**שקף 3**

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 4 :**

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 5:**

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 6:**

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 7:**

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 8:**

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 9:**

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 10:**

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 11:**

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 12:**

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.

**שקף 12:**

עכשיו, בואו קצת נחשוב על כמות הקוד שלא כתבנו. לא כתבנו callbacks עבור הsuccess ןהfail של הpromise , לא השתמשנו במשתני ביניים, הטיפול בשגיאות יטפל גם בשגיאות סינכרוניות וגם בשגיאות א-בינכרוניות. יש כאן המון קוד שנחסך. אנחנו נמנעים מnesting והקוד הופך להיות הרבה יותר איטואיטיבי שזו הבעיה הגדולה של קוד javascript .

**שקף 13:**

אז async await סוף סוף נותן לנו את האפשרות לטפל בשגיאות סינכרוניות ושגיאות א-סינכרוניות באמצעות אותו מנגנון – Try / catch . שזה המנגנון שאנחנו מכירים , מאוד אינטואיטיבי וכו'. בדוגמא שכאן אנחנו יכולים לראות שה-async await יטפל גם בשגיאות סינכרוניות מגיעות מה-Json parser , וגם בשגיאות א סינכרוניות שיגיעו מה-promise . טיפול בשגיאות נהיה הרבה יותר פשוט.

**שקף 14:**

אז בואו נסתכל על קטע הקוד הבא, יש פה use case שעובדים איתו הרבה – לפחות אני נתקלתי בו הרבה , שיש לנו promise 1 שמחזיר ערך, ומשתמש בו וקורא ל-promise 2 , ואז משתמש בתוצאות של ה-Resolve של שני ה-promises וקורא לpromise 3. אז אם נישאר ברמת ה-promises אז יש כל מיני פתרונות לעניין, כמו להכניס למערך וכו, אבל עם async await הקןד נהיה ממש ממש ממש פשוט.

**שקף 15:**

הדבר האחרון שאני רוצה להראות לכם זה איך לעשות parallel execution עם async await , כאן יש דוגמא של מערך של promises , וה-await מופיע לפני promise all .