



Workflow

LoftSchool
ОТ МЫСЛИТЕЛЯ К СОЗДАТЕЛЮ

Содержание:

- 1 Терминал
 - 1.1 Работа в терминале. Mac OS
 - 1.2 Командная строка в Windows и эмуляторы консоли
 - 1.3 Полезные советы для работы с терминалом
- 2 Node.js и npm
 - 2.1 Установка node.js
 - 2.2 REPL
 - 2.3 Глобальная установка пакетов
 - 2.4 Локальная установка пакетов
 - 2.5 Управление зависимостями. Package.json
- 3 Bower
 - 3.1 Bower.json
 - 3.2 Установка пакетов
 - 3.3 Обновление и удаление библиотек
- 4 Git
 - 4.1 Создаём репозиторий в существующем проекте
 - 4.2 Клонирование репозитория
 - 4.3 Определение состояния файлов
- 5 BrowserSync
- 6 Gulp

1 Терминал

Командная строка, консоль, терминал – в чём разница? В первую очередь давайте разбираться с терминами.

Командная строка (консоль) - это программная оболочка, позволяющая в текстовом виде давать компьютеру различные команды, обеспечивающая прямую связь между пользователем и операционной системой. Команды состоят из букв, цифр, символов, набираются построчно, выполняются после нажатия клавиши Enter.

Она встроена в ядро системы и будет доступна, даже если графический интерфейс не запустится.

Терминал - графическая программа, эмулирующая консоль, оболочка для командной строки, позволяющая не выходя из графического режима выполнять команды.

Терминал по сравнению с консолью имеет дополнительный функционал (различные настройки, вкладки, возможность запускать много окон, управление мышью в некоторых программах, контекстное меню, главное меню, полоса прокрутки).

Шелл (shell) - часть ОС, часто называемая интерпретатором. Шелл – это специальный интерфейс, созданный для обеспечения взаимодействия между пользователем и ядром, т.е. это программное окружение, обеспечивающее необходимые условия для запуска приложений. Можно выделить два типа шеллов: *графический* (например, Windows Explorer, Finder в Mac OS X) и *текстовый* (например, bash, sh, tsh, csh, zsh).

Эти три понятия часто используются, заменяя друг друга. Разница между ними сегодня становится все более и более размытой.

bash - это один из самых известных шеллов. Эту программу часто называют *"Bourne again shell"* («возрождённый» шелл) в честь Steven Bourn, который разработал классический shell. Одна из главных особенностей шеллов - все команды, как правило, являются небольшими программами, которые легко найти на своем диске. Bash – это тоже программа, расположенная в каталоге */bin/bash*. Особенно популярна в среде Linux, где она часто используется в качестве предустановленной командной оболочки. А также *по умолчанию установлена в Mac OS X*.

Также достаточно популярен интерпретатор **zsh**, который в свою очередь является улучшенным bash. Он пока не стал стандартом де-факто, но возможно станет в будущем.



Зачем вообще нужно использовать командную строку?

Когда-то привычного всем графического интерфейса в операционных системах не было и все делалось как-раз таки в командной строке. Дело в том, что некоторые вещи можно *быстрее* выполнить именно в командной строке, а некоторые настройки в принципе *отсутствуют* в графическом интерфейсе пользователя. Так же следует иметь ввиду, что до сих пор существуют утилиты, *не имеющие* графического интерфейса, а иногда он оказывается недоступен, например, из-за сбоя. Здесь и используется командная строка.

ПС: Работа с командной строкой - не такая страшная задача, как вы могли бы подумать. Чтобы использовать командную строку не требуется специальных знаний, это такая же программа, как и все остальные. Только вот создана для того, чтобы выполнять текстовые команды, поэтому отложите свою мышку в сторону и подвиньте поближе клавиатуру.

Подведем итоги: Многие программы и утилиты, с которыми мы будем работать (например, Git и Gulp), имеют графические оболочки (*англ. graphical user interface, GUI*). Однако, надо понимать, что GUI работает также через консоль (это скрыто за красивой картинкой). GUI всегда *посредник* между консолью и вашей программой или утилитой, что подразумевает наличие багов и отсутствие всех возможностей, которые нам предоставляет консоль.

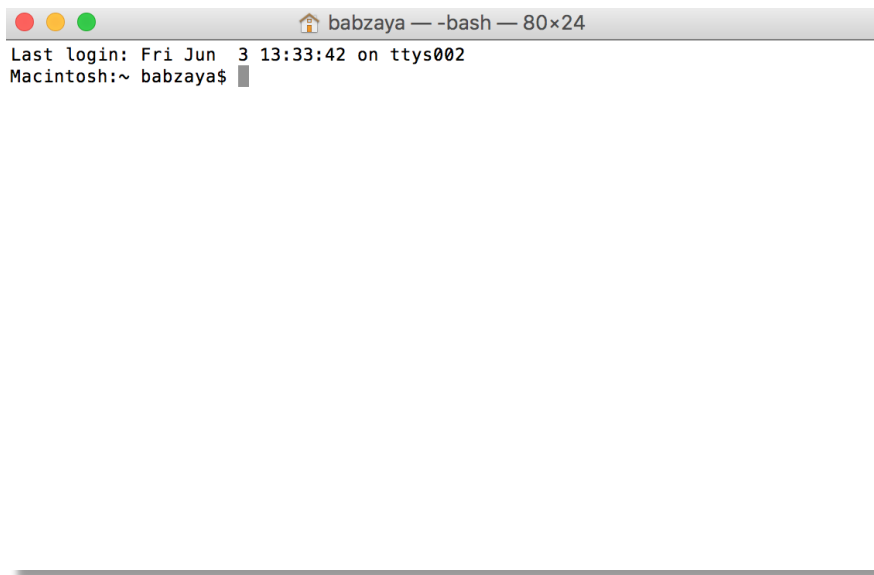
Умение работать в консоли – обязательно в командной работе над крупными проектами!

Работа в терминале. Mac OS

Как Mac OS, так и Linux, относятся к семейству UNIX-подобных ОС, поэтому информация в этой главе будет справедлива и для Linux-систем.

Отличной альтернативой встроенному терминалу является [iTerm2](#) - симпатичная и более продвинутая оболочка для вашего терминала.

Чтобы запустить терминал, нажмите *Ctrl + пробел* и начните вводить “Терминал”.



Командная строка начинается с *названия компьютера* (в примере это Macintosh), затем следует *название текущего каталога* — по умолчанию открывается домашний каталог пользователя, который в UNIX-системах обозначается знаком ~ (тильда).

В любой момент времени работы в терминале вы находитесь в некотором каталоге. При запуске терминала, текущей директорией является домашний каталог пользователя.

Текущий каталог - это то, что написано между символами : и \$.

Далее следует имя пользователя, а за ним знак \$, который называется приглашением – приглашением интерпретатору вводить команды.

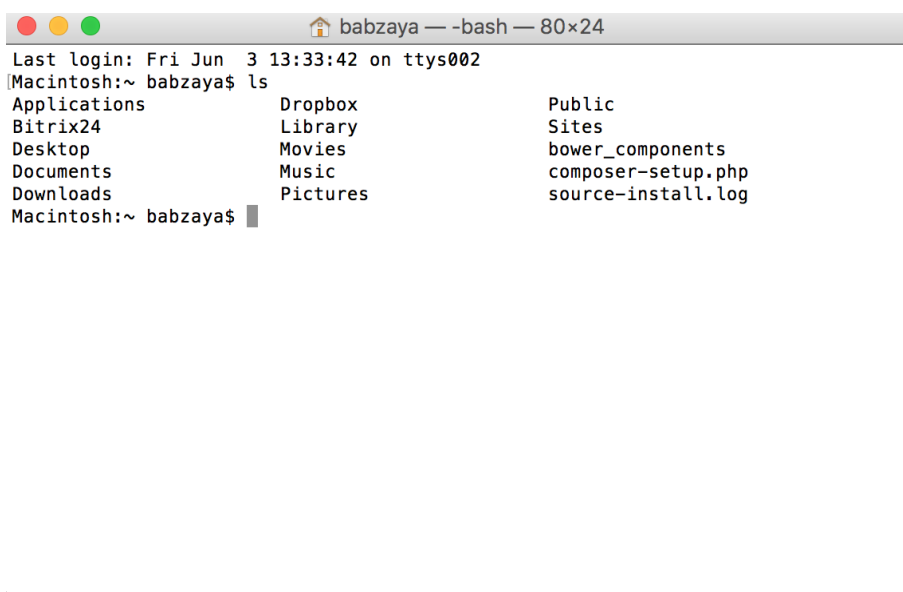
Вид командной строки и приглашения можно настраивать.

Именно после знака \$ и вводятся все команды интерпретатору.

В этом месте находится курсор — мигающий прямоугольник (кстати, его вид тоже можно настраивать).

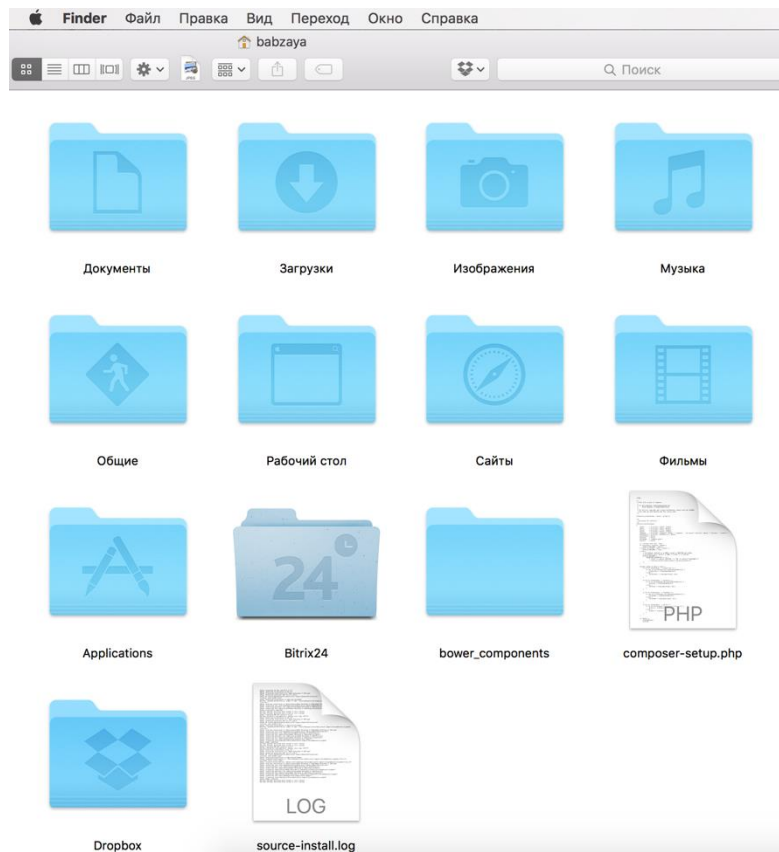
Основные команды:

Для просмотра каталогов и файлов текущей директории используйте команду **ls** (LiSt – список).



```
babzaya — -bash — 80x24
Last login: Fri Jun 3 13:33:42 on ttys002
Macintosh:~ babzaya$ ls
Applications      Dropbox           Public
Bitrix24          Library          Sites
Desktop           Movies           bower_components
Documents         Music            composer-setup.php
Downloads         Pictures         source-install.log
Macintosh:~ babzaya$
```

То же самое в Finder:



Важно знать, что команда **ls** показывает не все файлы, например, *скрытые файлы она не покажет*.

Для того, чтобы вывести весь список файлов (в том числе, скрытых) команду **ls** необходимо вводить с ключом **-a**

Вот так:

ls -a

```

Macintosh:~ babzaya$ ls -a
.                  .node_repl_history
..                 .npm
.CFUserTextEncoding .rmd
.DS_Store          .subversion
.Trash             .v8flags.4.6.85.31.babzaya.json
.adobe             .viminfo
.bash_history      .y3
.bash_profile      Applications
.bash_sessions    Bitrix24
.bxd               Desktop
.cache             Documents
.composer           Downloads
.config            Dropbox
.csshat            Library
.dropbox           Movies
.gitconfig         Music
.gitignore_global  Pictures
.hgignore_global   Public
.local             Sites
.mongorc.js        bower_components
.mysql_history     composer-setup.php
.node-gyp          source-install.log
Macintosh:~ babzaya$

```

Ключи (аргументы) можно комбинировать, например если мы введем **ls -la**, отобразится содержимое каталога вместе со скрытыми файлами (аргумент **a**), а также полный вывод информации (аргумент **l**)

Команда **pwd** покажет абсолютный путь до текущей директории.

cd - изменяет текущую рабочую директорию. Например, если вы находитесь в домашнем каталоге пользователя и введете **cd Downloads**, то перейдете в папку Downloads, а если вы хотите наоборот выйти из каталога на уровень выше, то для этого можно воспользоваться командой **cd ../**

[Список терминальных команд Terminal mac OS X от A до Z](#)

Важно: Есть небольшая особенность в работе с терминалом в UNIX-подобных системах. Если вам нужно изменить что-то в настройках системы, или установить глобальные пакеты (об этом поговорим позже), то перед такими командами необходимо вводить специальную команду **sudo**.

Она позволяет запускать последующие команды от имени администратора.

Запуская команды от имени обычного пользователя, вы получите ошибку. Увидев ее, не пугайтесь, просто допишите **sudo** перед вводом команды.

Не все команды требуют прав администратора, а только те, которые могут нанести вред системе, либо изменить системные настройки, каталоги и т.д.

Это пригодится нам для установки глобальных npm-утилит, а также в некоторых ситуациях - для смены прав каталога, например:

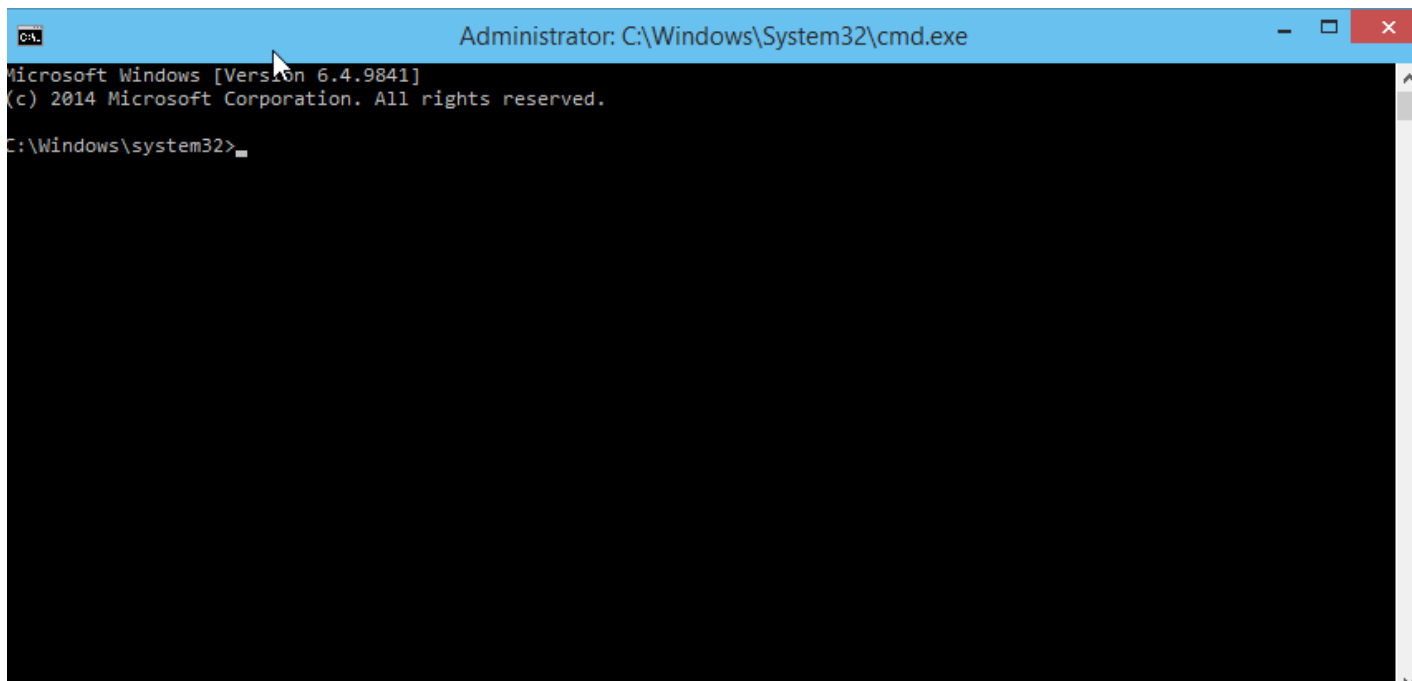
```
sudo npm install -g gulp
```

Командная строка и эмуляторы консоли в Windows

cmd.exe позволяет получить доступ к командной строке Microsoft Windows, это интерпретатор команд, который загружает приложения и направляет поток данных между приложениями, для перевода введенной команды в понятный системе вид.

Перейдем в меню Пуск и запустим его любым из способов:

- Пуск -> Все программы -> Стандартные -> Командная строка.
- Пуск -> Выполнить и вводится имя программы — **cmd.exe**



Мы видим *имя пользователя* и мигающее *нижнее подчеркивание*, показывающее, где будет выводиться набранная вами команда. Как видите, по умолчанию, фон командной строки черный, а текст - белый. Но внешний вид командной строки можно настроить под себя. Для этого кликаем правой кнопкой мыши по адресу cmd вверху, и выбираем в меню «Свойства».

В cmd.exe Windows команды частично совпадают с командами в unix-системах.

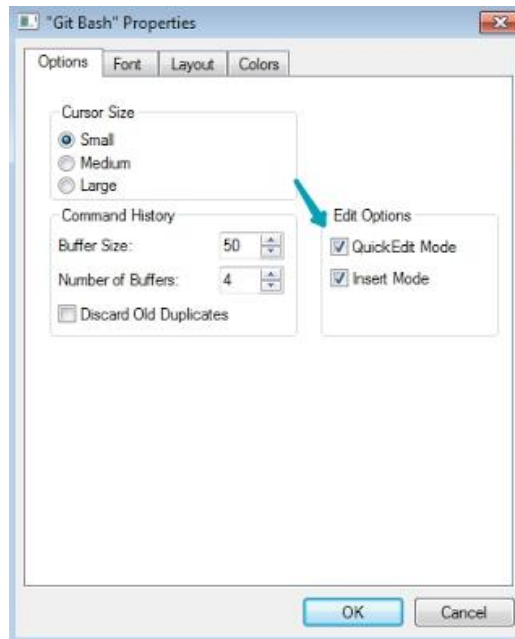
[Cmd команды в полном объеме](#)

Как и в случае со стандартным терминалом в Mac OS X, мы рекомендуем вам поставить более *продвинутой* командную оболочку, например **Git bash** (в котором будут работать все UNIX-команды), **Conemu** или **Cmder**.

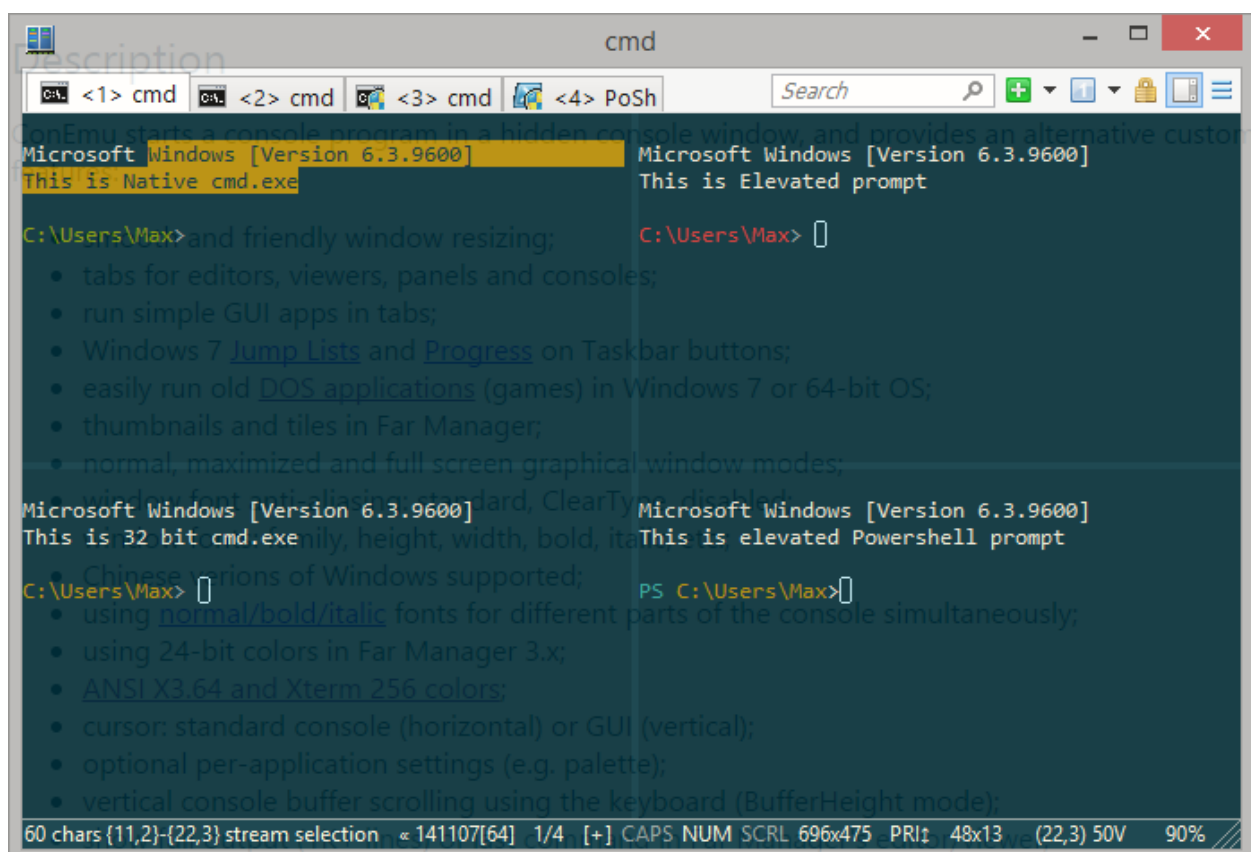
[Git bash](#) - консольная утилита, которая дополнительно устанавливается непосредственно при установке git. Она представляет собой bash-эмуляцию, что позволяет работать со всеми UNIX-командами.

git bash автоматически установится в систему вместе с Git, на рабочем столе появится ярлык, теперь вы можете запускать его и пользоваться всеми UNIX-командами.

Полезная настройка: одна из множества настроек – возможность назначить на клик правой кнопки мыши вставку текста из буфера обмена. Для этого в *Menu* → *Properties* поставьте галочку в пункте QuickEdit Mode:



[ConEmu](#) - еще одна оболочка командной строки, но в отличие от bash здесь есть возможность использовать различные оболочки и утилиты в разных вкладках, например, cmd, powershell, dn, putty, notepad.



Преимущества работы в Сопети:

- **Наглядность.** В табках могут отображаться не только заголовки консолей, но и *дополнительная информация* вроде активного процесса, прогресса архивации, chkdsk, powershell, копирования в Far Manager. Например, не нужно переключаться в таб чтобы узнать закончилась ли компиляция проекта, запущенная в этом табе. В статусной строке можно настроить список отображаемых «колонок» вроде координат видимой области и курсора, PID активного процесса в консоли, статусов CAPS/NUM/SCRL, коэффициента прозрачности и др. Многие колонки кликабельны, например, можно щелкнуть по «колонке» с прозрачностью для быстрого ее изменения.
- **Минималистичность.** Интерфейс самого терминала содержит всего два дополнительных графических элемента — табы и статусная строка. Но и их можно отключить, если вы предпочитаете «чистую» консоль.
- **Запуск любых приложений.** Пользователь может настроить любое количество предопределенных задач (Task) для быстрого запуска приложений в ConEmu или из списка переходов (jump list) панели задач. Задача может запускать один или несколько процессов или шеллов (powershell, SDK, компиляция проектов и т.д.) Можно даже запускать простые GUI приложения вроде PuTTY, TaskManager, GVim.
- **Работа с цветом.** Несколько предопределенных палитр (например, Solarized, PowerShell, xterm, и др.), возможность настройки своих цветов в консоли, поддержка управляющих кодов ANSI X3.64, 24-битный цвет при работе в Far Manager.
- **Интеграция.** Умеет добавлять себя (и выбранные команды-шеллы) в контекстное меню Windows Explorer. Умеет перехватывать создание стандартного терминала Windows.

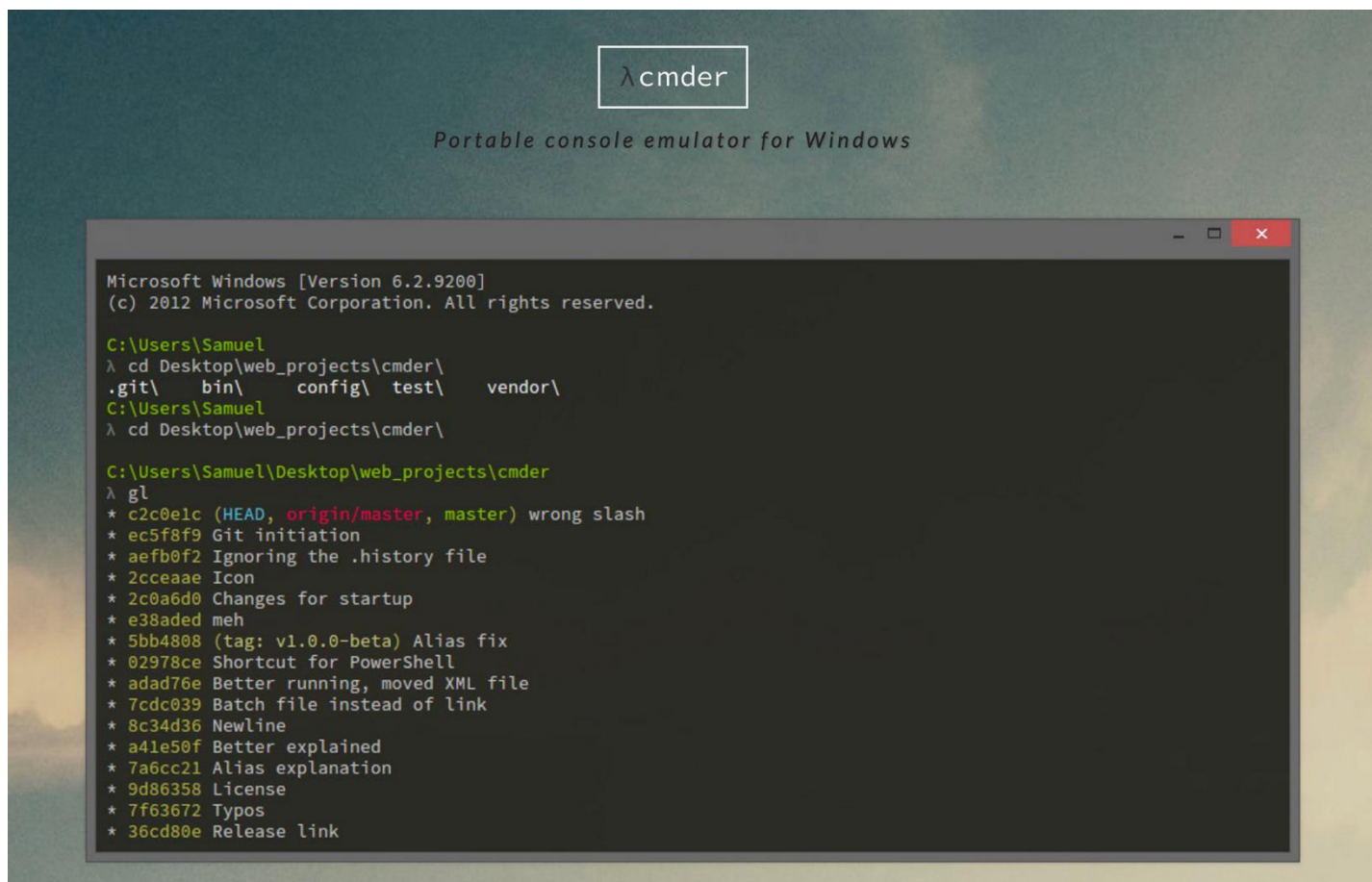
Полезные ссылки:

[Подробная документация по работе с Сопети на русском языке](#)

[Видеурок. Сопети - эмулятор консольного окна: установка, настройка и работа](#)

[Подробная статья на Хабре](#)

[Cmder](#) – еще одна утилита, предназначенная для упрощения работы с интерфейсом командной строки. Основой для создания Cmder послужили ConEmu и Clink – небольшая утилита с поддержкой обработки строк в Bash.



Преимущества работы в Cmder:

- *Портативная версия*, которую можно положить в dropbox.
- Не надо долго настраивать - распаковали и он сразу готов к бою.
- Удобная работа с глобальными переменными Windows.
- Также как и PowerShell, Cmder *поддерживает расширенное использования буфера обмена, запоминание введенных команд, функцию автозавершения команд, а также работу с алиасами*. Для более удобной и быстрой работы приложение предлагает небольшой набор клавиатурных комбинаций, также имеется возможность запуска нескольких сеансов в отдельных вкладках.
- Гибкая настройка и управление. В диалоге *Settings* можно найти настройки на любой вкус и цвет. Практически любому действию можно назначить комбинацию клавиш, а простейший макро-язык позволяет выполнять нестандартные действия.
- Cmder *доступна в трех редакциях* – mini, small и full (с компонентом msysgit). Установки программа не требует. *Запустить приложение можно двойным кликом по файлу Cmder.bat*.

[Горячие клавиши для cmder](#)

Полезные советы для работы с терминалом:

1. *Используйте автодополнение ввода.* Можно набрать только первые буквы команды и нажать клавишу **Tab** — и недостающие буквы команды будут автоматически добавлены. Если же существует несколько команд, начинающихся с тех же символов, которые вы ввели, то **двойное нажатие Tab** выведет все эти команды в качестве подсказки.
2. Также используйте *автодополнение для имён и путей к файлам и директориям.* Работает аналогично автодополнению команд.
3. Если в командной строке нажать клавишу **вверх ↑**, то будет выведена *последняя введённая вами команда*. Нажимая дальше клавишу вверх ↑ вы будете перемещаться по истории выполненных ранее команд. Полная история хранится в файле `~/.bash_history`.
4. Если ввести два восклицательных знака **!!** и нажать ввод, то вы выполните последнюю введённую команду. Также есть *шорткат(shortcut - сочетание клавиш) и для использования аргумента от предыдущей команды*, для этого надо ввести **имя_команды !\$** и нажать ввод — вместо !\$ будет подставлен аргумент от предыдущей команды.
5. Если вы что-то напутали при вводе команд, то попробуйте нажать **Ctrl+C**, это сочетание *прекращает выполнение текущей команды и закрывает её*.
6. Можно прочитать *руководство к любой команде* и узнать, что она делает, какие у неё есть параметры и аргументы. Для этого надо набрать **man имя_команды**.
7. Для *очистки окна терминала* можно воспользоваться клавишами **Cmd + K (Ctrl + K)**

Полезные ссылки:

[Очень удобный справочник по работе с консолью](#)

[Основные команды терминала](#)

[Курс на loftblog](#)

[Курс по командной строке](#)

[Ещё один курс для углубления знаний](#)

2 Node.js и npm



Node.js – это JavaScript на сервере, он позволяет вам запускать JavaScript-код вне браузера. Чтобы ваш JavaScript код выполнялся на *вычислительной машине вне браузера* (на **backend**), он должен быть интерпретирован и, конечно же, выполнен. Именно это и делает Node.js. Для этого он использует движок V8 VM от Google — ту же самую среду исполнения для JavaScript, которую использует браузер Google Chrome. Кроме того, Node.js поставляется со множеством полезных модулей, так что вам не придется писать всё с нуля, как, например, вывод строки в консоль. Таким образом, Node.js состоит из 2 вещей: среды исполнения и полезных библиотек.



npm - пакетный менеджер Node.js. Бездонный ящик с инструментами, с помощью которого вы сможете получать и устанавливать, почти все, что только может пожелать душа веб-разработчика. С его помощью можно *управлять модулями (устанавливать, удалять, обновлять....) и зависимостями*. Во время установки node.js, npm автоматически установится вместе с ним.

Установка node.js

Установить Node.js очень просто. Если вы используете Windows или Mac, установочные файлы доступны на [странице загрузки](#).

Посмотреть, где установлен Node и проверить версию можно следующими командами:

```
$ which node
/usr/local/bin/node
$ node --version
v5.7.0
```

Еще один способ установки и управления версиями node.js - [nvm](#) (Node Version Manager) Это довольно простой скрипт, который позволяет устанавливать, переключать и удалять версии Node.js налету. Проще говоря, nvm даёт вам возможность держать на одной машине любое количество версий Node.js. При установке новой версии, для неё создаётся отдельная директория, например, 5.0.0 или 4.2.2. При переключении версий скрипт подменяет путь до Node.js в PATH.

REPL

Сразу после установки вам становится доступна новая команда **node**. Её можно использовать двумя разными способами. Первый способ — без аргументов. Откроется интерактивная оболочка (**REPL**: read-eval-print-loop), где вы можете выполнять обычный JavaScript-код. REPL— простая интерактивная среда программирования. В такой среде пользователь может вводить выражения, которые среда тут же будет вычислять, а результат вычисления отображать пользователю.

Функция **read** читает одно выражение и преобразует его в соответствующую структуру данных в памяти.

Функция **eval** принимает одну такую структуру данных и вычисляет соответствующее ей выражение.

Функция **print** принимает результат вычисления выражения и печатает его пользователю.

REPL-среда очень удобна при изучении нового языка, так как предоставляет пользователю быструю обратную связь.

[REPL API](#)

Давайте введем команду **node** в терминале и выполним простой код `console.log('Hello World')` :

```
$ node  
> console.log('Hello World');  
Hello World  
undefined
```

Node.js выполнит этот код. `undefined` после него выводится потому, что оболочка отображает возвращаемое значение каждой команды, а `console.log` ничего не возвращает.

Для выхода из оболочки необходимо набрать `Ctrl + C`

Глобальная установка пакетов



При разработке приложения Node.js можно создать *структуру каталогов для хранения приложения*. В каталоге проекта будет подкаталог **node_modules**, содержащий все модули, которые можно установить локально.

Модули, установленные *локально*, могут быть использованы внутри проекта с помощью функции `require()`.

Чтобы иметь возможность использовать командную строку пакета, его нужно установить глобально. То есть, для того чтобы мы могли работать в консоли с командой `gulp`, его нужно установить *глобально*. Это касается и модулей *bower*, *jade* и т.д.

Приложения, установленные глобально, хранятся в каталоге `~/npm/`. Функции их командной строки будут доступны всем приложениям Node.js, но такие пакеты *нельзя* использовать при помощи `require()`.

Чтобы глобально установить npm-пакет (модуль), необходимо ввести команду:

```
$ npm install browser-sync -g
```

Вместе с `browser-sync` установились и его зависимости.

Посмотреть список глобально установленных модулей можно так:

```
$ npm list -g
```

Чтобы увидеть список модулей, но без зависимостей, мы пишем:

```
$ npm list -g --depth=0
```

Модули, установленные глобально, мы можем вызывать прямо из терминала:

```
browser-sync start --server --files "css/*.css"
```

Локальная установка пакетов

Локальная установка используется в npm *по умолчанию*, то есть достаточно не использовать флаг `--global`. Пакет будет установлен в каталог **node_modules** родительского каталога.

```
$ npm install jquery
```

Посмотреть список локально установленных модулей (с зависимостями и без) можно так:

```
$ npm list  
$ npm list --depth=0
```

Управление зависимостями. Package.json

Если у вашего проекта много зависимостей, то устанавливать вручную каждый раз не очень удобно, поэтому npm использует файл **package.json**

```
{  
  "name": "project",  
  "devDependencies": {  
    "browser-sync": "^2.11.1",  
    "gulp": "^3.9.0"  
  }  
}
```




Файл [package.json](#) содержит общие сведения о вашем приложении. Он может содержать множество настроек, но выше указан необходимый минимум. Секция *dependencies* описывает имя и версию модулей, которые вы хотите установить. Вы можете перечислить в этой секции столько зависимостей, сколько захотите.

Он генерируется с помощью команды `npm init`, которая выведет в консоль несколько вопросов для создания файла. Для того, чтобы не отвечать “yes” на каждый вопрос, необходимо ввести:

```
$ npm init --yes // параметр --yes позволяет сразу же автоматом ответить на все вопросы yes, которые будут предложены при команде npm init
```

Теперь вместо установки зависимостей по одной мы можем установить все сразу командой:

```
$ npm install
```

При запуске этой команды `npm` будет искать *package.json* в текущей директории, и если найдёт, то установит каждую указанную в нём зависимость.

Это очень удобно, особенно, если наш проект использует много зависимостей (например, *jquery*, *normalize*, *browser-cookies* и т.д.) и если все они прописаны в файле *package.json*, то для того чтобы другой разработчик смог начать работать с вашим проектом, ему достаточно будет клонировать с гита сам проект, прописать команду `npm install` и все указанные зависимости скачаются автоматически.

3 Bower

Как разработчик, вы должны стремиться к тому, чтобы максимально автоматизировать свою работу. Неэффективно грузить файлы библиотек и фреймворков в проект по отдельным файлам. Используя Bower вам не нужно будет заниматься загрузкой вручную. Например, вы хотите установить Angular в ваш проект. Вместо того, чтобы искать в интернете, где скачать файлы Angular или подключить CDN, вы можете запустить команду `bower install angular` и bower установит фреймворк в ваш проект.



Что делает Bower? Устанавливает нужные проекту пакеты подходящих версий вместе с их зависимостями. Другими словами: загружает файлы нужных библиотек и всё, что нужно для их работы в специальную папку.

Полезные ссылки:

[Официальный сайт](#)

Обязательно посмотрите урок: [Bower - подробное руководство #1](#)

Обязательно ознакомьтесь с [подробной статьёй про bower на сайте loftblog](#)

Для работы с Бовером вам потребуются Node.js и Git. Установка:

```
npm install -g bower
```

Bower.json

bower.json и **.bowerrc** – это два основных файла для работы с Bower.

Настроив bower.json вы можете использовать одни и те же пакеты в разных проектах. По принципу работы он очень похож на package.json. Он состоит из различных зависимостей для конкретного проекта. Эти пакеты определяются в виде JSON-объектов.

Пример содержимого файла bower.json:

```
{
  "name": "project",
  "version": "1.0.1",
  "dependencies": {
    "jquery": "1.8",
    "normalize-css": "normalize.css#~3.0.3",
    "jquery-placeholder": "~2.3.1"
  }
}
```

В данном примере мы задаем имя проекта “project”, версию “1.0.1” и зависимости. После запуска команды **bower install**, будут установлены пакеты jQuery, Normalize.css, и jQuery Placeholder. Найти нужные пакеты вы можете на [официальном сайте](#)

Установка пакетов

.bowerrc используется для определения конфигурации Bower, задания путей для сохранения библиотек в вашем проекте.

Например, если вы хотите, чтобы все библиотеки сохранились в папке *public/libs*, задайте в .bowerrc следующую строку:

```
{
  "directory": "dev/libs/"
}
```

Если вы не укажете путь сохранения, Bower сохранит все пакеты в папку *bower_components*.

Если мы хотим установить какую-то конкретную версию пакета, то пишем сразу после названия хэштег и номер необходимой версии. К примеру, если мы хотим установить jquery версии 1.11, мы пишем:

```
bower install --save jquery#1.11
```

Флаг **--save** говорит Bower, что он должен сохранить имя пакета и его версию в файл-манифест — **bower.json**. Вместе с именами пакетов можно указать версии, с которыми ваш проект гарантированно будет работать.

Обновление и удаление пакетов

Для обновления библиотек используется команда `update`, однако, чтобы полноценно с ней работать, обязательно нужно создать файл-манифест `bower.json`

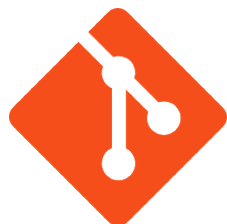
```
bower update jquery
```

Для удаления библиотек:

```
bower uninstall jquery
```

4 Git

Основательно о работе с Git мы поговорим на ближайших занятиях, сейчас же пройдемся по основам. Обязательно посмотрите [Видео для новичков по работе с Git](#)



Git (произн. «гит») — распределённая система управления версиями. Проект был создан Линусом Торвальдсом для управления разработкой ядра Linux, первая версия выпущена 7 апреля 2005 года. На сегодняшний день его поддерживает Джунио Хамано.

Зачем нужен git? Он синхронизирует работу с сайтом и хранит/обновляет версии файлов, что очень удобно, если над проектом работают несколько разработчиков одновременно. Гит дает возможность обновлять и править файлы проекта с учетом изменений внесенных другими.

Сейчас git используется во многих современных проектах: Android, jQuery, php, Drupal, Wine, Chromium некоторые версии Linux и т.д.

Если вы никогда ранее не использовали git, необходимо осуществить его установку. Выполните следующие команды, чтобы git узнал ваше имя и электронную почту:

```
git config --global user.name "Your Name"  
git config --global user.email "your_email@whatever.com"
```

Создаем репозиторий в существующем проекте

Если вы собираетесь начать использовать Git для уже существующего проекта, то вам необходимо перейти в проектный каталог и в командной строке ввести:

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем **.git**, содержащий все необходимые файлы репозитория — основу git-репозитория. На этом этапе ваш проект ещё не находится под версионным контролем.

Если вы хотите добавить под контроль git существующие файлы, вам нужно проиндексировать эти файлы и осуществить первую фиксацию изменений с помощью нескольких команд `git add` и `git commit`:

```
git add main.js // добавили файлы под контроль git
git commit -m // добавили main.js в проект, закоммитили файл
```

Клонирование существующего репозитория

Если вы желаете получить копию существующего репозитория Git, например, проекта, в котором вы хотите поучаствовать, то вам нужна команда `git clone`

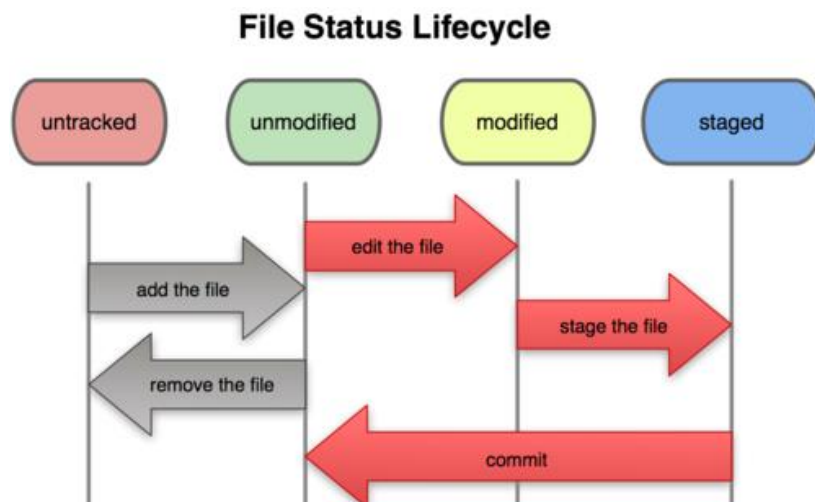
```
git clone git://github.com/your/path/to/repo.git
```

Определение состояния файлов

Каждый файл в вашем рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые).

Отслеживаемые файлы — это те файлы, которые были в последнем слепке состояния проекта (snapshot); они могут быть неизменёнными, изменёнными или подготовленными к коммиту (staged).

Неотслеживаемые файлы — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту. Когда вы впервые клонируете репозиторий, все файлы будут отслеживаемыми и неизменёнными, потому что вы только взяли их из хранилища (checked them out) и ничего пока не редактировали.



Основной инструмент, используемый для определения, какие файлы в каком состоянии находятся — это команда `git status`

```
$ git status
```

Если нам необходимо, чтобы git не следил за каким-то файлом или папкой, достаточно в корне проекта создать файл **.gitignore** и записать в него имя папки/ файла. После этого git перестанет отслеживать их.

5 BrowserSync



BrowserSync позволяет автоматически *перезагружать страницы* после изменения исходных файлов, в том числе и серверных (PHP, ASP, Rails и др); создаёт синхронизацию с десктопами, планшетами и смартфонами; *синхронизирует скроллинг* между браузерами, данные в формах и множество других действий; существует как Grunt так и Gulp-плагин; доступен на Windows, Linux, MacOS.

Чтобы воспользоваться *browserSync*, необходимо чтобы в системе был установлен *node.js*. Для установки BrowserSync необходимо ввести команду:

```
npm install -g browser-sync
```

Если мы хотим отслеживать *все css-файлы* и при их изменениях обновлять браузер автоматически, для этого вводим в консоль следующую команду:

```
browser-sync start --server --files "css/*.css"
```

`--server app` - указывает на то, в какой папке лежит наш index.html

`--files "app/**/*.css"` - указывает на то, за какими файлами следить

Таким образом мы запускаем локальный сервер и при изменении любого файла css в папке css автоматически обновляем страницу в браузере.

Важно: При использовании BrowserSync обязательно должен быть тег `<body>`

Не забывайте об этом даже тогда, когда хотите быстро протестировать что-то. Дело в том, что BrowserSync добавляет специальный функционал к тегу `<body>` и если этого тега нет, то работать BrowserSync не будет.

Полезные ссылки:

[Подробный урок по теме](#)

[Официальный сайт](#)

7 Gulp



Gulp - это инструмент сборки веб-приложения, позволяющий автоматизировать повторяющиеся задачи, такие как *сборка и минификация CSS- и JS-файлов, запуск тестов, перезагрузка браузера* и т.д. Тем самым **Gulp ускоряет и оптимизирует процесс веб-разработки.**

Gulp построен на Node.js, и файл сборки пишется на JavaScript. Сам по себе Gulp умеет не очень много, но имеет огромное количество плагинов, которые можно найти на странице со списком плагинов или просто поиском на npm. Например, есть плагины для запуска JSHint, компиляции SCSS, JADE, запуска тестов и даже для обновления номера версии сборки.

Установка Gulp крайне проста. Сначала установите его *глобально*:

```
npm install -g gulp
```

После этого ставим непосредственно в *сам проект*:

```
npm install --save-dev gulp
```

Для примера, давайте создадим *task для минификации JavaScript-файлов* (предположим, у нас есть один файл app.js в директории js). Для этого необходимо создать файл **gulpfile.js** в корне проекта, именно отсюда мы будем запускать task с помощью команды gulp. Поместим в файл gulpfile.js следующий код:

```
var gulp = require('gulp'), // подключаем галп в проекте
    uglify = require('gulp-uglify'); // подключаем плагин в проекте

gulp.task('minify', function () { // создаем taskу с названием minify
  gulp.src('js/app.js') // указываем путь входных файлов
  .pipe(uglify()) // запускаем минификацию на этих файлах
  .pipe(gulp.dest('build')); // кидаем готовый в папку build
});
```

Затем установим плагин **gulp-uglify**, для этого открываем консоль и вводим:

```
npm install --save-dev gulp-uglify
```

Теперь можно в консоли написать команду `gulp minify`, после чего наш task minify запустится и в папке build создастся минифицированный app.js

Функция **gulp.src()** в качестве параметра принимает *маску файлов* и возвращает поток, представляющий эти файлы, который уже может быть передан на вход плагинам. Примерами таких масок могут быть:

- `js/app.js` — файл `app.js` в директории `js`
- `js/*.js` — все файлы с расширением `.js` в директории `js`
- `js/**/*.js` — все файлы с расширением `.js` в директории `js` и всех дочерних директориях
- `!js/app.js` — исключает определенный файл
- `*.(js|css)` — все файлы в корневой директории с расширениями `.js` или `.css`

Мы рассмотрели самую малую часть возможностей Gulp, более подробно познакомимся с ним на последующих занятиях.

Полезные ссылки:

[Официальный сайт](#)

[Поиск плагинов](#)

[Видеокурс о Gulp](#)