**שקף 4:**

אז בואו נדבר על ה-child process module של node .

אפשר בקלות לעשות spawn ל-process נוסף אם נשתמש ב-child process module , כאשר ה-child processes יכולים בקלות לתקשר אחד עם השני בעזרת messages .

ה- child process מאפשר לנו לגשת לכל הפונקציונליות של מערכת ההפעלה בתוך child process שהוא process נוסף.

אנחנו יכולים לשלוט ב-Input stream של ה-child process , ולהאזין ל-output stream שלו. אנחנו יכולים לשלוט בארגומנטים שאנחנו מעבירים למערכת ההפעלה , ואנחנו יכולים לעשות מה שאנחנו רוצים עם ה- output שאנחנו מקבלים.

כל ה-Inputs וה-Outputs יכולים להיות מוצגים בעזרת Node.js streams .

**שקף 5:**

הפעולות שהמודול חושף הן : exec, execFile, Spawn , fork

**שקף 6:**

המתודה הזו תעשה spawn ל-subshell ותעשה Execute ל-command בתןך shell ותעשה buffer ל-general data . כאשר ה-child process מסיים, ה-callback תקרא עם :

Buffered Data אם הפקודה התבצעה בהצלחה

Error – אם הפקודה נכשלה.

**שקף 7:**

עושה execute ל-external application , נותן Optional arguments לאפליקציה ו-buffered output אחרי שהאפליקציה יצאה.

משתמש ב-PATH environment variable כדי להבין איפה הספריות של ה-Executables נמצאות.

אם האפליקציה לא נמצאת כחלק מה-paths ב-path variable , צריך לציין path בצורה explicit .

כדאי להשתמש בזה כאשר רוצים לבצע אפליקציה ולהביא את ה-Output . לדוגמא אפליקציית עיבוד תמונה שצריכה להמיר קבצים לפורמט מסויים , ואכפת לנו רק אם ההמרה הצליחה או לא .

לא להשתמש בו כשצריך לעשות processing להרבה data או שצריך Real time processing .

**שקף 8:**

בואו נעבור על דןגמת קוד קטנה שמריצה Node –v ב-exec. ואחר כך נראה איך היא עובדת עם ExecFile .

עבור Exec :

אנחנו נריץ על ה-shell את הפקודה node –v . מעבירים את הפקודה Node –v כארגומנט ראשון ל-exec . אנחנו מצפים לקבל את ה-Response ב-callback שקיבלנו. Stdout הוא מה שאנחנו בדרך כלל צריכים. זה ה-Output של הפקודה. Stderr יהיה ריק אם הכל בסדר.

עבור execFile :

בדוגמת קוד הזו , האפליקציה node תהיה executed עם הארגומנט version . כאשר ה-external application קיימת, ה-callback function נקראת. ה-callback function מכילה את ה-stdout ואת ה-stderr של ה-child process . ה-output stdout של ה-external application הוא Buffered internally .

**שקף 9:**

ה-spawn method עושה spawn ל-exeternal application ב-process חדש ומחזירה streaming interface ל-IO .

מתי כדאי להשתמש בו ?

הוא מחזיר stream based object , ולכן הוא מצויין לעבודה עם אפליקציות שמחזירות הרבה data ואו כדי לעבוד עם ה-data מיד כשצריך אותו. אנחנו יכולים להשתמש בכל היתרונות של stream שהם :

* Low memory footprint
* Automatically handle back-pressure
* Lazily produce or consume data in buffered chunks.
* Evented and non-blocking
* Buffers allow you to work around the V8 heap memory limit

**שקף 10:**

ב- Node.js function שאנחנו רואים כאן – דרך אגב – זה middleware של Express , אנחנו קוראים את התמונה בעזרת stream . אז אנחנו עושים convert לתמונה בעזרת spawn method . אז אנחנו נותנים ל-child process את ה-image stream . כל עוד הproc object מייצר data , אנחנו כוצבים את ה-data ל-resp (סוג של writable stream , אני לא אכנס לזה כרגע) , וה-users יכולים לראות את התמונה מיד מבלי לחכות לכל התמונה שתעלה.

**שקף 11:**

המתודה האחרונה שאני רוצה לדבר עליה היא FORK , שבעצם פותחת process נוסף שיש לו זיכרון וinstance של v8 משלו, ופותחת IPC channel שמאפשר לי לתקשר בין ה-processes .

השימוש הכי נפוץ של זה זה ה-cluster module – כולו בנוי על השימוש במתודה fork ולשם אני רוצה שנצלול.

**שקף 13**

אז Node.js רץ על single process , מה שאומר שהוא לא משתמש ביתרונות של multi core machines . אם יש לך מכונה עם 8 cpus ואתה מריץ אפליקציית node בצורה הרגילה , היא תרוץ על process אחד ונבזבז כוח של 8 cpus . בכל אופן – לא משנה איך נסתכל על זה – הload עבור CPU אחד הוא מוגבל.

**שקף 14:**

אז ה-cluster module בא לפתור את הבעיה הזאת. הוא נותן לנו סט של פונקציות שיעזור לנו להשתמש בכל הCPUs . מה שה-cluster module עושה מאחורי הקלעים הוא לעשות fork לכמה process חדשים , כאשר המטרה המרכזית היא להיות מסוגלים לטפל ביותר load בקלות ולעשות scale לאפליקציה בקלות.

**שקף 15:**

בcluster module מאפשר לנו ליצור chikd process שחולקים אותו server port . הם spawned על ידי fork process , אז הם יכולים לתקשר אחד עם השני בעזרת IPC .

חשוב לזכור שכל process הוא עצמאי . הוא מנותק מה-parent שלו לחלוטים חוץ מה-IPC שמתקיים בין השניים. לכל process יש זיכרון משלו, עם Instance של V 8 .

**שקף 16:**

ה-cluster module עובד בdesign של master process ו-worker process , כאשר ה-master process פותח את ה-worker processes , ובדרך כלל נפתחים processes כמספר הCPUs , המאסטר מחלק עבודה לprocess האחרים.

**שקף 17:**

אז בואו נראה דוגמת קוד שתסביר לנו על מה מדובר.

אנחנו ניצור master process שמוצא מה מספר ה-CPUs ועושה fork ל-worker process לכל CPU . כל child process מדפיס message ל-console ויוצא.

אז בואו נריץ את הקוד – להריץ מהcommand line כדי לראות את ה-output ב-console .

נסביר קצת מה קורה פה :

כשמריצים את server.js נוצר process ומתחיל להריץ את הקוד שלנו. אנחנו עושים Import ל-cluster module ואנחנו בודקים בעזרת if האם אנחנו בתוך ה-master process – בודקים property שנקרא isMaster .

ה-process הראשון שנומר הוא המאסטר, לכן הpropery isMatser יחזיר true . מה שירוץ זה הפונקציה ה-masterProcess . מה שהפונקציה עושה הוא לעשות loop על מספר הCPUs שיש במכונה ועושה fork לprocess הנוכחי בעזרת המתודה cluster.fork .

מה שהמתודה fork עושה הוא לפתוח process חדש , כאשר הפונקציונליות היא כמו להריץ node server.js מה-command line .אז זה בעצם כאילו יש לכם הרבה processes שמריצים node server.js במחשב ולא אחד.

אז שהchild process נוצר ועושים לו execute , הוא עושה בדיוק מה שה-master עושה , כלומר – Import ל-cluster module ולעשות Execute ל-if statement . ההבדל הוא שה-isMaster ישתערך ל-false , אז הם יריצו את הפונקציה child process במקום.

פה רק לשם הדוגמא אנחנו מסיימים את ה-master וה-child process . בעזרת process.exit שמחזיר קוד 0.

**שקף 18:**

כשנוצר worker , IPC channel נוצר בין ה-worker וה-master ומאפשר תקשורת ביניהם עם המתודה send , שמקבלת אובייקט javascript כפרמטר .

צריך לזכור שהם processes שונים אז אין להם shared memory כדרך תקשורת .

שליחה מה-master ל-child :

someChild.send

שליחה מהchild ל-master :

Process.send

**שקף 19:**

אז בואו נשנה קצת את הקוד שלנו כדי שה-processes יתחילו לשלוח messages אחד לשני .

נסתכל על השינויים שנעו במתודה childProcess :

מקשיבים ל-message event בעזרת process.on (‘message’ . לאחר מכן שולחים Message בעזרת process.send( עאשר ה-message הוא אובייקט javascript .

ה-master process מחולקת כעת ל-2 חלקים , בחלק הראשון עושים fork לprocesses חדשים לפי מספר ה-CPUs , כאשר המתודה fork מחזירה אובייקט worker שאותו אוספים במערך. הוא מייצג process .

רושמים Listener כדי לקבל messages שמגיעים מהworker instance .

אחר כך אנחנו עושים loop על ה-workers ושולחים message מתוך ה-master ל-worker קונקרטי.

**שקף 20:**

בעזרת ה-cluster module האפליקציה שלנו תוכל לשפר את ה-load שהיא יכולה לטפל בו אם היא עובדת על multi core machine . זה יכול להיות מאוד חשוב כאשר יש לנו servers שחושף API מסויים , למשל שרת שכתוב ב-Express .

בעזרת ה-cluster module תוכלו לעשות load balance ל-requests בין ה-worker processes השונים , ולשפר את הthroughput של האפליקציה. שימו לב שה-response time לא ישתפר.

**שקף 21:**

אחריות של ה-master process :

Spawn workers

Re-spawn workers

Load balancing the requests

כל ה-connections מה-clients מגיעים ל-matser process

ה-worker process אחראי על יצרית ה-Response

**שקף 22:**

נסתכל על דוגמא ממש בסיסית ליצירת שרת http מאוד פשוט. חילקנו את הקוד לשני חלקים, כאשר ה-master process עושה fork ל-workers processes עבור כל Node . כל אחד מה-workers יוצר http server שמאזין על port 3000 ומחזיר hello world .

נריץ מהcommand line

By default , ב-windows ה-routing הוא לא round robin אז ככה הגדרתי את זה.