**שקף 5:**

אז כאן יש דיאגרמה של הארכיטקטורה של NodeJS . אני בעיקר רוצה שנתמקד בצד שמאל של הדיאגרמה . שימו לב מה קורה כאן :

יש כאן את ה-queue של ה-events שבו אנחנו מקבלים את ה-events שצריכים להתבצע.

יש כאן event loop שרץ על Thread אחד. הוא בעצם מוציא את ה-events ומבצע אותם.

Blocking operations בעצם הן offload ל-worker pool .

הכל ממומש על LibUV .

**שקף 6:**

אז יש לנו מספר מאוד מצומצם של threads – אחד לevent loop ועוד מספר קבוע עבור ה-workers threads . לכן , במקום לפתוח thread נוסף לכל request , בעזרת מספר מצומצם של threads אנחנו משרתים יותר בקשות , ויכולים להתרכז בעבודה שאנחנו צריכים לעשות במקום בלבזבז משאבי CPU וזיברון על פתיחת Threads נוספים. בעצם כל thread לוקח מאיתנו overhead גדול כזה. ומביא איתו גם בנוסף overhead של context switching ו-thread scheduling .

**שקף 7:**

אז בעצם – blocking thread יכול להיות ברמת ה-event loop וברמת ה-workers pool . ברמת ה-event loop – אלגןריתם שהוא CPU intensive לדוגמא. ברגע ש-thread חסום הוא לא יכול לטפל ב-requests של clients אחרים וכל השרת עובד ב-throughput הרבה יותר נמוך. מכיוון שיש לנו מספר קטן של Threads , כל Thread הוא קריטי – ולכן אנחנו חייבים לא לחסום את ה-event loop .

**שקף 8:**

אז המסקנות הגדולות מהעניין הן בעצם 3 מסקנות עיקריות.

קודם כל , לא להשתמש ב-API סינכרוני. מפני שאם מדובר בפעולה I/O והיא סינכרונית, אתם הולכים לתקוע את כל ה-event loop שלכם, שיהיה עסוק בלבצע את הפעולה הסינכרונית ולא יוכל לטפל בבקשות אחרות שמגיעות לשרת שלכם.

הדבר השני – הוא לעשות כמה שיותר parallel I/O . עבור פעולות א –סינכרוניות שיכולות להיות I/O או timers , אם תקבעו אותן במקביל הן יכולות להתבצע במקביל אם יש לכם multi core – מכיוון שיש לכם כמה threads ב-worker pool שלכם. מכיוון שפעולות א-סינכרוניות הן offloaded ל-worker pool – אז אנחנו רוצים שה-event loop יעשה schedule לכמה שיותר מהן במקביל.

הדבר השלישי – אנחנו צריכים שכל client שפונה לשרת שלנו יעשה assign לכמות עבודה שהיא יחסית קטנה – מכיוון שאם הוא יעשה assign לכמות עבודה גדולה אנחנו נחסום את אחד מהthreads שיש לנו ומכיוון שאין לנו הרבה Threads כל אחד מהם הוא קריטי ואנחנו לא רוצים לחסום אותו.

**שקף 9:**

בואו נדבר על קוד סינכרוני ועל API סינכרוני לפונקציות.

**שקף 10:**

אז יש כאן את ה-API של read file . ב-API הזה ישנו API סינכרוני וAPI א-סינכרוני. הAPI הסינכרוני הוא מאוד בעייתי.הוא יתפוס לכם את הThread של ה-event loop והשרת שלכם לא יוכל לטפל בבקשות אחרות. לכן תמיד תשתמשו ב- API האסינכרוני ואל תשתמשו ב-API הסינכרוני.

**שקף 11 :**

בואו נדבר על איך אפשר לכתוב קוד סיריאלי מבלי לשים לב ב-Node .

**שקף 12:**

אז מה זה Serial code ?

מה שזה אומר זה שאנחנו מריצים פונקציות אחת אחרי השניה , ובמצב כזה אנחנו גם עושים Execution לפעולות א-סינכרוניות אחת אחרי השניה. כלומר , רק כשה-execution של פעולה א-סינכרונית אחת נגמרת , מתחיל ה-execution של הפעולה הא-סינכרונית השניה.

**שקף 13:**

אז למה להימנע serial code ?

אז ב-Node , בגלל מודל ה-event loop שלו, אין קוד שרץ במקביל. מה שכן יכול להתבצע במקביל הוא פעולות א-סינכרוניות, בגלל ה-worker pool של node.js . אז ה-scheduling תמיד יהיה One by one , אבל ה-execution יכול להיות ב-parallel .

תשימו לב – פעולות ה-I/O מתבצעות במקביל – אבל הן לא באמת רצות במקביל – בשביל הרצה מקבילית אמיתית אתם חייבים multi core CPU .

בל אופן – אם יש לכם serial I/O , אתם מפספסים את היתרון הגדול של Node.js שהוא הרצת I/O במקביל.

**שקף 14:**

אז בואו נדבר על איך אפשר ליצור serial code ב-Node.js .במיוחד בדרכים שאנחנו לא מודעים אליהם. בואו נתחיל מ-callbacks . אנחנו רואים כאן שיש לנו שתי קריאות לDB . אחרי שהקריאה ל-getUsers מסתיימת , מתחילה הקריאה ל-getCategories . תסתכלו על ה-flow ותראו שרק אחרי שהקריאה ל-users חוזרת , עושים schedule לשאילתה של ה-getCategories . שימו לב למשהו חשוב – השאילתות האלה לא תלויות אחת בשניה. לכן הרבה יותר יעיל מבחינת ביצועים לעשות כאן קוד מקבילי ולא לעבוד בצורה סיריאלית. זה למשל משהו שלא צועק לנו – אנחנו רגילים לכתוב ככה , אבל זה קוד שאפשר בוודאות לשפר את ה-performance שלו בצורה רצינית.

**שקף 15:**

אז אם בדוגמא הקודמת זה היה קוד סיריאלי שקל לפספס – שימוש בasync.series נעשה רק כשבאמת רוצים שהקוד יהיה סיריאלי. אז ב-async series זה בעצם מתודה שמבטיחה לנו שכל ה-callbacks של ה-task הראשון יסתיימו לפני שה-task השני יתחיל , אבל כאן אין באמת צורך בכך. לכן כדאי לשם לב מתי משתמשים ב-async series .

**שקף 16:**

אני רוצה להראות פיתרון אם אתם לא רוצים לעטוף את הקוד שלכם ב-promise , ברוב המקרים אני מאוד ממליצה כן לעטוף את הקוד ב-promises כי זה הופך אותו להרבה יותר קריא. אבל אם אתם לא יכולים לעשות את זה , אז יש את ספריית async ובה יש את async parallel שיתן לכם אפשרות להריץ את הפעולות הא-סינכרוניות ב-parallel .

**שקף 17:**

עוד דרך ליצור קוד סיריאלי שאנחנו מאוד רגילים אליו ומאוד רגילים לכתוב ככה הן nested promises . אז כאן אנחנו מפעילים את ה-promise הראשון שיעשה execute לשאילתה של ה-get users ויחזיר תוצאה, ולאחר מכן מחזירים את ה-promise של ה-get categories ושיפעיל את השאילתה ויתן תוצאה. השאילתות לא תלויות בניהן , אפשר לחלוטין להריץ אותן ב-parallel , אבל כאן הן ירוצו סיריאלית ואנחנו נאבד מהכוח של node להריץ דברים ב-parallel שלא לצורך.

**שקף 18:**

אז הפתרון לזה הוא להריץ promises ב-parallel מתי שאתם יכולים. אם יש לכם קריאות שלא תלויות אחת בשניה, אז אתם יכולים להשתמש ב-promise all . Node יעשה schedule לקריאות אחת אחרי השניה אבל הן כולן יגיעו ל-worker pool והThreads של ה-worker pool יריצו אותם בו זמנית (ואם ה-cpu שלכם הוא multi core זה באמת ירוץ במקביל) . וככה תקבלו ביצועים יותר טובים.

**שקף 19:**

עכשיו נסתכל על async await . הרבה מאוד מהקוד החדש מ2017 נכתב עם async await . אז זה בעצם קוד אסינכרוני שנראה כמו קוד סינכרוני. אבל שימו לב שממש ממש קל לכתוב קןד סיראילי ככה. גם כאן אין שום סיבה להריץ את getCategories אחרי getUsers – אפשר לחלוטין לעשות scheduling במקביל. הקוד הזה יתן לכם performance הרבה פחות טוב.

**שקף 20**

אז בואו נראה דוגמה לקוד א-סינכרוני שרץ ב-parallel עם async await , הדרך לעשות את זה היא עם await ל-promise all , שמאפשר לנו לעשות schedule ל-I/O אחד אחרי השני – כי שוב – אין באמת קוד שרץ במקביל ב-Node , מה שיש זה parallel על הפעולות הא-סינכרוניות בעזרת ה-worker pool , כאשר ה-scheduling שלהן הוא אחד אחרי השני אבל ההרצה שלהן היא מקבילית.

זו דוגמא מאוד חשובה כי יותר ויותר קוד עכשיו משתמש ב-async await .

**שקף 21:**

אז כמה נקודות חשובות כשאתם רוצים לשפר Performance של אפליקציה :

תגתכלו על המבנה של הקוד ותחפשו דברים סיריאליים. מאוד קל לנו לכתוב ככה. תנסו לשנות את המבנה של הקוד שלכם כדי לכתוב כמה שיותר קוד מקבילי.

באמת, באופן שוטף, תנסו להסתכל על הקוד וכל הזמן לראות מה אפשר להריץ במקביל.

**שקף 22:**

בואו עכשיו ננסה לדבר על איך אנחנו מטפלים ביותר load בשרתים שלנו.

**שקף 23:**

אז אני רוצה להזכיר שוב , Node.js הוא single process עם event loop , ה-process של Node.js ימופה ל-CPU אחד. עכשיו – יש גבול לכמות ה-load שprocess אחד יכול לטפל ובה, ולא משנה כמה המכונה שלנו חזקה. אז כדי לטפל ביותר load אנחנו נהיה חייבים להשתמש ביותר processes .

**שקף 24:**

<https://medium.com/@fermads/node-js-process-load-balancing-comparing-cluster-iptables-and-nginx-6746aaf38272>

אז אני רוצה שנדבר על כמה setups אפשריים.

ה-setup הראשון ל-cluster שלנו יהיה עבודה עם ה-cluster module של Node.js . אז יהיה לנו master process שיהיה אחראי על spawn ל-processes לפי מספר ה-CPUs , יהיה אחראי על respawn במידה ואחד ה-processes מת, ויהיה אחראי על ה- load balancing ל-requests .

**שקף 25:**

ה-setup השני שלנו הוא עבודה עם IPTables עבור ה-routings של ה-requests , כאשר יהיו לנו מספר docker containers כמספר ה-CPUs שיש לנו על המכונה, כל docker container יכיל Instnace של השרת שלנו וימופה ל-CPU אחר כאשר הוא גם יאזין ל-port אחר , ונכתוב rule ב-iptables שינווט בקשות שמגיעות ל-Port 80 לאחד מה-Ports שה-containers מאזינים להם.

**שקף 26:**

ה-setup השלישי שלנו הוא עבודה עם load balancer עבור ה-routings של ה-requests , כאשר יהיו לנו מספר docker containers כמספר ה-CPUs שיש לנו על המכונה, כל docker container יכיל Instnace של השרת שלנו וימופה ל-CPU אחר כאשר הוא גם יאזין ל-port אחר , ונשתמש ב-HAProxy או Nginx כload balancer , על מנת לנווט בקשות שיגיעו לשרת , לאחד מה-Ports שה-containers מאזינים להם.

**שקף 27:**

אז בואו נסתכל על מספר הבקשות שיש לנו סה"כ בשרת :

אז סה"כ , בנשך 5 דקות , אנחנו רואים שטיפלנו בפחות load כאשר השתמשנו בקונפיגורציה של ה-cluster module . ניתן לראות שבקונםיגורציה של ה-cluster module טיפלנו בכ-10,000 requests פחות ממה שטיפלנו ב-setups האחרים – למרות שכולם מריצים אותו מספר processes .

**שקף 29+28:**

בואו רק נסתכל על מספר הבקשות שprocess טיפל בו – אז אני מראה כאן רק את ההשוואה ל-ngnix , אבל תראו שכל worker ב-nginx טיפל בהרבה יותר בקשות ממה שטיפל worker ב-cluster module . אפיךו ה-process הכי לא יעיל ב-nginx טיפל בהרבה יותר בקשות ממה שה-process הכי יעיל ב-cluster module טיפל בהם. הגרף מעט מטעה – אז תסתכלו על המספרים.

**שקף 30:**

אז למה ה-cluster module מטפל בפחות load מ-load balancer ?

בשימוש ב-luster module , ה-master process אחראי על load balancing של ה-requests . זה לא בא בלי מחיר :

קודם כל – בדרך כלל רואים שהוא משתמש בפי 3 יותר זיכרון מה-processes האחרים.

בנוסף, ה-master process בדרך כלל גם יטפל בבקשות וגם יעשה routing ל-requests ל-workers השונים. אז אנחנו נבזבז את ה-CPU שלנו על routing כשמה שצריך לזכור זה שהשרת שלנו לא עושה routing בצורה אופטימלית – רחוק מזה . ה-efficiency של ה-routing של הmaster process שלנו לא יכולה להתחרות ב-efficiency של load balancer .

בנוסף מדובר כאן על processes שונים ולכן יש לנו IPC באמצע – כלומר שגם כאן אנחנו נסרלז אובייקטים ונעביר אוצם בין ה-processes כמו שנעשה ב-load balancer – אז כאן אין יתרון.

**שקף 31:**

מסקנות :

אם רוצים ממש לסחוט את הביצועים ולקבל כמה שיותר load , אז כדאי ללכת על load balancer . לעומת זאת , ה-cluster module מהווה פיתרון נוח , out of the box , לטפל ביותר load , שלא מצריך מאיצנו אינטגרציה עם קומפוננטה נוספת.

**שקף 32**

אז עכשיו בואו נדבר קצת על performance של promises .

**שקף 33:**

אז promises ממש נהיו best practice לפיתוח ב-node . הם נהיו קונבנציה מאוד חזקה. יש המלצה גורפת לכתוב את ה-api שלנו ב-node עם promises .

ישנן הרבה ספריות promises חיצוניות ומאז ES6 הן גם native בשפה.

אז בואו נבין מה ה-Overhead של ה-performance שה-promises נותנים על callbacks , ובנוסף נבין באיזה ספריית promise כדאי להשתמש.

**שקף 34:**

אז יש כאן דמו קטן. אנחנו נעשה Resolve ל-10,000 promises של bluebird לעומת 10,000 promises של native .

נריץ מה-command line . נראה שהקוד ממש ממש זהה.

שימו לב שה-native promises רצות כמעט פי 2 יותר לאט מה-Bluebird Promises

**שקף 35:**

אז למה זה קורה

אני לא אכנס כאן לעומק למימוש הפנימי אבל הן מקצות מקום באופן הרבה יותר יעיל, מתשתמשות בהרבה מאוד זיכרון, ובאופן כללי, הן הרבה הרבה יותר optimized .

לכן המסקנה היא תמיד להשתמש ב-bluebird promises – לפחות כרגע.

**שקף 36:**

<https://kyrylkov.com/2017/04/25/native-promises-async-functions-nodejs-8-performance/>

אז בואו נראה תוצאות של טסט שנעשו , מי שרוצה אני אשלח לכם את הלינק לטסט הזה בפרטי.

בואו ננסה לנתח את מה שאנחנו רואים כאן :

אז ישן כאן השוואה של אותו קוד שנכתב עם bare callbacks , לעומת כתיבת אותו סנריו עם:

bluebird promises

native promises

async await + bluebird promisify

async await – util promisify .

קודם כל אנחנו רואים בבירור שיש overhead ל-promises על ה-bare callbacks בלי עטיפות של promises בכלל.

אז אנחנו רואים ש-native ES6 promises הן פי 2 יותר איטיות ומשתמשות ב-50% יותק זיכרון מ-bluebird . אנחנו רואים שגם async await מוסיף overhead על promises .

אנחנו גם רואים שEs7 async functions יחד עם native promisify עובדות רק ב-30% יותר לאט מ-bluebird promisify + async await .

**שקף 37:**

אז בוא נסכם כמה מסקנות :

קודם כל – promises ו-async await מוסיפים overhead שאנחנו רואים בבירור. עדיין , ההמלצה הגורפת היא לעבוד עם promises ובמיוחד עם async await – כל הקוד החדש נכתב ככה, המון apis חדשים נכתבים ככה וכו.

כ-best practice – תמיד להשתמש ב-bluebird promises .הכי יעיל והכי optimized .

עכשיו – מילה שלי לסיום, היתי במצבים שבהם אני צריכה להילחם על ה-milliseconds . במצבים כאלה אפשר לשקול לכתוב callbacks ולהוריד את ה-promises .

**שקף 38:**

אז בואו עכשיו קצת נדבר על code optimizations ב-javaScript .

<http://voidcanvas.com/javascript-performant-coding-tips/>

**שקף 39:**

כלל אצבע חשוב – תמיד להעדיף v8 native libraries על ספריות חיצוניו כמו loadash וJQuery . הסיבה העיקרית היא שהן צכילות קוד שנועד לתמיכה ב-multiple browsers שפשוט לא צריך אותו ב-server programming , והקוד הזה יכול מאוד להזיק ל-performance שלכם.

היוצא דופן הן bluebird promises שדיברנו עליהן , שנותנות ביצועים הרבה יותר טובים מ-native promises .

**שקף 40:**

אז בואו נדבר על הדרך הטובה ביותר לעשות iterations על arrays .

אז כאשר עושים איטרציות על arrays – האם אנחנו צריכים להשתמש ב-for loop או ב-foreach . צריך להעדיף foreach בעיקר בגלל שהיא נמנעת מלעבור על indexes אם אלמנט הוא Undefined .

**שקף 41:**

אז בואו נסתכל כאן על דמו קטן. יש כאן מערך בגודל 100 כאשר יש ערך רק באינדקס 0 ובאינדקס 434. בואו נריץ את הדמו. תראו שלולאת ה-for מדפיסה את הערך 1000 פעמים לconsole , את ההדפסה I am for , ולולאת ה-foreach מדפיסה רק פעמיים I am foreach – רק עבור הערכים שיש להם ערך. אז אני חושבת שהחיסכון כאן ב-performance הוא ברור.

**שקף 42:**

בואו נדבר על הJIT compiler . v8 משתמש בו כדי לעשות אופטימיזציה ל-hot code . hot code היא פונקציה או אובייקט שמשתמשים בו באופן שוטף. V8 מאחסן את ה-compiled binary version של הפונקציה או האובייקט אם החתימה שלהם לא משתנה. זה נותן בןסט גדול לביצועים, כי כאשר אתה שם קוד של פונקציה ולא משנה את ה-contract , ה-JIT compiler v8 יקמפל אותה ויעשה לה אופטימיזציה .

**שקף 43:**

עוד אופטימיזציה של ה-JIT compiler – כאשר אתם יוצרים אובייקטים עם אותה חתימה כדאי לעשות contstructor של class בשבילם.

בצורה הזאת אתם יכטלים לעשות single hidden class עבור כל ה-Instances שיש לכם מה שיתן ל-JIT compiler את ההזדמנות לעשות לכם אופטימיזציה לקוד.

**שקף 44:**

משתמשים המון ב-delete כדי למחוק properties ן-variables . אז זו שגיאה קשה , כי מה שה-JIT compiler עושה זה משנה את ה-Hidden class של האובייקט , זורק את ה-compiled code של האובייקט ועושה לו invalidate , ו-v8 מתייחס לאובייקט בנפרד.

**שקף 45:**

קצת הקדמה למערכים ב-JavaScript :

אז מערך צריך להיו Bulk של זיכרון רציף , שנועד לאחסן בתוכו values .

יש כאן ייצוג בתמונה של representation של array , עכשיו בואו נניח שב-array יש 4 אלמנטים וזה array של int , אז int הוא בגודל של 4 byte ולכן ב-array יהיו 16 byte רציפים.

זה מבנה נתונים מאוד יעיל.

ב-JavaScript , array הוא hashmap . שזה מצויין שרוצים לעבוד עם key value אבל כשרוצים לפנות לערך מספר 1 ואז לערך מספר 2 – זה נותן ביצועים הרבה פחות טובים.

**שקף 46:**

אז ב-ES6 הוצגו types חדשים של מערכים – הוצגו arrayBuffer ו-typedArrays . כאשר משתמשים בהם , הJIT קומפיילר מקצה זיכרון רציף למערך , וכך הקריאה והכתיבה מהמערך הן הרבה יותר יעילות. אנחנו נראה דוגמת קוד עוד מעט למערכים החדשים שהוצגו ב-ES6 , אבל הם יכולים לתת ביצועים יוצר טובים.

**שקף 47:**

אז בואו נראה קוד של הכנסה למערכים ונראה את ההבדלים בין המערכים הישנים למערכים החדשים ב-javaScript .

אנחנו נעשה לולאה של הכנסה של מיליון אלמנטים למערך. בדוגמה הראשונה נכניס את הערכים למערך מהType החדש . מגדירים אותו כ-arrayBuffer ועחיו בונים את ה-typed array . במקרה הזה יוקצה לנו זיכרון רציף.

בדוגמה השניה זה אותו סנריו, רק מערך מהסוג הישן, ואני אכניס איבר מtype אחר כדי שה-compiler יתיחס למערך הזה כ-hash .

בדוגמה השלישית והרביעית , יש קריאה של הערכים , כאשר בדוגמה הרביעית נכניס בכוונה ערך שהוא מ-type אחר , כדי שנתייחס למערך כ-hash , וניקח סנריו של עבודה לפי אינדקסים.

אז אנחנו רואים שהכתיבה הרבה יותר מהירה במערכים מהסוג החדש, ולעומת זאת הקריאה יותר מהירה במערכים החדשים אבל לא בהרבה.