מהו שרת

בקשת REST שמגיעה מקליינט => קאש, ולידציות, עדכון DB

דרישה בסיסית משרת:

* טיפול בקבצים
* יכולת לתקשר ברחבי האינטרנט
* יכולת לקבל בקשות ולשלוח ריספונסים
* יכולת טיפול בפעולות שלוחות זמן IO גבוה.

דרישות שלנו משרתים

* לייטנסי נמוך - זמן תגובה קטן, בקשות מהירות
* רוצים להתמודד עם IO גבוה – אינטרנט גדול.
* רוצים סקייל החוצה ומעלה
* עלויות (מחשוב ענן)
* קהילה חזקה מאחורי הטכנולוגיה
* תמיכה בספריות packages, מודולים ומידור
* זמן פיתוח

דרישות שלנו מ-JS: כמו ילד שהגיע לגיל מצוות ודורשים ממנו יותר:

* מודולריות של הקוד – (לצורך שיפור תחזוקה, קריאות וכו')
* יכולת להגיב ל-events (של מערכת הפעלה / אפליקטיביים)

מי משתמש ב-Node.JS:

Netflix, Walmart, Microsoft, uber, linkedin, ebay, paypal ,NASA ועוד...

כן נאסא – יש node על לויינים שלהם.

מהו Node.JS?

בהגדרה רשמית – מנוע אסינכרוני מכוון-אירועים לבניית אפליקציות רשת סקלביליות.

אסינכרוניות בג'אווהסקריפט – callback functions. והיום פרומיסים.

מוכוון איבנטים – ה-eventLoop שנמצא בבסיסה של JS.

הליבה של Node.JS:

* הקוד שלנו – 100% ג'אווהסקריפט
* NodeJS – 50% ג'אווהסקריפט , 50% C++
  + נותן לנו ממשק שמאפשר לנו לכתוב 100% ג'אווהסקריפט בקוד האפליקציה שלנו.
  + עוטף את החלקים שמתחת ב-API’s ברורים ונוחים לשימוש בקוד JS – מודולים (fs, http וכו').
* מנוע V8 של גוגל כרום – "השרת" (סינגל ת'רד)- JS+C++
  + מפרסר את קוד האפליקציה שאנחנו כותבים ב-JS. (בדיוק כמו שכרום מפרסר קבצי JS).
  + עושה לו אופטימיזציה
* מנוע Libuv (C++) (מולטי ת'רד – מחזיקה ת'רד פול בתוכה):
  + מנוע קרוס-פלאטפורם (פועל בכל מערכת הפעלה), הפועל בתחום IO אסינכרוני (ת'רדים).
  + הכוונה – פותח ת'רדים, פועל מול file system, פועל מול כרטיסי הרשת וכד'.
  + הוא עושה בשבילנו את עבודת ה-IO מול מערכת ההפעלה.
* מערכת הפעלה

עיקר העבודה של Node היא לתווך בין V8 ל-Libuv.

רוב הסיכויים שבין הפונקציה שכתבתים לקולבאק שלה – מתרחש הקסם ה-non blocking של Node

תוכנית נוד = פרוסס עם ת'רד אחד שבתוכו קיים EL.

לולאה שבכל איטרציה בודקת את הדברים הבאים:

* האם יש טיימרים שהגיע הזמן לטפל בהם (setTimout, setInterval, setImmediate) ?
* האם יש משימות OS בטיפול (למשל האזנה לפורט)?
* האם יש אופרציה בביצוע כרגע (למשל קריאת קובץ)?

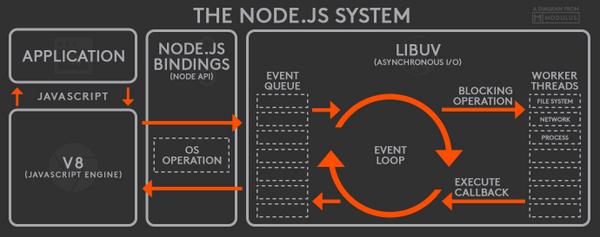
אם אחד מהם כן – הלולאה תבצע את האיטרציה הנוכחית, אחרת התוכנית תסתיים.

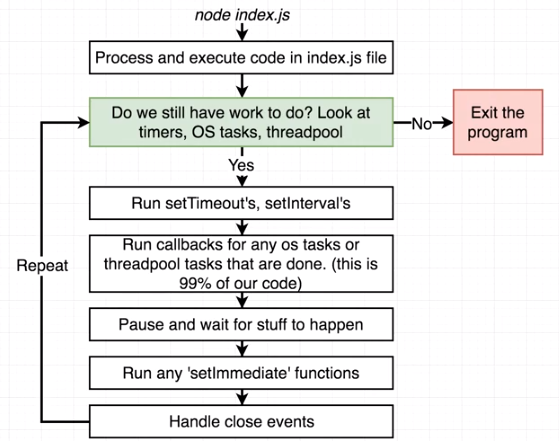
במהלך ביצוע האיטרציה – אם טיימר צריך טיפול – ייקרא הקולבק שלו, ואם משימת OS או אופרציה יורות איבנטים (לדוגמא – בקשת http שמגיעה לפורט מסוים, או צ'אנק של קובץ שנקרא – כל אלה יורים איבנטים) ואז יופעלו הקולבקים של אותם איבנטים.

עוד בביצוע האיטרציה – אם צריך לעשות cleanup – כמו לשחרר זיכרון לאובייקטים "מתים" וכו', נוד יעשה אותם.

בסוף האיטרציה הנוכחית, התוכנית נכנסת למצב המתנה ולא מתחילה ישר עוד איטרציה, אלא חוסכת משאבים וממתינה בסבלנות, עד שנורה איבנט שקשור לאחת מ-3 הפעולות (טיימרים / OS / אופרציות).

ברגע שנורה איבנט שכזה, התוכנית מתעוררת ומתחילה לבצע את האיטרציה הבאה של ה-EL.



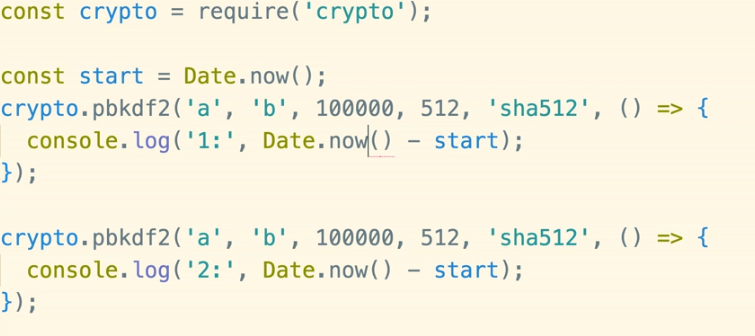


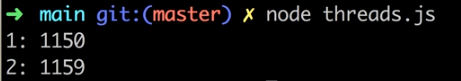
האם נוד היא סינגל ת'רדד?

ה-EL – הוא סינגל ת'רדד.

חלקים משמעותיים בפונקציונליות ובמנוע של נוד (libuv) – הם מולטי ת'רדד. הם רצים מחוץ ל-EL.

או שהם רצים תחת הת'רד פול של libuv:





מתייחסים לאותו מועד start, ז"א ששניהם התבצעו במקביל, אחרת שורה 2 הייתה כפולה בזמן.

ה-thread pool של libuv מספק לנו 4 ת'רדים באופן דיפולטי. יש לנו אפשרות לשנות את ההגדרה הזו בקוד האפליקציה שלנו (שכתוב ב-JS) באופן כזה שישנה הגדרות בת'רד פול של libuv:



או שהם רצים תחת מערכת ההפעלה – למשל בבקשות רשת – אם נוציא בקשת פוסט מהשרת שלנו החוצה, libuv תעביר את הטיפול למערכת ההפעלה OsTask, ותחכה שהפעולה תסתיים בהצלחה.

הפעולה עצמה תתבצע בצורה אסינכרונית – באופן שבו מערכת ההפעלה תראה לנכון לבצע - בפרוססים / ת'רדים וכו'.

ככה שאם באפליקציית הנוד שלנו נייצר 6 פניות לגוגל אחת אחרי השניה, אנחנו נראה שכל 6 הפניות יחזרו כמעט באותו זמן.

לא משנה אם זה מתבצע בת'רד פול של libuv או במערכת ההפעלה – זה לא חוסם לנו את ריצת התוכנית שלנו שכתובה ב-JS.

האידיאולוגיה של נוד:

להפוך דברים לפשוטים !

היתרונות של נוד:

פשטות - אפשר לכתוב שרת http ב-7 שורות קוד, בלי קונפיגורציות, בילדים, תיקיות bin

קרוס פלאטפורם

Lightweight – צורך מעט מאוד משאבים (Module based), בהתאם לצורך.

אקוסיסטם עצום – יכולת להתקין ספריות (פקג'ים) בלחיצת כפתור. כל ספריה שאי פעם חשבתם שצריך – כנראה כבר מישהו כתב אותה והעלה אותה, ההתקנה שלה בפרויקט שלכם היא פשוטה ביותר!.

קהילה חזקה מאחורי – קהילת NODE + קהילת Javascript.

סקיילינג ו-load balancing – ניתן לבצע סקייל בקלות – באמצעות cluster mode מייצרים עבור כל ליבת Cpu עותק של שרת ה-node. כל הליבות יכולות למקבל טיפולים בפניות.

שני מצבים:

* באלאנסינג פנימי - אחד השרתים מתפקד גם כמאסטר של היתר. הוא אחראי על יצירת / הריסת אינסטנסים בהתאם לעומס, הוא אחראי על חלוקת הטיפול ביניהם.
* באלאנסינג חיצוני – אף אחד מהשרתים לא מאסטר. הניהול מתבצע ממקום שבו האפליקציה מתארחת (nginx/ pm2).

באלאנסינג חיצוני + מוניטורינג – באמצעות PM2 – "תרים לי קלאסטר של 10 שרתי נוד – ותעקוב אחריהם, אם מישהו נופל תרים אותו מחדש".

תמיכה בתקני ES – מובילות טכנולוגית שמכתיבה לאחר מכן את התקן הכללי.

* Import / export
* Async await
* Workers
* Event Emitters
* Pipes
* ועוד

טייפסקריפט – עובד מצויין – כותבים את השרת ב-TS, מקמפלים, ומריצים את הקוד המקומפל.

אבני הבניין של NODE – מודולים – יחידות קוד גדולות שמחולקות לנושים

HTTP, FS , Events , Stream , Cluster

מה להיות טובים ב-Node:

70% להיות טובים ב-Javascript

10% להכיר את המודולים של Node ואיך להשתמש בהם.

10% להיפתח לאקוסיסטם – לא לפחד להשתמש ב-NPM

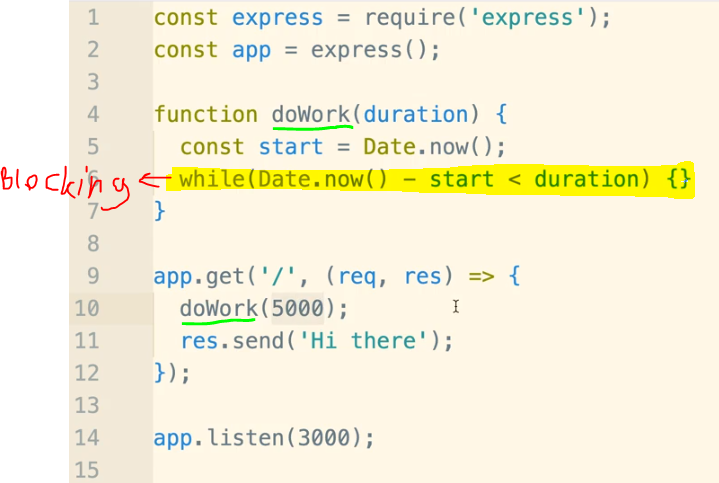
10% לא לתקוע את ה-eventLoop. כלומר להבין איך לנצל לטובתנו את הצד ה-סינגל ת'רדי, ואת הצד המולטי ת'רדי.

שיפורי ביצועים בנוד:

* מצב cluster (יציב)
* שימוש ב-worker threads (אקספרימנטלי)

בעיית ביצועים מוכרת – חסימת ה- EL.

רקע – אם נכתוב אלגוריתמיקה כבדה על שרת הנוד שלנו (לולאות while ענקיות), אנחנו גומרים על ה-EL. (מהסיבה שהקוד לא רץ על ת'רד פול / מע' הפעלה), אלא נוד תריץ את הקוד שלנו בסינגל ת'רד ולא תוכל לעשות שום דבר אחר בזמן הזה – לא לטפל בשום בקשה, כלום – ה-EL כולו מושקע בלרוץ על איזושהי לולאה...



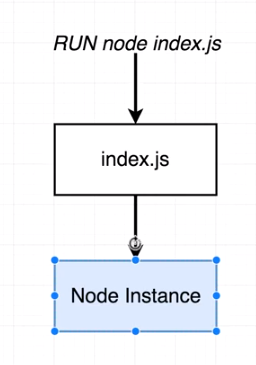
קלאסטרינג – הרצת כמה אינסטנסים (פרוססים), של אותה אפליקצייה. כל אחת בעלת הת'רד שלה.

הפרוססים הללו ינוהלו תחת cluster manager אחד, שאחראי לעשות לפרוססים מוניטורינג ולבדוק את "הבריאות" שלהם.

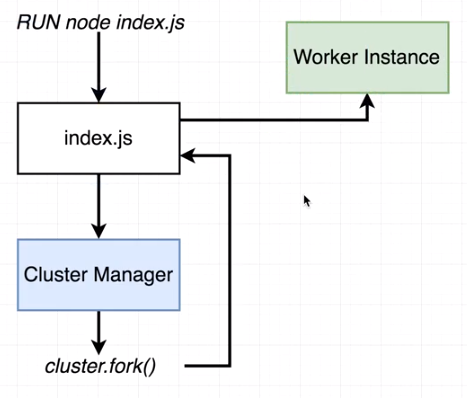
הקלאסטר מנג'ר לא מריץ "קוד אפליקציה", כלומר לא יטפל בבקשות / יבצע ביזנס / ישמור ל-DB ...

מה שהוא כן יעשה – ירים אינסטנסים חדשים של האפליקציה, יסגור אינסטנסים קיימים, יעשה להם ריסטרט וכו'.

ריצה של תוכנית רגילה (לינארי):



ריצה של תוכנית במצב קלאסטר – האינסטנס הראשון נוצר במצב מנג'ר, והוא יוצר אינסטנסים נוספים במצבי "בן", ע"י הספרייה cluster של נוד, ושימוש בפונקציה fork.



cluster.isMaster - פרופרטי שמחזיר true רק עבור הפרוסס הראשי (האינסטנס הראשון).

ככה אפשר לכתוב קטע קוד שמתייחס לפעולות של ה-cluster manager בלבד, בתוך קוד האפליקציה שלנו.

קלאסטרינג הוא פתרון לבעיית תקיעת ה-EL כשמדובר בטיפול בפניות לשרת – אם יהיה יותר מאינסטנס אחד של השרת, אז גם אם אחת הבקשות תפעיל בסופו של דבר איזו לולאה חוסמת / אלגוריתם כבד ותגרום לאותה בקשה להיות איטית, יהיו אינסטנסים נוספים של השרת שיוכלו לטפל בינתיים בבקשות אחרות (אשר אחרת היו צריכות להמתין).

הגדלת כמות אינסטנסים כפיתרון לפניות היא לא אינסופית – היא מוגבלת בסופו של דבר לליבות מעבד – אם תהיה כמות משמעותית גדולה של אינסטנסים מכמות הליבות – (6 אינסטנסים על 2 ליבות), הם אמנם יצליחו לטפל ב-6 בקשות במקביל, אבל הביצועים לכל הבקשות יחד יירדו דרסטית.

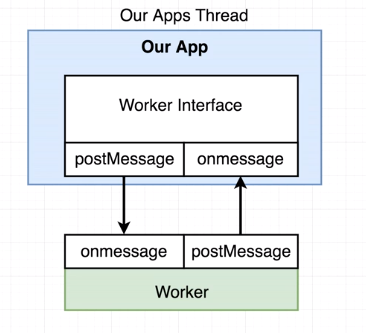
כלל אצבע – להתאים את כמות הילדים לכמות ליבות (או ליבות לוגיות לפי יכולות מולטית'רדינג פר מעבד)

נכון יותר יהיה לנהל קלאסטר מנג'ר מחוץ לאפליקציה שלנו – למשל על ידי pm2.

Pm2 start index.js –i 0

ה-i מייצג כמות אינסטנסים, ה-0 הוא פרמטר שאומר ל-pm2 להחליט על הכמות לפי כמות ליבות פיזיות / לוגיות.

Worker Threads (אקספרימנטלי):



הריבוע הכחול – הת'רד הראשי של האפליקציה שלנו – ה-EL.

הריבוע הירוק – ת'רד נפרד שפתחנו. בגלל שהוא לא חלק מה-EL הוא יכול לעשות חישובים גדולים בלי לתקוע את ה-EL.

צריך לזכור שכשאנחנו משתמשים במודולים של NODE הם משתמשים למטה איפה שצריך בת'רדים של Libuv או של מערכת ההפעלה, אז אין צורך לפתוח Workers בשבילם, הם גם ככה רצים בצורה אסינכרונית.

**כלומר, צריך להשתמש ב-Workers רק בשביל קוד חוסם שאנחנו מבצעים – בקוד ה-JS.**

בין הת'רדים אין בדיוק מרחב משותף – כלומר אין רפרנסים למשתנים במרחב זיכרון משותף.

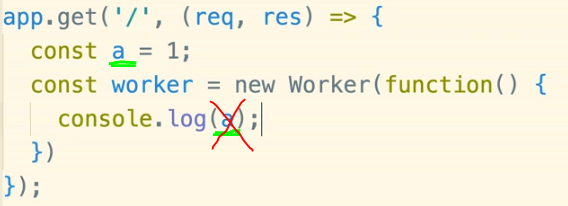
התקשורת היא מאוד "ישירה" באופן כזה שהת'רדים מתקשרים בהודעות (events / messages). אחד עם השני בהם הם יכולים להעביר מידע.

הממשק זהה ב-2 כיווני התקשורת:

postMessage מעביר מידע מגורם א' לגורם ב'.

כאשר המידע מגיע לגורם ב' יופעל onMessage שלו.

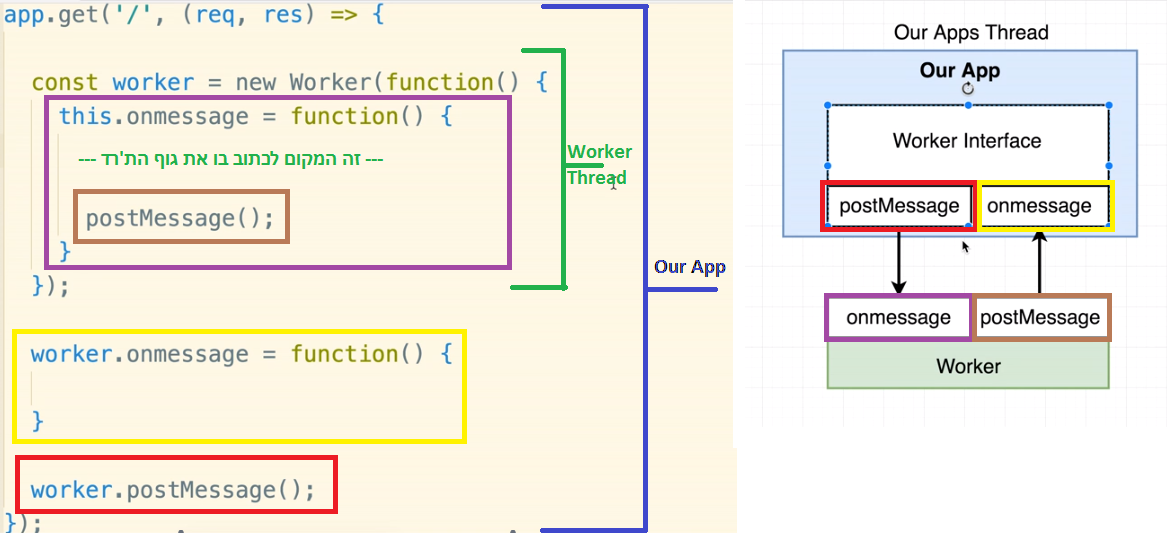
**הערה חשוב על Scope של WORKER:**



אין ל-WORKER יכולת לגשת למשתנים שהוגדרו מחוצה לו, גם אם אנחנו כותבים אותו Inline style.

הסיבה היא שכשהתוכנית רצה, כל קטע הקוד של ה-worker עובד "סטרינגיפקציה" ומורץ מחדש במרחב זיכרון אחרי לגמרי, שאינו קשור למרחב הזיכרון של האפליקציה.

שלד אפליקציה + Worker Thread:



* הערה חשובה – בכוונה משתמשים בסינתקס function(){} ולא בסינתקס ()=>{} כי חייבים פה סקופ נפרד. אחרת נשארים בסקופ.

הסיבה היא שה-this שבתוך סקופ ה-worker הוא אובייקט שמייצג את הת'רד החדש. אם היינו פונים ל-this דרך arrow function אז היינו נשארים באפליקציה ולא בת'רד החדש.

* onmessage – באותיות קטנות בכוונה – זה הסינתקס.

האובייקט שעובר ב-postMessage נקלט ב-onmessage תחת אובייקט message שבו יש פרופרטי data שבה האובייקט שהעברנו.