**שקף 3**

בואו ניזכר בגישה של ה-Non blocking IO . אז אם ניזכר במה שהיה לנו בתחילת היום, בשפות כמו Java ו-python , שהן bloking IO , אז במידה וניח מריצים שאילתה ל-DB , שורת הקוד הבאה תתבצע רק כשה-DB query תחזור. בשביל זה המון פעמים פותחים הרבה Threads , ואז התוכנה שלנו תקצה משאבים ל thread scheduling ו-context switching במקום לבצע את הפעולות שהיא באמת צריכה לבצע – את ה-execution של התוכנה.

אז כאשר ניסו להרי. את ה-event loop של javascript ב-server הפרדיגמה השתנתה ןה-event loop של JavaScript התפתח להיות ה-event loop של Node כמו שאנחנו מכירים אותו היום.

אז עם Node , IO functions הן Non Blocking – הן ממשיכות ב-Background באופן א-סינכרוני , והתוכנית תעבור לבצע את שורת הקוד הבאה בלי לחכות ל-IO operation שתסתיים. ב-JavaScript זה מושג על ידי העברת CallBack function והפיכת הפונקציות ל-first calss citizens בשפה.

אז כל זה מאוד מגניב ונחמד, אבל בסופו של דבר זה מאוד מאוד לא קריא. אז יחד עם הגישה הזו – נולד לו קטע הקוד הבא שהוא מאוד מפורסם – שנקרא the pyramid of doom , בגדול אנחנו כאן פותחים כמה menus ומדפיסים את המילה Doom , אבל זה סופר לא קריא ורמת ה-Nesting פה היא ממש מטורפת.

אז זה בעצם הפך קוד Node לקוד שמאוד מאוד קשה לתחזק, וגם היה ממש קשה לשלןט על הcontrol flow .

**שקף 4 :**

אז הפיתרון הראשון שאומץ היה promises , מה שהוריד את רמות ה-nesting ונתן שליטה על ה-control flow . promises מייצגות placeholder לפעולה א-סינכרונית או לבלוק א-סינכרוני, ונותנת callback להצלחה או כישלון שלו. תזכורת של –promises ישנם שני מצבים – fulfilled או Rejected , וכאשר ה-promise משתערך אז מופעל ה-callback על ה-event המתאים.

אז בעזרת promises אנחנו יכולים להוריד מאוד את רמות ה-nesting של הקוד שלנו ולממש Control flows שאנחנו צריכים להריץ כמו parallel לדוגמא שימומש עם promise.all .

**שקף 5:**

הפתרון הבא שהוצג היה generators – שמייצגים פונקציה שאפשר לעצור את ה-execution שלה ולהמשיך אותו בשלב יותר מאוחר, כאשר ה-execution של הפונקצייה נעצר מבפנים וcontrol flow חיצוני ממשיך אותו. Generators גם מאוד מאוד שיפרו את ה-error handling ונתנו לנו להשתמש במנגנונים שאנחנו רגילים ונוח לנו לעבוד איתם כמו Try catch ומאוד מאוד פישטו את הטיפול בשגיאות. בדוגמא שלנו אנחנו רואים קוד שנראה כמעט שטוח על יהיה שימוש ב-generators כאשר כל next עושה resolve ל-promise .

**שקף 6:**

אז EC7 בעצם הציג את הפיצ'ר הכי מהפכני – זה הפיצ'ר שהכי שינה את התכנות ב-javacsript עד היום – שמוכר בשם Async / Await . זה feature שברגע שמתכנתים מגלים אותו הם לא מפסיקים להשתמש בו.

**שקף 7:**

אז מה זה async await ? זה בעצם מורכב משני מרכיבים. קודם כל אני רוצה להסביר מהן Async functions . אז Async function מתחילה , ב-decleration של הפונקציה, עם המילה השמורה async .

Async function זה בעצם אומר שהפונקציה מחזירה promise למי שקרא לה. אז בתוך async functions אתם יכולים לעשות כמה דברים מגניבים :

אתם יכולים להשתמש במילה השמורה await כדי לחכות ל-promise אחר. אז ב await משתמשים רק לפני פונקציות שמחזירות promise . אז await יחכה עד שה-promised הוא resolved או עד שהוא יהיה rejected

ניתן גם להקיף את await בTry catch - וסוף סוף לטפל ב-errors בצורה שפויה , await ידאג לכך שהשגיאה תגיע ל-catch block .

מאחורי הקלעים יש כאן קוד א-סינכרוני.

**שקף 8:**

אז בואו קצת נדבר על ה-keyword החדש await ועל המאפיינים שלו – ניתן להשתמש ב-await רק בתוך async functions , משתמשים בו כדי לחכות ל-promise , וחשוב שאפשר להגיד עליו הוא שהוא גורם ל-execution של הפונקציה לעצור – כמו בgenerators – אותו קונספט בעצם.

בעזרת הכלי הזה הקוד ב-javacsript נראה הרבה יותר פשוט והרבה יותר קריא וזו המהפכה ש-async await הביאה.

**שקף 9:**

אז בואו נסתכל על דוגמת קוד וקצת על syntax – שימו לב לפונקצייה main כאן – כאשר ב-declaration שלה יש את המילה aync שמופיעה לפני הקוד. זו איזושהי דוגמת קוד לפעולות על קבצים. אז אנחנו מחכים לפעולת ה-read בעזרת await כאשר פעולת ה-read מחזירה promise , ולאחר מכן מחכים לפעולת הכתיבה לקובץ בעזרת await . לאחר שפעולת ה-write הסתימה אנחנו שוב קוראים את ה-content המעודכן מהקובץ ואנחנו שוב מחכים לפעולת הקריאה שתקרא את ה-Content של הקובץ בעזרת await . שימו לב שהקוד נראה ממש כמו קוד סינכרוני והוא הרבה יותר קריא.

**שקף 10:**

בואו נדבר קצת על ה-Observations שלנו מהקוד הזה. אז קודם כל את העובדה שניתן להשתמש ב-await רק בתוך async function הזכרנו , אבל עוד משהו שחשוב להגיד הוא שאתם לא יכולים להשתמש ב-await ב-top level קוד שלכם – אלא אם כן הוא עובד עם async functions , לכן ב-top level code אתם תהיו חייבים לעשות Resolve ל-promise באופן explicit . שוב – במידה וה-top level code כן תומך ב\_async אתם לא צריכים .

**שקף 11:**

אז בואו נדבר קצת על היתרונות של async await .

**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-סינכרוניות. 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 .