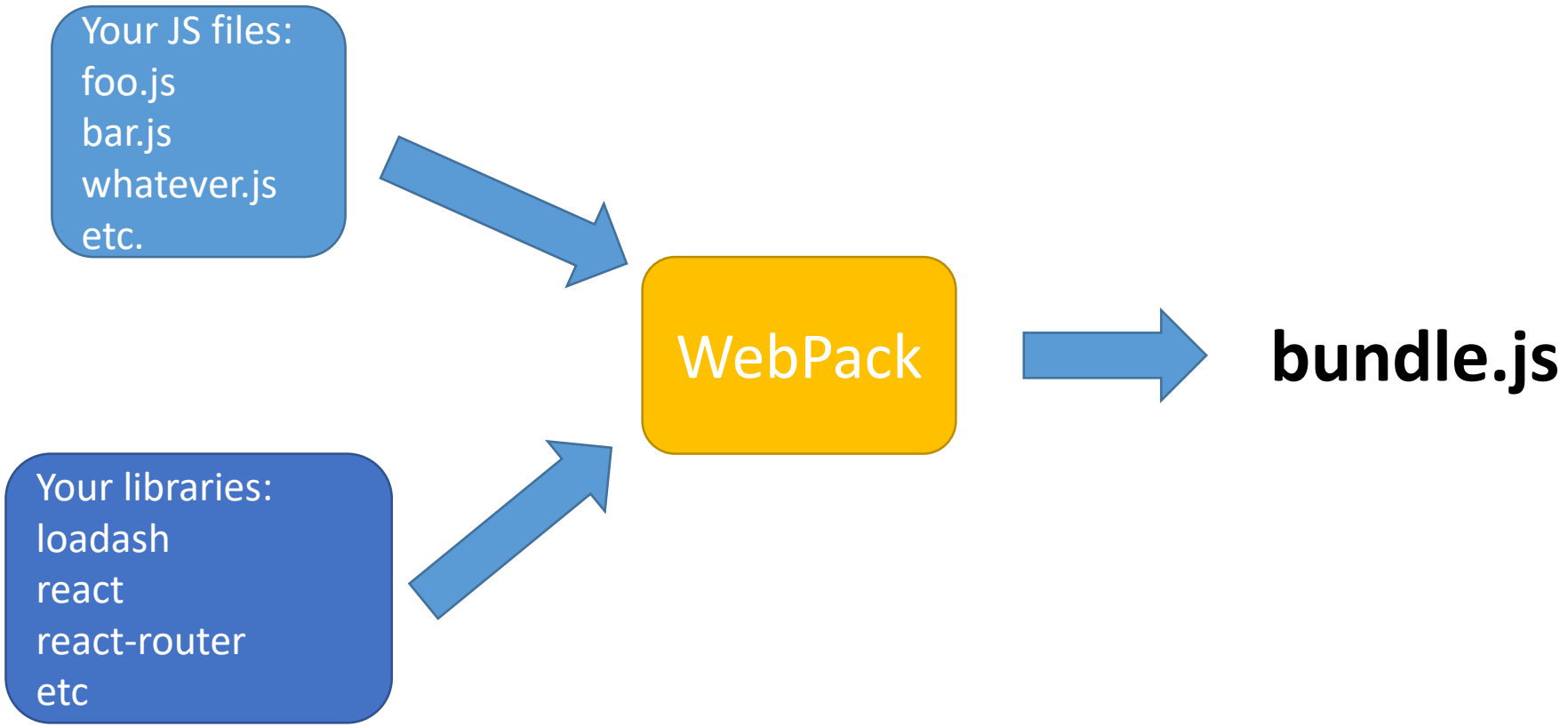
- Once you install the dependencies, you will see that YARN does create a *yarn.lock* file
- Dependencies need to define which version to use, eg 1.0.2
- You can use caret **^** to represent the most recent major version
  - e.g., **^1.0.2** will match **^1.4.1**, but not **2.0.0**
- *yarn.lock* just tells YARN to use the exact same versions of the libraries when such file was created
  - extremely important when working in a team, and new minor updates break backward compatibility or introduce new regression bugs

WebPack

- Downloading dependencies is not enough
- Such dependencies need to be accessed by the HTML pages
- Might be cumbersome to update HTML files for each dependency, for each different version
- Furthermore, we might only need a small set of functions from a specific library
- Solution: *bundling* with **WebPack**

Bundling

In the end, we get a *single* JS file



Configuring *WebPack*

- Needs to be installed and called with YARN from *package.json*
- *webpack-dev-server* is a useful tool that starts a HTTP server with hot reloading
  - ie, if you modify your JS files, it automatically re-bundle them

```
"scripts": {
    "dev": "webpack-dev-server --open --mode development",
    "build": "webpack --mode production"
},
"devDependencies": {
  "webpack": "^4.16.5",
  "webpack-cli": "^3.1.0",
  "webpack-dev-server": "^3.1.5"
}
```

webpack.config.js

- Besides being called from *package.json*, WP also needs its own configurations
  - Eg, name of the file to create, and which directory to save it into
- Configuration done in a JavaScript file

Code Transformation

- Bundling is not enough, might need to do *transformations*
- Support other languages: *TypeScript* and **JSX**
  - which are not natively supported by browsers, which only deal with JS
  - JSX will be essential when dealing with *React*
- Support old browsers:
  - eg, transform code using new JS features (eg, *async/await*) into equivalent, valid old JS
- Minification:
  - eg, remove comments and empty spaces from JS files to decrease their size, needed to make their download faster
- etc.

Babel

- **Babel** is the main tool to make JS transformations
- Need to be installed from *package.json*
- Need its own **.babelrc** configuration file, specifying which transformations to apply
  - *Warning*: the "**.**" in front of a file/folder name makes it "hidden" by default is some OSs, like Mac and Linux, which is not really ideal for a configuration file…

Testing

Testing

- To check the correctness of a program, writing test cases is very important
- For *dynamically typed* language, is even more important
  - as you lose a lot of warnings and checks from a compiler
- Different libraries in JS for testing, but we will use **Jest**
  - which is one of the most popular

Configuring *Jest*

- Need entry in **scripts** to start it
- Need extra configurations to find out where the tests are

```
"scripts": {
   "test": "jest --coverage"
},
"jest": {
  "testRegex": "tests/.*\\.test\\.(js|jsx)$",
  "collectCoverageFrom": [
    "src/**.(js|jsx)"
  ]
}
```

Need *Babel* for *Jest*

- JS code running in browser
- But where are the tests run? We need a JS runtime: *NodeJS*
- But *frontend* code might not be directly executed on *NodeJS*
  - eg, different ways to handle JS *modules*
- Using *Babel* to make the required transformations to be able to run such code on *NodeJS*