**שקף 3**

אז בתיאוריה – לא אמורות להיות לנו בעיות concurrency . ב-node.js ה-event loop עובד כ-single threaded ולכן , אם יש לנו thread אחד אז אנחנו יכולים לא לדאוג – תמיד יהיה לנו ערך קונסיסטנטי בכל המשתנים שלנו , אין לנו Interrupts הכל צריך להיות חלק.

**שקף 4**

אז באמת בתיאוריה זה ככה , ויש לנו event loop אחד , והcritical section שלך רץ לבדו כאשר אין עוד events ששייכים ל-requests אחרים .

בפועל – ב-event loop רצים גם events ששייכים ל-Requests אחרים, ובכל פעם שיש פעולה א-סינכרונית, אז יתחילו לרוץ events אחרים ב-event loop , שחלקם שייכים ל-requests אחרים . ובמידה ויש לך פעולה א-סינכרונית באמצע critical section אז events מ-requests אחרים יתחילו לרוץ, ובמידה והם ישנו אובייקטים שאותם צריך עבור הקוד של ה-critical section בין multiple Requests אתם תהיו בצרות.

**שקף 5**

אז יש פה דיאגרמה שמנסה להסביר את הבעיה. יש שני Requests ב-event loop , ושתיהן רצות במקביל. במידה והן עובדות על אותו ערך, אנחנו לא יכולים להבטיח שאכן האובייקטים אכן יהיו קונסיסטנטיים. כל פעולה אסינכרונית תגרום ל-events של requests אחר לרוץ, ותגרום ל-flow שלו להיות resumed .

**שקף 6**

אז איך יכולות להיות לנו פעולות א-סינכרוניות ?

אם למשל אנחנו משתמשים בפונקציות שעובדות עם Timers במימוש הפנימי שלהם – יש פה דוגמאות , setInterval, settimeout – כל timer הוא שלב אחר ב-event loop . ןיגרום ל-events של Requests אחרים לרוץ.

אם במימוש הפנימי של הפונקציות שאנחנו משתמשים בהן יש פעולות IO שאנחנו לא מודעים אליהן. אם אנחנו משתמשים ב-Node.js module שבו יש פעולה א סינכרונית.

בשורה התחתונה – פעולות א-סינכרוניות יכולות להגיע מכל מיני מקומות ואנחנו אף פעם לא יכולים לשלוט במימוש הפנימי של הפונקציות שאנחנו משתמשים בהן.

**שקף 7:**

אז באופן כללי – בואו נדבר על אסטרטגיות כלליות כי להתמודד עם בעיות concurrency :

קודם כל, ישנן patterns מוכרים של נעילות – locking , async lock וכו . צריך להתחיל לממש critical section ב-node.js

בנוסף אפשר לקחת patterns אחרים שמעבירים את הבעיה למערכת אחרת – optimistic locking , טרנזקציות במידה ונתמכות וכו'.

**שקף 8**

אז בואו נתחיל לצלו לסנריואים ספציפיים שנתקלתי בהם במציאות . בואו נדבר על lazy initialization לדוגמא.

**שקף 9**

Lazy initialization היא בעצם טקטיקה של לעשות Delay ל-Execution של פעולה יקרה, לאתחל את הערך שלה פעם אחת , ולדאוג לזה שלא תתבצע מספר פעמים. משתמשים בזה כדי לדאוג שיתבצע למשל creation של אובייקט פעם אחת, חישוב של ערך או פעולה יקרה אחרת מכל סוג שהוא.

זה בדרך כלל מושג על ידי הפעלה של getter כדי לבדוק האם הערך כבר אותחל בprivate member שמתנהג כ-cache . אם הוא כבר אותחל הוא מוחזר במיידי, ואם לא נוצר Instance חדש או שהפעולה מתבצעת והערך מתאתחל.

**שקף 10 :**

כמה דוגמאות שימוש לpattern הזה בשפות קונבנציונליות :

* יש לנו את ה-singleton pattern – או anti pattern . סינגלטון זה אובייקט שצריך להיות Instance אחד שלו ואסור שיווצר יותר מinstance אחד. אני מניחה שכולם כאן מכירים את ה-pattern הזה ויצא לכם להשתמש בו מדי פעם. אני השתמשתי בו לדוגמא במערכת שהיתה monolite גדול ואתחלתי configuration manager שהיה צריך להחזיק ולהחליט איזה project context יווצר , ובתור manager היה חשוב שיווצר instance אחד שלו.
* הדבר הבא הוא factory method , כאשר ב-factory אנחנו מאתחלים map עם אובייקטים בהתחלה ומחזירים את אותו אובייקט בכל פעם .
* אפשר גם להשתמש בזה ב-caching layers , כאשר אנחנו צריכים להחזיר את אותו אובייקט עבור key מסויים, וחשוב לנו שלא יהיה initialization כפול לאותו אובייקט.
* בנוסף עבור immutable objects , בדרך כלל מאתחלים אובייקט שרוצים שאף אחד לא יוכל לשנות אותו. מאתחלים אותו ב-map ומוציאים אותו מה-map בעזרת get . גם שם אנחנו רוצים לאתחל את האובייקט פעם אחת.

ישנם עוד הרבה שימושים ל-pattern הזה שאפשר להזכיר.

**שקף 11:**

אז יש כאן דוגמא לקטע קוד של java singleton , שבו אנחנו רואים אתחול של אובייקט בפעם הראשונה שמנסים לקרוא לו. יש לנו private constructor שנקרא מה-get אם האובייקט ריק. אז זה שמוש ב-lazy initialization שאני מקווה שכולם מכירים והוא מאוד ברור.

**שקף 12:**

אז הקוד הזה הוא לא thread safe , כי אם יש כאן כמה threads , אז יכול להיות שthread אחד בדיוק באמצע יצירת האובייקט וThread אחר בודק האם האובייקט שווה ל-null , ואז הוא לא שווה ל-null והאובייקט יווצר פעמיים. יש פה דוגמא ל-pattern של double check locking שמהפתרונות המוצעים הוא הפיתרון הכי פחות טוב לבעיה . ישנם גם פתרונות של לשים synchronize על המתודה, ואז הlock הוא ברמת המתודה, ישנו פתרון של static initialization שמתבצע כשהclass עולה בפעם הראשונה ועוד.

**שקף 13:**

אז בואו נדבר על use case ספציפי ל-lazy initialization ו-concurrency . מימשתי caching layer ב-node , כאשר עבור key מסויים היתי צריכה לאחסן תוצאות של אובייקטים שחזרו מה-DB . המנגנון היה היה בשרת כאשר כמובן יש המון concurrent requests שמגיעות לשרת. מה שהיה חשוב זה שאף תוצאה לא תאוחסן פעמיים ב-cache . בנוסף, כאשר consumers מקבלים את האובייקטים – אנחנו חייבים לדאוג שהם לא יוכלו לשנות אותם.

**שקף 14 + 15:**

אני רוצה להסביר לעומק למה נוצרה בעיית concurrency כאן. אז בוא נגיד ששני Requests שונים מריצים את אותה שאילתה בdatabase , ורוצים לאתחל את התוצאה ב-cache . עבור ה-request הראשון יתחיל ה-initialization ונעבור את ה-check של value == null . לאחר הבדיקה הזו תהיה לנו פעולה א-סינכרונית שתגרום ל-context switch ב-event loop ולכך שה-flow של ה-initialization של ה-cache של ה-request השני יתחיל.

אז גם ב-flow השני נעבור את הבדיקה של value == null ונתחיל את ה-initialization של ה-cache , ואז ניתקל באותו async operation שיגרום ל-flow של ה-query הראשון לחזור מאותה נקודה שבה הוא הפסיק , ולבצע את האיתחול עבור ה-key הספציפי, ולאחר מכן ה-flow השני ימשיך ונאתחל את הערך שוב. ואז קיבלנו את הבעיה של האתחול פעמיים.

**שקף 16:**

הפתרון הראשון – שהוא פתרון לא טוב – הוא לעשות את הקוד באותה נקודה סינכרוני שלא יהיו בו פעולות א-סינכרוניות. זה לא פיתרון טוב מפני שקוד סינכרוני יכול לחסום לנו את –event loop ולגרום לביצועים גרועים , ובנוסף , אנחנו כל הזמן משתמשים בקוד שלא אנחנו כתבנו , ואנחנו פשוט לא יכולים לשלוט על מה שיש בתוך הקוד הזה. מספיק שיש שם איזה setInterval או משהו כזה וזהו – אנחנו מקבלים בעיית concurrency .

**שקף 17:**

אז הפתרון הראשון הוא לממש lock , ואני רוצה להסביר איך אפשר באופן יחסית פשוט לממש lock עם promise .

אז איך ממשמשים כזה דבר עם promises ?

אז בקריאה הראשונה – מה שצריך לעשות זה לאחסן את ה-promise ב-memory container שלנו לפי ה-key – וכך אנחנו מבטיחים שמי שניגש לאותו key יקבל את אותו instance של promise .

עכשיו מכיוון שה-promise הוא בסטטוס של pending אז עד שהוא לא יהיה Resolved בעצם אין promise . ןלכן Requests אחרים שמגיעים יקבלו את אותו promise ויחכו עד שהוא יהיה Resolved .

ב-then callback נבצע את האיתחול הספציפי ונשים את הערך ב-container .

**שקף 18**

אז פה יש דוגמת קוד של ה-get – אם הערך הוא promise אז מאתחלים את ה-container ומחזירים את הערך , ואם הערך הוא לא promise אז מחזירים את הערך. הערך לא יהיה promise אחרי ה-first initialization .

**שקף 19:**

אז בואו נדבר על immutability של אובייקטים.

**שקף 20:**

אז immutability הוא pattern גנרי ואפשר לעשות לו reuse גם ל-concurrency . בעיקרון , תחשבו שב-Node.js ישנן הרבה פונקציות א-סינכרוניות ששיכות ל-events שונים שרצים, כאשר ה-events שייכים למספר בקשות שרצות במקביל. יכול להיות שמספר בקשות יעבדו על אותן אובייקט במקביל – ואז אנחנו נקבל ערך לא קונסיסטנטי . כי מספר פונקציות א-סינכרוניות ישנו את הערך , flows של requests שונים מפסיקים וממשיכים באופן לא צפוי וכו'. במידה וכל ה-consumers שלכם יקבלו clone , הם יוכלו לשנות את הערך כמה שהם רוצים – הערך המקורי נשאר כמו שהוא ולא משתנה.

חשוב – לעשות deep clone ולא shallow clone .

**שקף 21**

שיטות על קצה המזלג לעשות deep clone לאובייקטים : json parse + json stringify , loadash מספק deep clone ועוד.

**שקף 22**

בואו נדבר על נעילה של אובייקט כאשר מספר פונקציות א-סינכרוניות משתמשות בו.

**שקף 23:**

אני אתן לכם סנריו שהיה לי במערכת שעבדתי עליה.

אז יש לי Entity שיש לו property שנקרא interest rate – שער ריבית .

בנוסף יש לו פונקציה שאם מפעילים אותה היא מפעילה חישוב , החישוב הוא א-סינכרוני, כאשר אחד מהמשתנים שמשתתפים בחישוב הוא שער הריבית.

עכשיו – אם אנחנו ניתן לעדכן את שער הריבית באמצע החישוב – מבחינה עסקית ניצור מצב לא תקין. כי אם הורדנו אותו אז נוכל לקבל תביעות של לקוחות – למה לא השתמשנו בשער החדש בחישוב . מצד שני – יכולים להגיע requests אחרים לשרת שיעדכנו את שער הריבית – ומכיוון שהחישוב הוא א-סינכרוני אז יתכן וה-event loop יבצע אותם לפני שהחישוב יסתיים.

**שקף 24:**

בואו נסתכל על ציר הזמן. אז בנקודה מסויימת מתחיל חישוב הריבית, ובאמצע מגיע ה-Request כדי לעדכן את שער הריבית. במקרה כזה אנחנו נהיה בבעיה. צריך למצוא דרך שהעדכון לשער הריבית לא יתבצע בזמן שבו מתבצע החישוב.

**שקף 25:**

אז אם ה- DB שלכם תומך בטרנזקציות אתם יכולים להשתמש במנגנון הזה. אבל המון DBs לא תומכים בטרנזקציות.

אז הפתרון כאן יורכב משני שלבים –

אם אנחנו מתעסקים ב-single Node - נעילה של האובייקט בזיכרון ויצירת critical section סביבו שמיוחסת לגרסה הנוכחית שלו – שבמסגרת זו בקשות אחרות שרוצות לעדכן את אותו version מחכות. ב-single node הפתרון הזה יספיק.

אם אנחנו רוצים להרחיב את הפתרון ל-multi Node – אז Request שיגיע לnode אחר יכול לעדכן את הערך – כאן כדאי לשלב את זה עם פתרון routing – לממש sticky session שבעזרתה requests שעובדים על אותו אובייקט תמיד יגיעו לאותו Node ב-cluster .

**שקף 26:**

אז קודם כל אני ממליצה ליצור verion של האובייקט ברמת האפליקציה ולעדכן את ה-version בכל עדכון של האובייקט. כך אם שתי resuests מנסות לעדכן את אותו אובייקט שתיהן יעבדו על אותו version .

הנעילה של האובייקט בזיכרון נעשית על ידי ה-version number . אז ב-Node נעילה אומרת ש-events שמשוייכים ל-key הזה יחכו ב-event loop ואחרים ימשיכו לרוץ באופן רגיל.

**שקף 27:**

אז ספרייה שאפשר לעבוד איתה ונותנת לנו Critical section ב-Node.js היא הספרייה async lock . מה שאני רוצה להדגיש כאן הןא שחשוב לנעול את שני הצדדים –

גם את החישוב הא-סינכרוני עצמו

וגם את העדכון של האובייקט.

חשוב מאוד לנעול את שניהם על אותו key וה-version number הוא מושלם לזה.

כאן יש דוגמת קוד לשימוש בספרייה async lock – כאשר נועלים פונקציה לפי key .

**שקף 28:**

אז אם רוצים לעשות Extend לפתרון ל-cluster , יהיה לנו מצב שעדכון יגיע ל-Node אחר ואז אין לנו הגנה של locks . נהיה חייבים לטפל בזה ברמת ה-routing – ליצור sticky session ולכוון את כל הבקשות ששייכות לאובייקט הספציפי הזה לאותו מקום ב-cluster .

**שקף 29:**

אז בואו נדבר על עדכון של שני Requests במקביל .

**שקף 30:**

אז מה הבעיה שנוצרת כאן בעצם :

שני Requests רצים במקביל בשרת. הבקשה הראשונה מתחילה את ה-update flow . אז יש פעולה א-סינכרונית והflow של הבקשה השניה מתחיל , אותה פעולה א-סינכרונית מופיעה ואנחנו חוזרים שוב ל-flow של ה-request הראשון שהוא resumed . הבקשה הראשונה מבצעת את ה-update ואז הבקשה השניה מבצעת את ה-Update . אנחנו מגיעים למצב ששני ה-updates דורסים אחד את השני.

**שקף 31 + 32:**

אז optimistic locking זו אסטרטגיה לעדכון רשומות שמבטיחה consistency . באסטרטגיה זו אנחנו מתחזקים גרסה על האובייקט ומעדכנים את הגרסה בכל Update לאובייקט , וכל consumer שלנו יודע בדיוק על איזו גרסה הוא עובד.כדי להשלים את המימוש אנחנו מבצעים שאליתה עם הDB שלנו שתראה בצורה של update \_\_ where version = my version . במידה ומישהו כבר עדכן לפני , ולאובייקט יש גרסה אחרת בDB העדכון יכשל. יש פה דיאגרמה שמראה מה יקרה כאשר שני consumers ינסו לעדכן את אותו אובייקט ומראה שאחד מהם יכשל.

**שקף 33:**

אז היתרונות של הפתרון הוא שאנחנו מקבלים תמיכה ב-cluster במיידי ושהפתרון הוא consistent

החיסרון הוא ש-requests יכשלו וה-consumer יצטרך לטפל בכישלון.