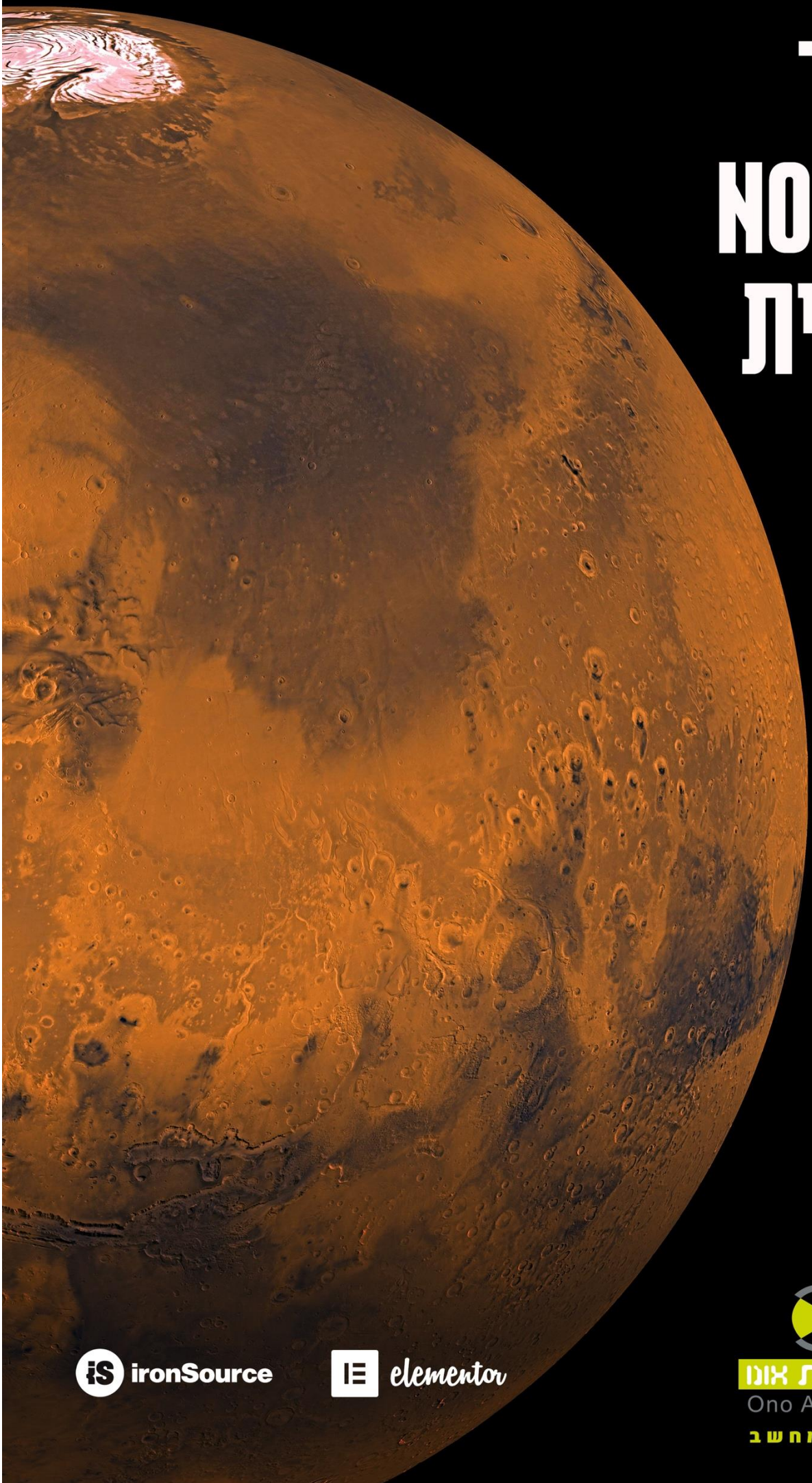
ללמוד NODE.JS בעברית

רן בר-זיק



ironSource
elementor
Ono
הקריה האקדמית אונו
Ono Academic College
החוג למדעי המחשב

ללמוד Node.js בעברית

רן בר-זיק

מהדורה: 1.2.1

עריכה לשונית: יעל ניר

הגהה: חנן קפלן

עיצוב הספר והכריכה: טל סולומון ורדי (tsv.co.il)

הפקה: כריכה – סוכנות לסופרים

www.kricha.co.il

תוכן עניינים

על הספר

הספר "ללמוד Node.js בעברית" מלמד על הפלטפורמה הפופולרית Node.js, המשמשת לפיתוח ג'אווהסקריפט בצד השרת ובסביבת מערכות הפעלה. אפשר למצוא היום Node.js בכל מקום: משרתים של חברות ענק ועד תוכנות תחזוקה ופיתוח שונות. Node.js הפכה בשנים האחרונות לאחת התשתיות החשובות ביותר של הרשת ועולם הפיתוח. גם אנשים המתכנתים על גבי פלטפורמות אחרות ושפות אחרות משתמשים בתוכנות מבוססות Node.js למטרות שונות – בין אם בדיקת הקוד שלהם, הרצת בדיקות או כל משימה אחרת.

לימוד Node.js מחייב הכרה מעמיקה עם שפת ג'אווהסקריפט. בספרי הקודם, "ללמוד ג'אווהסקריפט בעברית", לימדתי ג'אווהסקריפט ברמה המספיקה להתחלת הקריאה בספר זה. הספר מלמד Node.js ומתחיל בבניית סביבת העבודה והתקנת הפלטפורמה. הוא ממשיך בהקניית העקרונות החשובים לפלטפורמה הזו: איך בונים מודול בסיסי בשפה, איך משתמשים במודולים אחרים. אנו מסקרים גם אספקטים מתקדמים החשובים להבנה עמוקה של הפלטפורמה: סטרימים, סוקטים ובניית CLI. בספר יש פרק ארוך ונכבד המלמד על אקספרס, המודול הפופולרי לבניית שרת רשת. אנו לומדים גם על העלאת האפליקציה שלנו לענן באמצעות "הרוקו". בסיומו של כל פרק רלוונטי יש תרגילים והסברים מפורטים הכוללים גם שרטוטים.

הספר מיועד לכל מתכנת ג'אווהסקריפט שמעוניין ללמוד על העולם המופלא של Node.js ולמתכנתים המכירים את Node.js אך זקוקים לחיזוק או לתגבור של הידע שלהם באספקטים מסוימים.

על המונחים בעברית

אני כותב בעברית על טכנולוגיה ותכנות כבר יותר מעשור והדילמה "באילו מונחים בעברית להשתמש" מלווה אותי תמיד. מצד אחד, האקדמיה ללשון העברית מספקת לנו מונחים רבים בעברית. מצד שני, בתעשיית ההייטק, שממנה אני מגיע, איש לא משתמש ברבים מהמונחים האלו. אם תגיעו לראיון עבודה ותגידו: "במפגש המתכנתים האחרון שמעתי על דרך חדשה לבצע הידור שבודק הזחות במנשק מבוסס הבטחות", סביר להניח שלא תקבלו את העבודה. אבל אם תגידו "במיטאפ האחרון שמעתי על דרך חדשה לבצע קִמפול שבודק אינדנטציה ב-API מבוסס

פרומיסים" – יבינו על מה אתם מדברים. זו הסיבה שלא תמצאו מילים כמו "הידור", "מחלקה" או "מרשתת" אלא "קמפול", "קלאס" ו"אינטרנט". המונחים שבהם השתמשתי הם המונחים שבהם משתמשים בתעשייה בפועל. בכל מקום שבו אני משתמש לראשונה במונח בעברית, אני מספק גם את הגרסה שלו באנגלית, כדי שתוכלו להכניס אותו לחיפושים שלכם בגוגל.

חשוב לציין שאיני בז כלל לאקדמיה ללשון ושחלק מהמונחים שלה אכן נכנסו לשפה המדוברת במרכזי הטכנולוגיה השונים (למשל: קובץ או מסד נתונים), אבל בכל מקום שהייתה לי ברירה בין להיות מובן לבין לעמוד בכללי הלשון, העדפתי להיות מובן.

על המחבר

רן בר-זיק הוא מפתח תוכנה משנת 1996 במגוון שפות ופלטפורמות ועובד כמפתח בכיר במרכזי פיתוח של חברות רב-לאומיות, מ-HPE ועד Verizon, שם הוא מפתח בטכניקות מתקדמות הן בצד הלקוח הן בצד השרת, ושם דגש על בניית תשתית פיתוח נכונה, על שימוש ב-CI\CD וכמובן על אבטחת מידע.

נוסף על עבודתו כמפתח במשרה מלאה, רן הוא עיתונאי ב"הארץ" במדור המחשבים, שם הוא מסקר נושאים הקשורים לטכנולוגיה ולאבטחת מידע וכותב על אינטרנט ורשתות.

משנת 2008 מפעיל רן את האתר "אינטרנט ישראל" (internet-israel.com), שהוא אתר טכני המכיל מדריכים, מאמרים והסברים על תכנות בעברית, ומתעדכן לפחות פעם בשבוע.

רן הוא מחבר הספרים "ללמוד ג'אווהסקריפט בעברית", "ללמוד ריאקט בעברית", "ללמוד MySQL בעברית" ו"ללמוד פיתוח ווב מעשי בעברית".

רן נשוי ליעל ואב לארבעה ילדים: עומרי, כפיר, דניאל ומיכל. רץ למרחקים ארוכים וחובב טולקין מושבע.

על העורכים הטכניים

בנג'מין גרינבאום

בנג'מין גרינבאום הוא מתכנת מנוסה, מומחה לג'אווהסקריפט בעל רקע עשיר של עבודה במגוון חברות רב-לאומיות ובמגוון תפקידים ובוגר תואר ראשון למדעי המחשב באוניברסיטה העברית. הוא מפתח בצוות הליבה של Node.js ובמסגרת תפקידו הוא כותב קוד של Node.js ממש, מציע ומצביע על פיצ'רים בשפה ושותף בהחלטות השונות הרלוונטיות ל-Node.js.
בנג'מין היה שותף כעורך טכני לשורה של ספרים מובילים בתחום בנושא ג'אווהסקריפט, כגון You Don't Know JS ו-Exploring ES6 . הוא מרצה בכנסים בארץ ובחו"ל וחבר מוביל בקהילות פיתוח בארץ ובעולם.

גיל פינק

גיל פינק הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert, Microsoft Developer Technologies MVP והמייסד של חברת sparXys.
כיום הוא מייעץ לחברות ולארגונים שונים, שם הוא מסייע בפיתוח פתרונות מבוססי אינטרנט ו-SPAs. הוא עורך הרצאות וסדנאות ליחידים ולחברות המעוניינים להתמחות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט (Microsoft Official Course MOC), מחבר משותף של הספר " Pro Single Page Application Development" (Apress) ושותף בארגון הכנס הבינלאומי AngularUP.
לפרטים נוספים על גיל: http://www.gilfink.net

על החברות התומכות

אלמנטור

אלמנטור מפתחת פלטפורמת קוד פתוח לבניית אתרים שמשנה את הדרך בה בונים אתרי אינטרנט בשוק המקצועי. אלמנטור מעניק למעצבים את החופש ליצור עמודי אינטרנט ללא צורך בקוד ולמפתחים את החירות לדחוף את הגבולות, לרענן ולהרחיב את המערכת בצורה קלה ומהירה באמצעות API ידידותי למפתחים, ובכך לחסוך זמן פיתוח ולהיות יעילים ורווחיים.

עם מיליוני אתרים הפעילים על אלמנטור וצמיחה חודשית מדהימה, התגבשה סביב הפלטפורמה קהילה חזקה המונה מאות אלפי חברים, מפתחים, משווקים ומעצבים, המקיימים מיטאפים בכל רחבי העולם. מידי יום האלמנטוריסטים מייצרים וצורכים אלפי שעות של הדרכות, סרטי השראה ובלוגים מעמיקים, ומפתחים תורמים קוד ורעיונות באמצעות GitHub. האקוסיסטם המקצועי של אלמנטור מתפתח ללא הפסקה והוא אוצר המוסיף ומעשיר את היכולות של כל יוצר אינטרנט.

באלמנטור אנחנו משתמשים בטכנולוגיות קוד פתוח מתקדמות לפיתוח כלי אינטרנט חדשניים ומהירים. אם גם אתם רוצים להיות חלק מהטכנולוגיה שמשנה את חווית האינטרנט בעולם ויש לכם את הידע כדי לבנות עולם יפה יותר אנחנו מחפשים אתכם, מעצבי UI&UX, מפתחי Full Stack, מהנדסי Big Data ו DevOps עם מומחיות בניהול Kubernetes על פלטפורמות הענן של GCP & AWS.

ironSource

חברת ironSource, הנחשבת לחברה מובילה בכל הקשר למונטיזציה באפליקציות ולפלטפורמות פרסום בווידאו, וחולשת על יותר ממיליארד וחצי שחקנים מרחבי העולם, הצופים בפרסומות על גבי תשתית החברה. החברה עוזרת למפתחים לקחת את האפליקציות שלהם לשלב הבא, זאת גם בזכות רשת הוידאו העצומה שלה - והמספרים מדברים בעד עצמם, עם יותר מ-80,000 אפליקציות המשתמשות בטכנולוגיות של החברה בכדי לפתח את העסק שלהן.

הקדמה – מה זה ואיך זה התחיל

Node.js הופיעה בשנת 2009. מדובר בסביבת הרצה של ג'אווהסקריפט בסביבת שרת. סביבת ההרצה הזו בנויה כולה על מנוע ההרצה של כרום V8. מדובר במנוע חזק ומהיר מאוד שמשתמשים בו בכרום. ב-Node.js ההרצה היא מחוץ לדפדפן, אך מנוע V8 מאפשר לג'אווהסקריפט לרוץ מהר מאוד ויעיל מאוד. מרכיב נוסף של Node.js היא ספריית libuv, הכתובה ב-C ומאפשרת הרצה של פעולות קלט ופלט במהירות רבה.

מתכנת בשם ריאן דאל רצה לבנות סמן התקדמות של טעינת קובץ. הוא ניסה לעשות זאת בשרתים הקודמים, ובראשם Apache, אך לא הצליח לעשות כן בגלל בעיות ביצועים. הוא החליט לבנות שרת מבוסס על 8V המהיר, עם דרכים פשוטות לבצע קלט ופלט למערכת ההפעלה ועם ג'אווהסקריפט.

בניגוד לסביבות הרצה אחרות של שפות אחרות, שבהן המשתמש נדרש לנהל את התהליכים של המעבד, ב-Node.js הקוד של המשתמש רץ על תהליך אחד של המעבד ואינו חוסם אותו כאשר הוא מחכה לנתונים שמגיעים. תהליכים נוספים מנוהלים אוטומטית דרך ספריית libuv. דרך הפעולה הזו מאפשרת ל-Node.js לעבוד מהר מאוד עם פלט וקלט, כיוון שאם היא מבצעת בקשה כלשהי לשרת אחר, מערכת קבצים או מסד נתונים, התהליך אינו נחסם אלא הבקשה נשלחת ו-Node.js ממשיכה לרוץ. זה מתאפשר בגלל האסינכרוניות המובנה שיש ב-Node.js והופך את סביבת ההרצה הזו לטובה מאוד בקלט ופלט.

דאל הציג את התוצאה בנובמבר 2009 וכמה חודשים לאחר מכן נוצר npm, מאגר המודולים החופשיים של Node.js, שבו יש מודולים שכל מתכנת ב-Node.js יכול להשתמש בהם בקלות.
סביבת ההרצה של Node.js יכולה לרוץ בכל סביבת שרת שהיא, גם בשרת מבוסס על חלונות וגם בשרת מבוסס על לינוקס. זה אומר בעצם, במילים אחרות, שאם אנו רוצים לעבוד עם Node.js אנחנו יכולים לעשות את זה בקלות רבה בלי שום קשר לפלטפורמה שלנו. יש לנו מחשב מבוסס חלונות? מק? לינוקס? אין כל בעיה – Node.js אמורה לעבוד על כולם באופן זהה. לא תמיד זה קורה, אבל זו הכוונה ולמרות שיש הבדלים, רובם מטופלים.

Node.js פופולרית להדהים. בשעת כתיבת ספר זה (יוני 2019), יש יותר ממאה אלף חבילות תוכנה בקוד פתוח שזמינות למשתמשים ב-Node.js לשימושים שונים. שרתים רבים נכתבים

בעולם על Node.js ומשתמשים בתוכנות מבוססות Node.js בכל מקום: מאפליקציות מובייל ועד אפליקציות דסקטופ, כלי עזר לשרתים באמצעות ה-CLI ולשפות אחרות ועוד. Node.js נמצאת בכל מקום.

כיום מי שמוביל את Node.js הוא מוסד ללא כוונת רווח שנקרא OpenJS Foundation – מוסד שבנוי על פי עקרון "הממשל הפתוח" וכל אחד שיש לו מספיק רצון יכול להשתתף בדיונים ולהשפיע על ההתפתחות העתידית של סביבת ההרצה.

Node.js היא לא שפה, השפה היא ג'אווהסקריפט. Node.js היא סביבת הרצה. קל לכל מתכנת או מתכנתת ג'אווהסקריפט לעבוד היטב עם Node.js. ספר זה אינו מלמד ג'אווהסקריפט ואני יוצא מנקודת הנחה שהקוראים מכירים היטב ג'אווהסקריפט ובדגש על ג'אווהסקריפט מודרני ואסינכרוני. אם אינכם מכירים היטב את השפה הזו, אני ממליץ לכם לקרוא את ספרי הקודם, "ללמוד ג'אווהסקריפט בעברית", שיצא בהוצאת הקריה האקדמית אונו. לימוד של הספר הקודם יביא אתכם למצב שתוכלו להבין את הספר הזה היטב.

בספר נלמד Node.js משלב ההתקנה ועד השלב שבו נדע לשלוט בה באופן מושלם. הדבר החשוב ביותר שכדאי לזכור ב-Node.js הוא שעושר הספריות העצום שלה בעצם חוסך המון מזמן הכתיבה. אנו נלמד פה למשל איך מקימים שרת HTTP, אך הסיכוי שתצטרכו לעשות את זה בחיים האמיתיים הוא אפסי, כיוון שהמודול הפופולרי Express משמש את רוב המתכנתים ליצור שרת HTTP. כמו כן תוכלו להשתמש בידע שתלמדו בספר הזה כדי להוסיף למודולים קיימים או לכתוב אפליקציות של ממש או שרתים של ממש שמשתמשים במודולים של Node.js. ברגע שתבינו איך עובדים עם ג'אווהסקריפט על סביבת ההרצה הזו – השמיים הם הגבול. כאמור, משתמשים ב-Node.js בכל מקום. גם במקומות שבהם כותבים בעיקר בשפות תכנות אחרות, כיוון שהכוח של Node.js הוא ביכולת שלה לפעול בכל מקום, גם במשימות תחזוקה וגם במשימות של אבטחת מידע.

דרך הלמידה

דרך הלמידה היא פשוטה ביותר – קריאת הפרק ותרגול של התרגילים שנמצאים בסופו. התרגול הוא קריטי, בגלל זה חשוב מאוד להשקיע זמן בפרק הראשון ולבנות את סביבת העבודה שלכם. ללא בנייה של סביבת העבודה ותרגול – הקריאה לא תהיה אפקטיבית מספיק. ראשית יש להבין את החומר, לקרוא פעם, או פעמיים או שלוש, ואז לבצע את התרגילים. אחרי שהצלחתם לפתור ולהבין את המשימות – נסו לשחק עם הקוד. נסו לפתור אתגר אחר או לשנות מעט את הקוד כדי להבין מה הוא עושה.

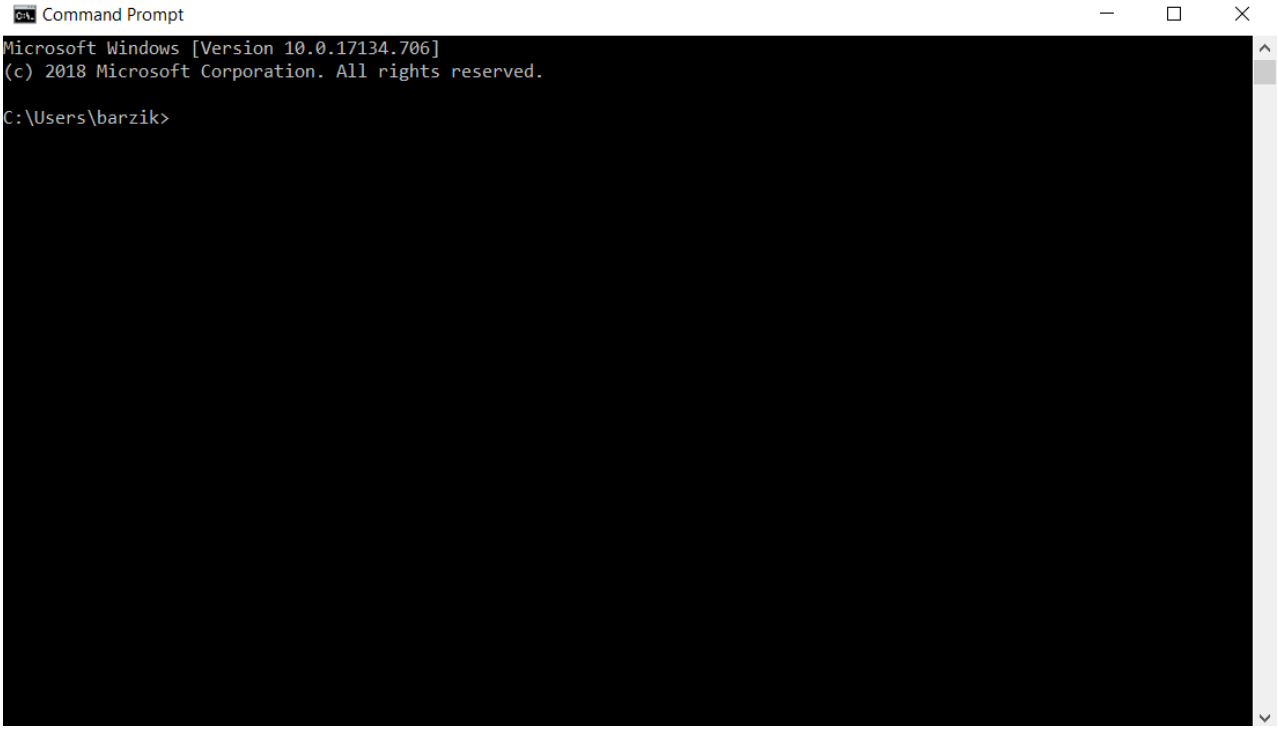
שפת תוכנה או אפילו סביבת הרצה לומדים דרך הידיים. בעבודה קשה. לא תוכלו ללמוד Node.js ללא לכלוך הידיים וכתיבה אמיתית. הספר הוא כלי עזר, הוא לא יחליף את ההקלדה שלכם. בדרך כלל הקושי האמיתי הוא בבניית סביבת עבודה יציבה וטובה, לכן הפרק הראשון שעוסק בהתקנת סביבת עבודה הוא קריטי.

לא תמיד כל הסבר המופיע בספר הוא קולע או מתאים. אם קראתם את הפרק פעם ופעמיים ושלוש פעמים ועדיין לא הבנתם – הבעיה לא בכם אלא בהסבר. לא להתייאש – פה כדאי להתייעץ בקהילות של ג'אווהסקריפט ויש לא מעט כאלו בפייסבוק ובמקומות אחרים. גם חיפוש בגוגל לפעמים יכול להוציא אתכם מבוץ אמיתי. נתקעתם? אל תתייאשו – הבעיה לא בכם. Node.js היא קלה אבל יש בה כמה חלקים קשים. נתקעתם? לא לדאוג – בקשו חילוץ. חפשו בגוגל, שחקו שוב ושוב עם הדוגמאות ובסוף זה יֵשב. אם אני הצלחתי – כל אחד יכול.

ניתן להעזר גם בבינה מלאכותית על מנת למצוא פתרון לשאלות. בינה מלאכותית טובה ללמידה היא Chat GPT הזמינה באתר https://chat.openai.com – ניתן להקליד שאלות בשפה חופשית, קטעי קוד או שגיאות שונות ולקבל פתרון כמעט מיידי. כדאי להזהר עם חלק מהתשובות וגם לא להסתמך על הבינה המלאכותית יותר מדי, כיוון שהמוטרה היא ללמוד Node.js. עם הידע שלכם ב-Node.js תוכלו ללמוד לעבוד עם בינה מלאכותית ולדעת לכוון אותה טוב יותר או לבדוק את הקוד שלה, אבל את שלבי הלימוד כדאי לעשות בעצמכם.

התקנת סביבת עבודה ועבודה עם טרמינל

כאמור, Node.js היא סביבת הרצה, וכדי שהיא תוכל לרוץ צריך להתקין אותה על המחשב, ממש כמו כל תוכנה אחרת. מה שההתקנה הזו עושה הוא פשוט למדי – היא מאפשרת לנו להפעיל את Node.js כמו כל תוכנה אחרת. זה הכול. אנו רגילים לפתוח תוכנות באמצעות אייקונים, אבל חלק מהתוכנות עובדות באמצעות הטרמינל. מה זה טרמינל? מקום שבו אתם יכולים להקליד פקודות. הוא קיים בכל מערכת הפעלה. בחלונות מגיעים לטרמינל באמצעות לחיצה על הזכוכית המגדלת (בחלונות 10) והקלדה של cmd – ראשי תיבות של command. נגיע לחלון שנראה כך:

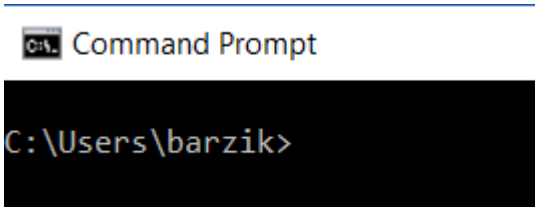


פִּתחו את החלון הזה, הקלידו notepad ולחצו על enter. ייפתח ה-Notepad של חלונות. ברכותיי! הפעלתם תוכנה באמצעות שורת הפקודה. הקלידו calc ולחצו על enter, תוכנת המחשבון תיפתח. זו עוד תוכנה שהפעלתם באמצעות ממשק הפקודה. הממשק הזה, או הטרמינל בשפת העם, הוא סביבת העבודה של Node.js. מפעילים את Node.js באמצעות הטרמינל. זה בדיוק מה שקורה בשרת "אמיתי". זכרו ששרת בסופו של דבר הוא מחשב – ייתכן שמחשב ללא מערכת הפעלה גרפית אלא רק עם טרמינל – אבל מחשב שמבוסס על חלונות או על לינוקס. ייתכן שהשרת חזק בהרבה מהמחשב הביתי שלכם – אבל עדיין מדובר במחשב לכל דבר. כאמור, Node.js רצה היטב

על שרתים מבוססי חלונות ועל שרתים מבוססי לינוקס. לצורך העניין, המחשב שלכם עכשיו הוא שרת.

צריך להכיר מעט את ממשק הפקודה של הטרמינל ולהתמצא בו. כיוון שהטרמינלים שונים בין חלונות ללינוקס, יש שוני קטן בין הפקודות. כיוון שחלונות היא מערכת ההפעלה הנפוצה, ומשתמשי לינוקס בדרך כלל מיומנים יותר בטרמינל, אני מסביר פה על הפקודות בחלונות. בסוף הפרק יש טבלה קטנה שבה מובאות הפקודות בלינוקס ובחלונות.

הטרמינל תמיד נפתח בהקשר של תיקייה כלשהי. תמיד אנחנו "נמצאים" בתוך תיקייה. בדרך כלל כשאני פותח טרמינל, הוא נפתח במיקום של המשתמש שלי. כך למשל, אם אני נכנס ל-cmd במחשב שלי – אני אראה את המיקום שלי:



אפשר לראות שאני נמצא בכונן C, בתיקיית Users ובתת התיקייה barzik, שזה שם המשתמש שלי בחלונות. אם אני אפתח את סייר הקבצים, אני אוכל לנווט לתיקייה הזו. הטרמינל הוא פשוט דרך נוספת לשוטט במחשב ולפעול בו – דרך שהיא לא גרפית, אבל כל מה שאני יכול לעשות בממשק הגרפי אני יכול לעשות בטרמינל.

כדי לראות את רשימת הקבצים בתיקייה, אני צריך להקליד dir. הקלדה של dir ואז enter תראה לי את רשימת הקבצים שיש בתיקייה שבה אני נמצא. רשימת הקבצים הזו תהיה זהה לחלוטין לרשימת הקבצים שאני רואה בסייר הקבצים כשאני נכנס לאותו מיקום. אם אצור קובץ או תיקייה בסייר הקבצים ואקליד שוב dir בטרמינל כשאני באותו המיקום של סייר הקבצים, אוכל לראות את הקובץ או את התיקייה בטרמינל.

כדי להיכנס לתיקייה מסוימת, אני צריך להקליד cd ואז את שם התיקייה ואז enter. אני יכול להשתמש במקש TAB על מנת לבצע השלמה אוטומטית. אם יש רווח בשם התיקייה, אני צריך להקיף אותו במירכאות. אם אני משתמש ב-TAB הוא יעשה את זה עבורי.

```
C:\Users\barzik>cd "My Documents"

C:\Users\barzik\My Documents>
```

אם אני רוצה לחזור לאחור, אני אכתוב cd .. שתי הנקודות יעלו אותי לתיקיית האב. אם אכתוב cd ../.. אני יכול לחזור לתיקיית האב של האב וכך הלאה.

```
C:\Users\barzik\My Documents>cd ../..

C:\Users>
```

ההפעלה של Node.js נעשית תמיד דרך הטרמינל. יש תוכנות שעוטפות את Node.js (כמו אלקטרון) שלא מחייבות אותנו לעשות את זה, אבל אנו לא נתייחס לכך בספר הזה.

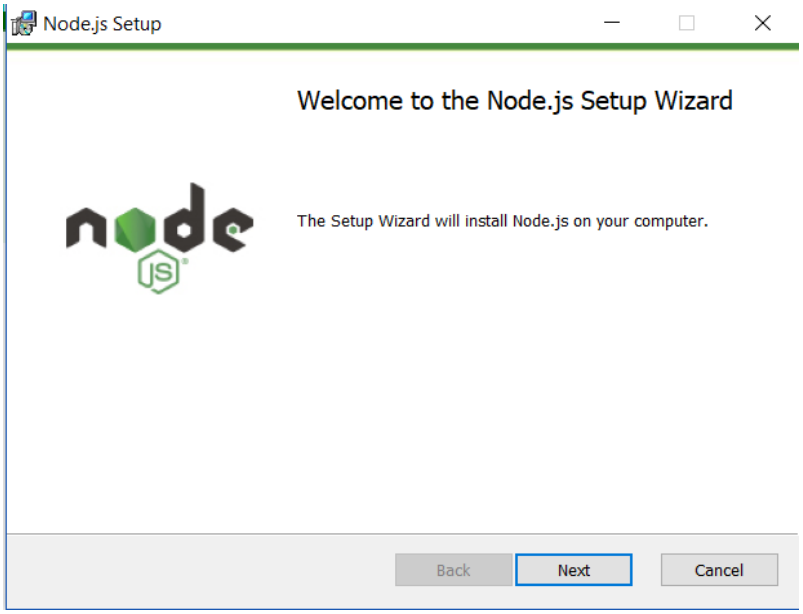
פקודה	פקודה בחלונות	פקודה בלינוקס/מק
הצג את רשימת הקבצים והתיקיות בתיקייה	dir	ls -al
עבור לתיקייה אחרת	cd	cd
יציאה מהטרמינל	exit	exit
ניקוי המסך	cls	clear

לאחר שאנו יודעים איך לעבוד עם הטרמינל, נתקין את Node.js. ההתקנה שונה במערכות הפעלה שונות אבל בכולן היא קלה למדי. בחרו את מערכת ההפעלה שלכם והתקינו את Node.js לפי ההוראות.

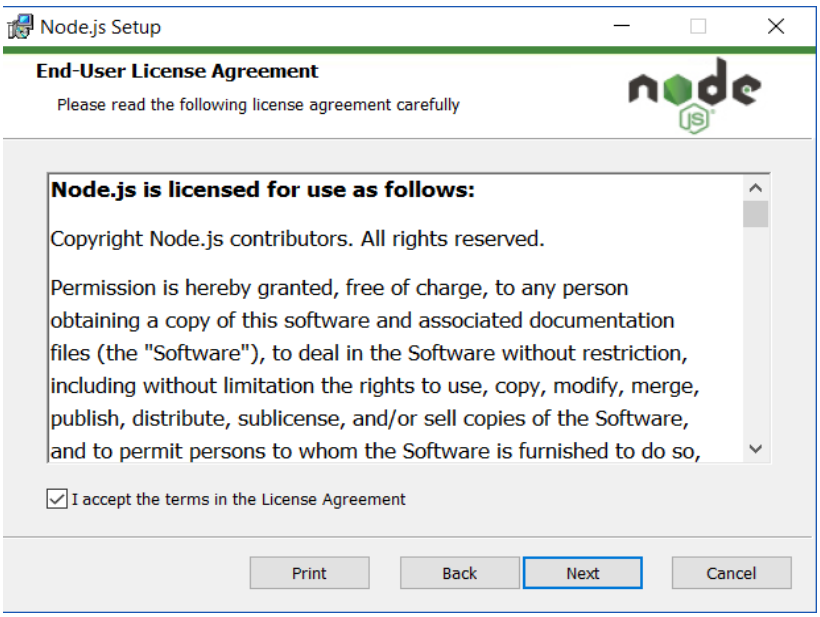
התקנה על חלונות

ההתקנה של Node.js על חלונות היא פשוטה מאוד. נקליד בגוגל Node.js Download או ניכנס אל: https://nodejs.org/en/download/

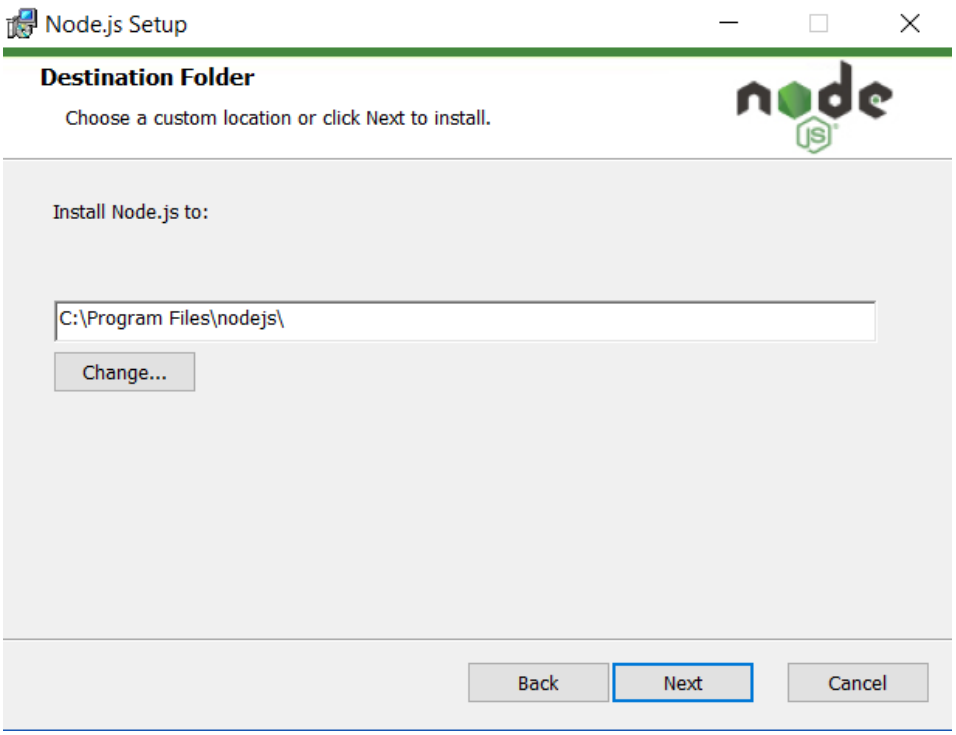
אנו נבחר בגרסת LTS – ראשי תיבות של "גרסה לטווח ארוך", ונבחר במערכת ההפעלה שלנו – אם מדובר בחלונות, יש לנו installer נוח. מורידים, לוחצים על התוכנה שיורדת:



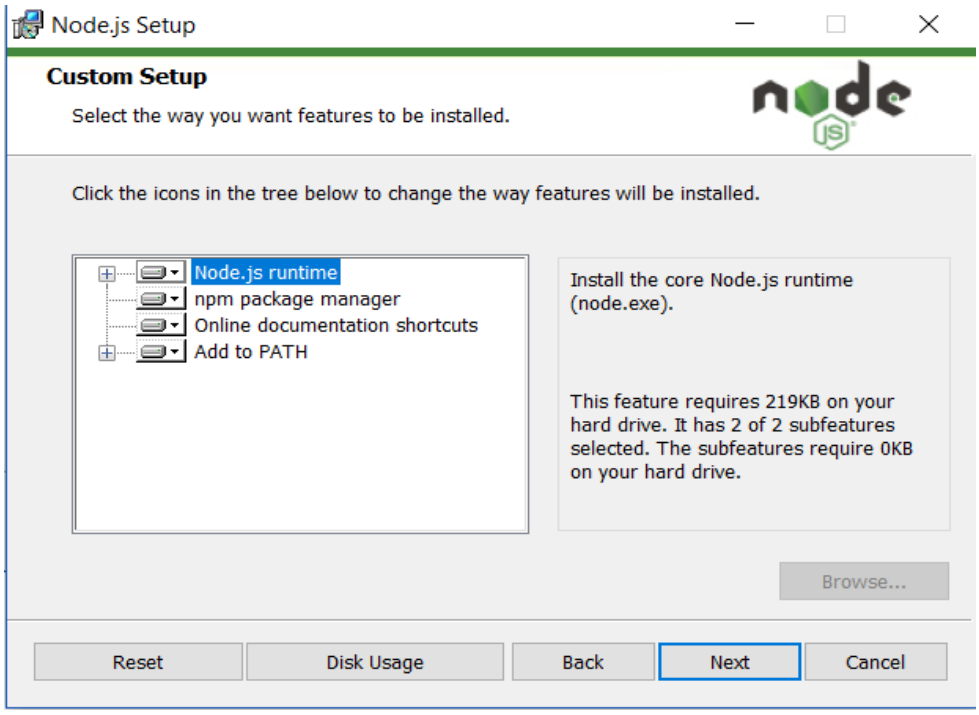
מקבלים את התנאים:



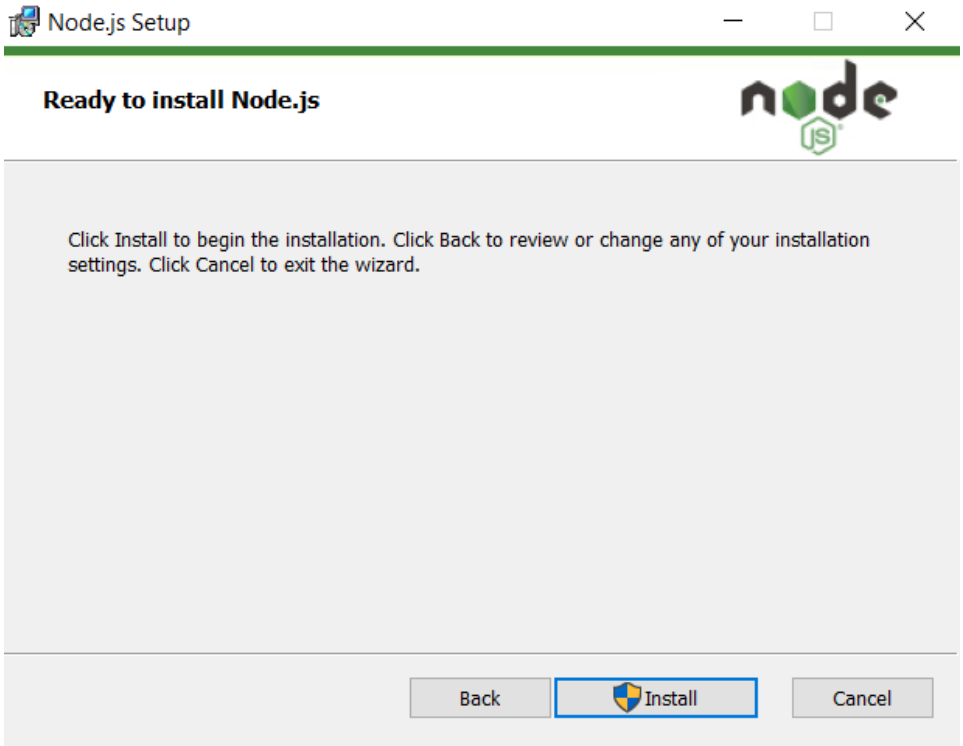
לוחצים על next:



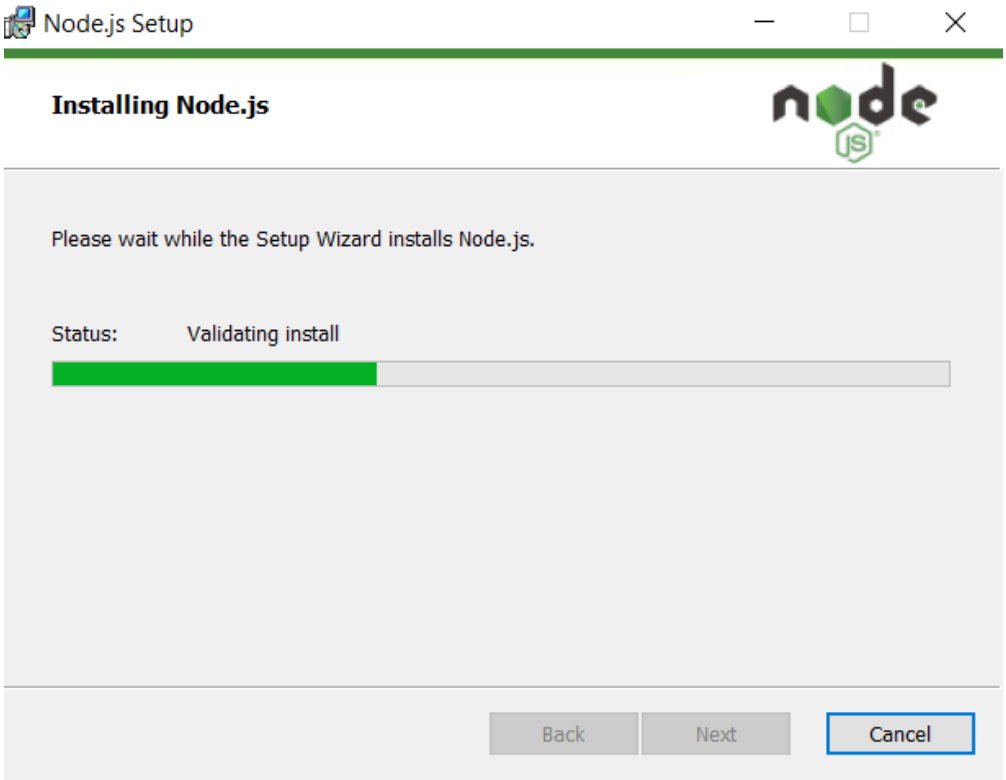
ושוב על next:



לחיצה על Install תתקין לבסוף את התוכנה:



כל מה שנותר הוא לחכות לסוף ההתקנה:



אחרי שההתקנה הושלמה, נפתח את הטרמינל שלנו (אם היה לנו טרמינל לפני ש-Node.js הותקנה, נצטרך לסגור ולפתוח אותו מחדש) ונקליד node -v. אם הכול תקין, אנו נראה את מספר הגרסה של Node.js.

```
C:\Users\barzik>node -v
v10.15.3

C:\Users\barzik>
```

התקנה על מק

ההתקנה של Node.js על מק היא פשוטה מאוד. נקליד בגוגל Node.js Download או ניכנס אל:
https://nodejs.org/en/download/

אנו נבחר בגרסת LTS – ראשי תיבות של "גרסה לטווח ארוך", ונבחר במק – יֵרד קובץ dmg שאותו אפשר להתקין כמו כל תוכנה אחרת בהנחה שהמחשב שלכם הוא לא מחשב ארגוני שמונע התקנות מהאינטרנט. ההתקנה היא פשוטה ביותר.

אם אתם משתמשים ב-Zsh או ב-Oh My Zsh אז אני ממליץ להתקין את Node.js בעזרת homebrew, באמצעות הפקודה (אם homebrew מותקנת אצלכם, וכדאי שהיא תהיה מותקנת):
brew install node

כך או אחרת, לאחר ההתקנה, כניסה לטרמינל והקלדה של node -v תראה לכם את מספר הגרסה בדיוק כמו בחלונות.

התקנה על לינוקס

אם אתם משתמשים בדביאן, אז בדרך כלל ברוב ההפצות sudo apt-get install node יטפל בהתקנה, אך אתם עלולים להתקין גרסה ישנה של Node.js וזה עלול להוות בעיה. למרות הפיתוי, היכנסו אל הקישור וקראו לפני ההתקנה את המדריך המלא לכל ההפצות של לינוקס, שמסביר על ההתקנות.
https://nodejs.org/en/download/package-manager/

אני יוצא מנקודת הנחה שמשתמשים בלינוקס הם מיומנים בהרבה ממשתמשי חלונות ויודעים להתקין חבילת תוכנה ללא הסברים נוספים. כך או אחרת – לאחר ההתקנה, כניסה לטרמינל והקלדה של node -v תראה לכם את מספר הגרסה בדיוק כמו בחלונות או במק.

תקלות נפוצות

זה נשמע מצחיק, אבל זה השלב הקשה ביותר שיש בכל למידת שפה חדשה, סביבה חדשה או כלי חדש – שלב ההתקנה. הסיכוי הגבוה ביותר לתקלות ולייאוש הוא פה. אם התרחשה תקלה – אל דאגה! Node.js היא אולטרה-פופולרית והסיכוי שאנשים אחרים נתקלו באותה תקלה הוא גבוה מאוד. נתקלתם בתקלה? העתיקו את מספר התקלה או טקסט מהודעת השגיאה וחפשו ברשת – סביר מאוד להניח שמישהו אחר נתקל באותה בעיה. בדרך כלל מדובר בבעיית אינטרנט של מחשבים ארגוניים שעובדים מאחורי רשת ארגונית. בדף הזה יש הסבר על תקלות נפוצות ופתרונן:

https://docs.npmjs.com/common-errors

אל תתייאשו אם זה קורה, נסו שוב ושוב והתעקשו עד שזה יצליח. אני מבטיח לכם ש-Node.js שווה את זה.

כתיבת התוכנה הראשונה

נפתח תיקיית עבודה – למשל node_projects – וניכנס אליה באמצעות הטרמינל.

```
C:\Users\barzik>cd node_projects

C:\Users\barzik\node_projects>
```

נפתח את ה-IDE החביב עלינו (אני משתמש ב-Visual Studio code), ניכנס לתיקייה וניצור קובץ בשם hello.js, שבו נכתוב:

```
console.log('Hello World!');
```

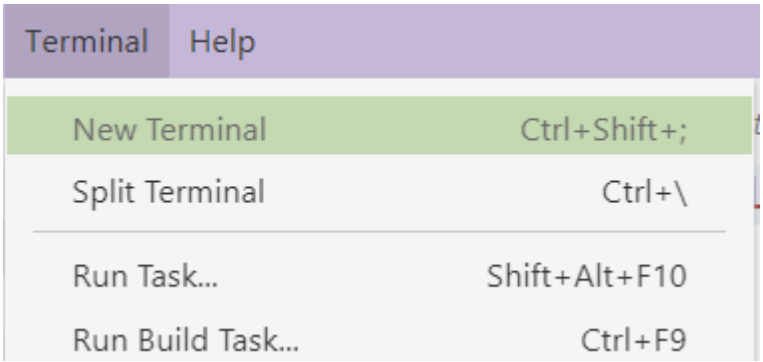
נשמור ואז נחזור לטרמינל ונכתוב node hello.js או נראה שמודפס לנו המשפט !Hello World:

```
C:\Users\barzik\node_projects>node hello.js
Hello World!
```

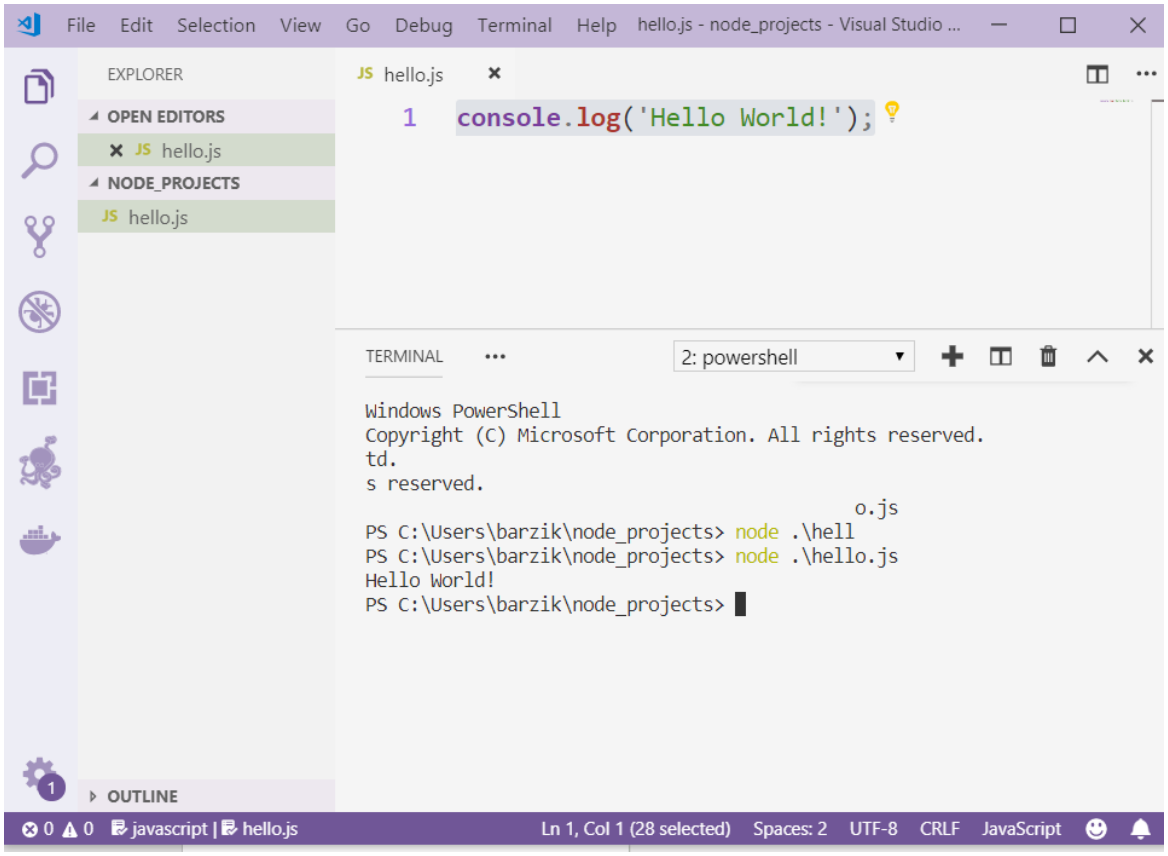
איזה כיף! כתבתם את תוכנית ה-Node.js הראשונה שלכם!

חשוב! אני יוצא מנקודת הנחה שאתם יודעים ג'אווהסקריפט ויודעים מה זה console.log ומה זה IDE ואפילו כבר מותקן לכם WebStorm, Atom, Visual Studio Code או כל IDE אחר על המחשב. אם זה נשמע לכם כמו סינית – אתם חייבים ללמוד ג'אווהסקריפט על מנת להתקדם בספר זה.

הערה חשובה נוספת: ב-Visual Studio Code וגם בסביבות עבודה אחרות הטרמינל מובנה ב-IDE. חפשו בתפריט העליון Terminal ולחצו עליו. בחרו ב-New Terminal. ייפתח לכם בתחתית המסך טרמינל במיקום של הקבצים שלכם.



זהו טרמינל הזהה אחד לאחד לזה של חלונות או של לינוקס או של מק. פשוט הוא נפתח בסביבת העבודה. אני ממליץ לכם לעבוד כך. אחד היתרונות הגדולים ביותר לעבודה באופן הזה הוא שאפשר לעבוד עם הדיבאגר המובנה של Visual Studio Code ממש מאפס. בספר זה אני לא מלמד על הדיבאגר.



תרגיל:

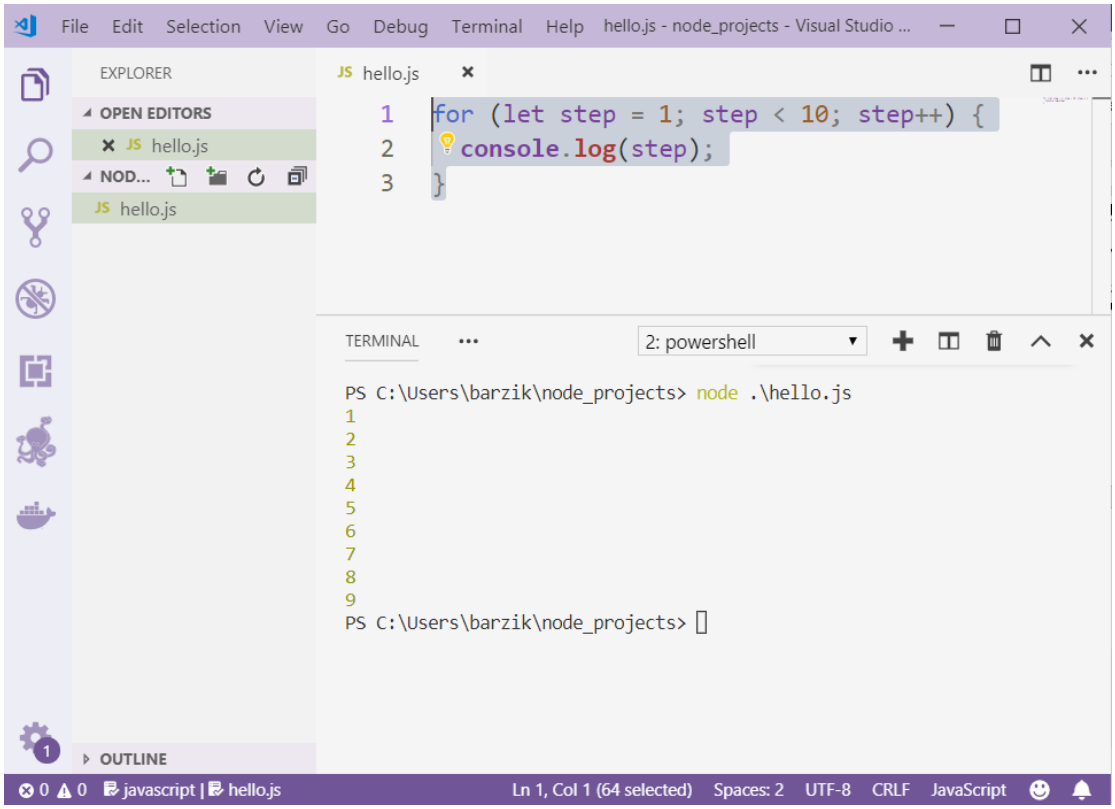
כתבו לולאה שרצה מ-1 עד 10 ומדפיסה את המספר של הלולאה. הריצו אותה ב-Node.js.

פתרון:

בתיקיית העבודה שלי אני יוצר קובץ בשם כלשהו, למשל hello.js. בתוכו אני כותב ג'אווהסקריפט רגיל לחלוטין של לולאה.

```
for (let step = 1; step < 10; step++) {
  console.log(step);
}
```

אני נכנס למיקום התיקייה, או באמצעות הטרמינל במערכת ההפעלה שלי או באמצעות הטרמינל ב-IDE שלי. אני מקפיד לוודא שאני בתיקייה שבה נמצא הקובץ וכותב node hello.js. אני אראה את המספרים 1 עד 10.



שימו לב שהשתמשתי כאן ב-TAB. ההשלמה האוטומטית הוסיפה את התווים \. שמסמנים "תיקייה נוכחית".

תרגיל:

כתבו קוד שזורק הערת שגיאה. הריצו את הקוד.

פתרון:

בתיקיית העבודה שלי אני יוצר קובץ בשם כלשהו, למשל hello.js. בתוכו אני כותב ג'אווהסקריפט רגיל שבו אני זורק שגיאה:

```
throw new Error('This is an Error!');
```

אני נכנס למיקום התיקייה, באמצעות הטרמינל במערכת ההפעלה שלי או באמצעות הטרמינל ב-IDE שלי. אני מקפיד לוודא שאני בתיקייה שבה נמצא הקובץ וכותב node hello.js. אני אראה שגיאה וגם את ה-stack trace – השרשרת של הפקודות שהובילה לשגיאה. בראשיתה אני בעצם רואה את הסיבה לבעיה – השורה הראשונה בתרגיל:

```
1   throw new Error('This is an Error!');
```

```
PROBLEMS   TERMINAL   ...   2: powershell

C:\Users\barzik\node_projects\hello.js:1
(function (exports, require, module, __filename, __dirname) { throw ne
 Error('This is an Error!');
                                                              ^

Error: This is an Error!
    at Object.<anonymous> (C:\Users\barzik\node_projects\hello.js:1:69
    at Module._compile (internal/modules/cjs/loader.js:701:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:7
2:10)
    at Module.load (internal/modules/cjs/loader.js:600:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:539:12)
    at Function.Module._load (internal/modules/cjs/loader.js:531:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:754:12)
    at startup (internal/bootstrap/node.js:283:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:622:3)
PS C:\Users\barzik\node_projects>
```

במהלך הלימודים, אתם תראו את ה-stack trace של השגיאות לא מעט. הוא אמור לעזור לכם להבין איפה טעיתם בהקלדה ואיפה שגיתם בסינטקס. אין מה להתבלבל או להיבהל. במערכות מורכבות הוא מסייע מאוד להבין מה הבעיה בדיוק. כאן זרקנו שגיאה בקובץ אחד, אז נראה את המקור בשורה הראשונה. במערכות מורכבות יותר הבעיה האמיתית תופיע יותר בתחתית. אבל העיקרון הוא אותו עיקרון – שגיאה נראית כך.

פרק 1

REQUIRE ומודולים

Require ומודולים

הכוח הגדול של Node.js הוא חבילות התוכנה שלו. ל-Node.js יש יותר ממאה אלף חבילות תוכנה שכל אחד יכול להשתמש בהן. איך משתמשים בהן? ישנן שתי דרכים עיקריות בהתאם לשיטת המודולים שהמתכנת בוחר. אנו נסביר על השיטה שנקראת CommonJS – כיוון שהיא ברירת המחדל של Node.js. על השיטה השניה, ECMAScript Modules אנו נלמד בהמשך.

שיטת ברירת המחדל, CommonJS עובדת באמצעות **require**. אחד ההבדלים הגדולים בין Node.js לבין ג'אווהסקריפט בסביבת דפדפן הוא ה-require. הוא כמעט ייחודי ל-Node.js (יש ספריות נוספות שמשתמשות בו, אך ללא ספק הוא סימן היכר משמעותי של Node.js) ומשמש ליבוא ולשימוש בחבילות תוכנה. ל-Node.js יש חבילות תוכנה שבאות איתו כברירת מחדל ואנו נשתמש בהן בהתחלה.

חבילת התוכנה הראשונה שאנו נשתמש בה היא fs, שידועה גם כ-file system. זוהי חבילת תוכנה שבאה תמיד עם Node.js (אי-אפשר להתקין את Node.js בלעדיה) ומסייעת לנו לטפל במערכת הקבצים של המחשב. יש לה מתודות רבות שמתפעלות את מערכת הקבצים. מתודה אחת שאבחר בה היא readdir – מתודה שמקבלת שני פרמטרים. הראשון הוא הנתיב של התיקייה שאני רוצה לבדוק והשני הוא פונקציית קולבק. פונקציית הקולבק נקראת על ידי readdir בגמר הפעולה ומחזירה שני ארגומנטים – אובייקט שגיאה (אם היא מתקיימת) ואובייקט תוצאה שמציג את הקבצים והתיקיות שיש לנו בנתיב.

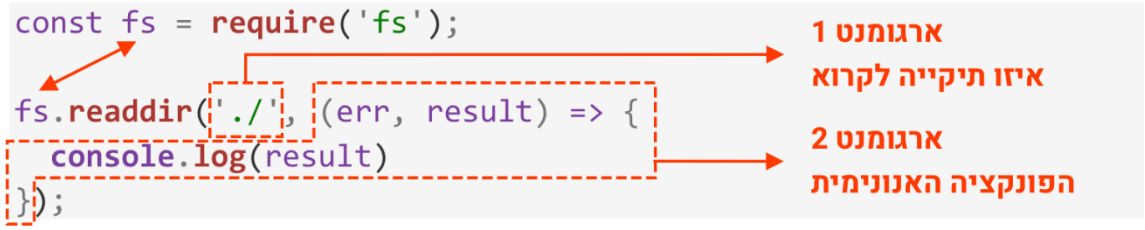
אצור קובץ בשם app.js ואכניס בו את הקוד הבא:

```
const fs = require('fs');

fs.readdir('./', (err, result) => {
  console.log(result);
});
```

מה הקוד הזה בעצם אומר? הדבר שלא אמור להיות ברור למתכנת ג'אווהסקריפט מן השורה הוא ה-require. כאן אני בעצם קורא לחבילת התוכנה או למודול שנקרא fs. מדובר בעצם באובייקט כמו כל אובייקט שאנו מכירים, שיש לו מתודה שנקראת readdir. זוהי מתודה אסינכרונית שמקבלת ארגומנט ראשון של התיקייה שבה מחפשים וארגומנט שני של קולבק. בקולבק יש שני

ארגומנטים – אובייקט שגיאה ואובייקט תוצאה. זהו פורמט סטנדרטי של קולבקים ב-Node.js ואני ארחיב על כך בפרק על פרומיסים ב-Node.js.



כאמור, מתכנת ג'אווהסקריפט אמור להבין איך קוד אסינכרוני עובד ואיך קולבקים עובדים. אם זה נראה לכם כמו סינית, זה הזמן לחזור על החומר.

אריץ את האפליקציה שלי באמצעות node app.js. מה שאראה בקונסולה הוא את רשימת הקבצים שיש בתיקייה – במקרה הזה app.js בלבד.

```
PS C:\Users\barzik\node_projects> node .\app.js
[ 'app.js' ]
```

בואו ניצור קובץ באמצעות fs. יצירת הקובץ נעשית באמצעות המתודה writeFile. המתודה הזו מקבלת שלושה ארגומנטים. הראשון הוא שם הקובץ, השני הוא תוכן הקובץ והשלישי הוא קולבק שבו מועבר אובייקט שגיאה. אם אין שגיאה, האובייקט ריק.

```
const fs = require('fs');

fs.writeFile('./test.txt', 'Hello World!', (err) => {
  if (err) throw err;
  console.log('Created file!');
});
```

אם תשמרו את הקוד הזה ב-app.js ותריצו אותו, תראו שנוצר קובץ בשם test.txt. אם תפתחו אותו, תראו שהתוכן הוא hello world!.

אפשר לבצע require לכמה מודולים בו-זמנית. מודול נוסף שבא יחד עם Node.js הוא מודול os, שנותן מידע על מערכת ההפעלה. מתודה אחת מתוך os היא מתודת homedir – המתודה הזו לא

מקבלת ארגומנטים, ומחזירה את תיקיית ה"בית" של מערכת ההפעלה. אם אני למשל בחלונות, תיקיית הבית שלי היא c:\Users\barzik. אם אני בלינוקס, תיקיית הבית שלי היא home/barzik/. אם אני כותב סקריפט של Node.js, אני רוצה שהוא יעבוד בלי קשר למערכת ההפעלה ואני לא מעוניין לדעת מה היא. שימוש ב-os הוא הדרך.

כך אכתוב את הקוד:

```
const fs = require('fs');
const os = require('os');

const homeDirectory = os.homedir();

fs.writeFile(`${homeDirectory}/test.txt`, 'Hello World!', (err) => {
  if (err) throw err;
  console.log('Created file!');
});
```

אפשר לראות שפשוט עשיתי require ל-os והשתמשתי במתודה homeDir. מדובר במתודה סינכרונית שלא מקבלת קולבק, אז אין בעיה מהותית להשתמש בה.

ה-require הוא לא קסם או וודו אפל. מדובר בקבלה של מודול, וברגע שקיבלתי אותו אני יכול להשתמש בו בדיוק כמו שאני משתמש בכל מודול אחר בג'אווהסקריפט. כך למשל, אם אני רוצה לייעל את הקוד הקודם ולחסוך שורה, אני יכול לבצע require ל-os ומייד לקרוא למתודה, וכך לחסוך משתנה:

```
const fs = require('fs');
const homeDirectory = require('os').homedir();

fs.writeFile(`${homeDirectory}/test.txt`, 'Hello World!', (err) => {
  if (err) throw err;
  console.log('Created file!');
});
```

אני לא חושב שמומלץ להשתמש בדוגמה שלעיל, אבל היא אמורה להבהיר לכם שלא מדובר בקסם. באחד מהפרקים הבאים אנו נראה מקרוב איך ה-require עובד כאשר נכתוב מודול משלנו.

מודולים של ECMAScript

שיטה נוספת לצריכת מודולים היא זו שנקראת ECMAScript Modules והיא תופסת תאוצה בשנים באחרונות באקוסיסטם של ג'אווהסקריפט וגם Node.js הולכת ומטמיעה אותה, למרות שעדיין ברירת המחדל היא Common.js ו-require. אבל רואים יותר ויותר שימוש ב-ECMAScript modules.

בשיטה זו, אנו צורכים מודולים באמצעות. המילה השמורה import. אני מקליד import ואז את שם המשתנה שאני בוחר ואז את המילה השמורה from ואז את שם החבילה. אני אתן דוגמה לקוד של ECMAScript modules שתהיה מובנת:

```
import fs from 'fs';
fs.writeFile(`./test.txt`, 'Hello World!', (err) => {
  if (err) throw err;
  fs.readdir('./', (err, result) => {
    console.log(result);
  });
});
```

הדוגמה זהה לחלוטין לדוגמה האחרונה שנתנו עם require, רק שפה אנו משתמשים ב-import. זה כל ההבדל. במקום הפקודה הזו, המייבאת את fs באמצעות require:

אנו משתמשים בפקודה הזו, המייבאת את fs באמצעות import:

על מנת להשתמש ב-import, אנו זקוקים לשמור את שם הקובץ בסיומת mjs ולא בסיומת js. כלומר הקוד שלעיל יעבוד אך ורק אם הוא יהיה בקובץ שהסיומת שלו היא mjs כמו app.mjs.

גם פה לא מדובר בקסם אפל אלא בדרך פשוטה להשתמש במודולים. ישנם מודולים שמייצאים כמה פונקציות או מחלקות ואז אנו צריכים לייבא את המשתנים שלהן באמצעות סוגריים מסולסלים סביב שם הפונקציה. Homedir שיש בקובץ os. הדגמנו בפרק זה איך משתמשים בפונקצית homedir שהיא אחת מהפונקציות שנחשפות במודול os של Node.js. כיצד אני אשתמש בה עם import? באופן הזה:

```
import fs from 'fs';
import { homedir } from 'os';

const homeDirectory = homedir();
fs.writeFile(`${homeDirectory}/test.txt`, 'Hello World!', (err) => {
  if (err) throw err;
  console.log('Created file!');
});
```

כיוון שמודול 'os' חושף כמה פונקציות או מתודות, אני אקרא למתודה שאני צריך באמצעות הצבת שם המשתנה המכיל אותה בתוך סוגריים מסולסלים וכמובן אשמור את הקובץ בשם עם סיומת mjs.

ישנן דרכים נוספות לגרום לקוד שלנו לעבוד עם import מלבד לעבוד עם קבצים בסיומת mjs ואנו נדון בכך בהמשך. אנו לומדים על השיטה של require ברוב הספר, אך חשוב להכיר גם את השיטה של ECMAScript modules שהיא .

תרגיל:

צרו תוכנת Node.js שתיצור קובץ בתיקייה ומייד אחרי כן תציג את הקבצים בתיקייה (אחד מהם אמור להיות הקובץ).

פתרון:

```
const fs = require('fs');

fs.writeFile(`./test.txt`, 'Hello World!', (err) => {
  if (err) throw err;
  fs.readdir('./', (err, result) => {
    console.log(result);
  });
});
```

הדבר הראשון שאני עושה הוא require ל-fs. אני יוצר את הקובץ עם מתודת writeFile ואני מעביר לה שלושה ארגומנטים. ארגומנט ראשון הוא שם הקובץ שאותו אני יוצר, הארגומנט השני הוא ה-Hello world והשלישי הוא קולבק. פונקציית הקולבק נקראת אחרי שהקובץ סיים להיווצר. בתוכה אני אבצע קריאה נוספת ל-fs, לקריאת התיקייה ולהדפסת התוצאות.

```
const fs = require('fs');

fs.writeFile('./test.txt', 'Hello World!', (err) => {
  if (err) throw err;
  fs.readdir('./', (err, result) => {
    console.log(result);
  });
});
```

קולבק: פונקציית חץ

הפונקציה נמצאת בתוך הקולבק

תרגיל:

חיזרו על התרגיל הקודם אך בצעו import ולא require.

פתרון:

```
import fs from 'fs';

fs.writeFile(`./test.txt`, 'Hello World!', (err) => {
  if (err) throw err;
  fs.readdir('./', (err, result) => {
    console.log(result);
  });
});
```

הפתרון זהה לחלוטין מבחינת הקוד שלו לפתרון הקודם, אך אנו מחליפים את require ב-import. פשוט מקלידים את המילה השמורה import, השם של המודול שאנו מייבאים, במקרה הזה fs, ואז המילה השמורה from ומהיכן אנו מייבאים אותה.

פרק 2

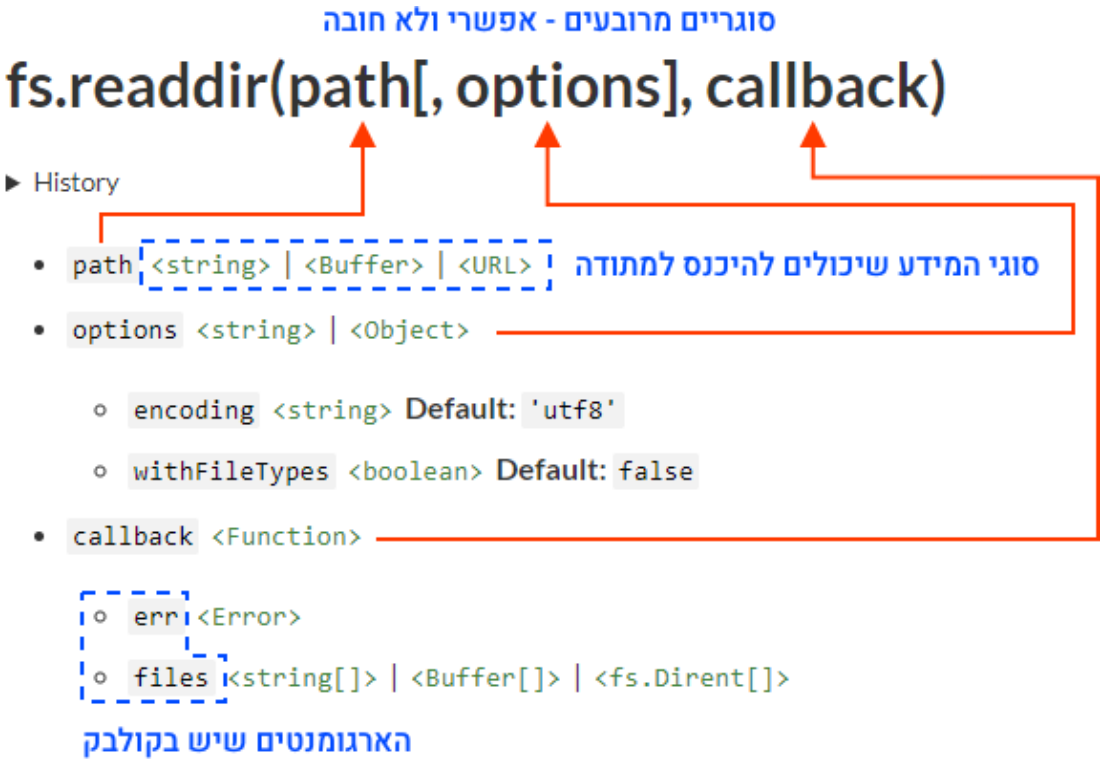
היכרות עם הדוקומנטציה של NODE.JS

היכרות עם הדוקומנטציה של Node.js

בפרק הקודם הסברתי על המודולים של ברירת המחדל fs וגם os. מהיכן הכרתי אותם? הידע הזה לא בא לי בחלום אלא הוא כתוב בדוקומנטציה המפורטת של Node.js. בדוקומנטציה הזו יש פירוט של כל המודולים שבאים כברירת מחדל עם Node.js.

הדוקומנטציה נמצאת באתר הרשמי של Node.js בכתובת: https://nodejs.org/api/ אם תחפשו בו את המודול **File System** תוכלו לראות את כל המתודות באופן מפורש. אחת מהן היא readdir. אם תיכנסו לדוקומנטציה שלה תוכלו לראות את כל הארגומנטים שהמתודה readdir מקבלת. העיצוב של האתר משתנה לעיתים, אבל בסופו של דבר זה נראה כך: שם המתודה, מה היא עושה והארגומנטים שהיא מקבלת. אם יש קולבק – הפונקציה הנקראת לאחר השלמת הפעולה, יהיה מידע על שמה של הפונקציה.

בואו נסתכל על הדוקומנטציה של readdir על מנת לנסות להבין:



ראשית אני רואה את שם המתודה, fs.readdir, ואני רואה שאני יכול להעביר לה שלושה ארגומנטים. הראשון הוא path, השני הוא options שסביבו יש סוגריים מרובעים כדי לרמז על כך שהוא אפשרי ולא חובה. השלישי הוא הקולבק.

מתחת לכותרת של המתודה, אני רואה את הפירוט של סוג המידע שיכול להיכנס לכל ארגומנט. הראשון, path, יכול להיות מחרוזת טקסט, כפי שראינו קודם, אבל הוא יכול להיות גם סוגי מידע אחרים שאני לא אפרט כאן.

השני, options הוא לא חובה. אנו יודעים את זה בגלל הסוגריים המרובעים סביב הארגומנט הזה בכותרת. אני יכול להעביר לו מחרוזת טקסט או אובייקט המכיל את כל האפשרויות.

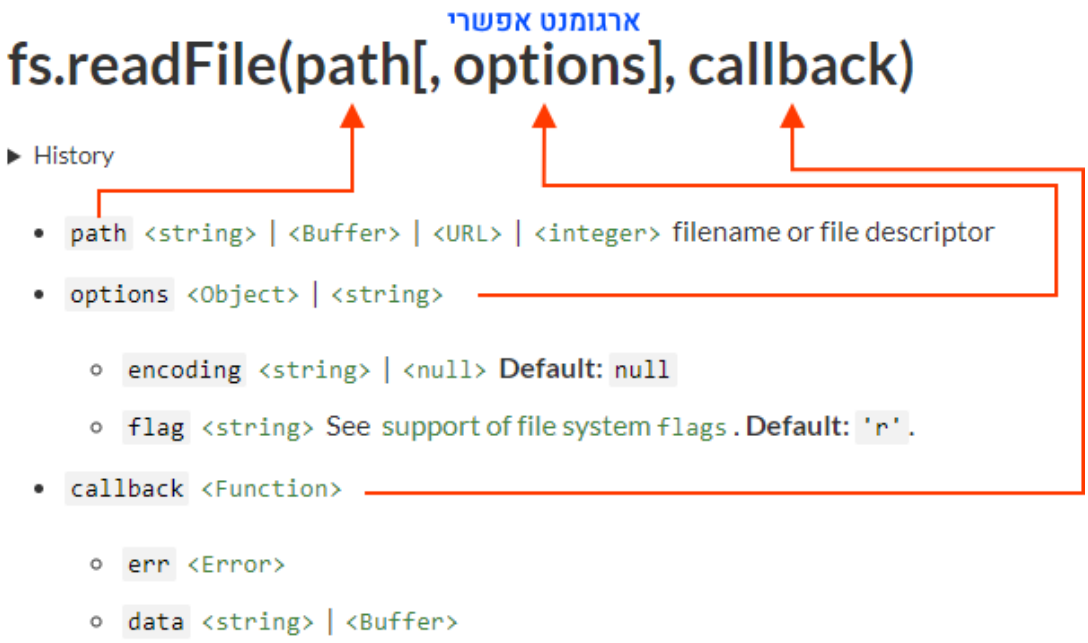
השלישי, הוא הקולבק שלנו. זוהי פונקציה (מקובל להעביר פונקציית חץ) שמופעלת לאחר שהפעולה מסתיימת. כלומר ברגע ש-fs.readdir מסיימת לקרוא את תוכן התיקייה, היא לוקחת את הפונקציה שהעברנו כארגומנט שלישי ומפעילה אותה. כשהיא מפעילה אותה היא מאכלסת את שני הארגומנטים בקולבק. אני יכול לטפל בקולבק בארגומנטים האלו או לא.
הינה מה שיצרתי בעקבות הקריאה בדוקומנטציה:

```
const fs = require('fs');

fs.readdir('./', {encoding: 'hex'}, (err, result) => {
  if (err) throw err;
  console.log(result); // [ '6170702e6a73', '746573742e747874' ]
});
```

כדאי לשים לב שביקשתי קידוד (encoding) אחר באמצעות ארגומנט ה-options. בדוקומנטציה מפורט שהקידוד של ברירת המחדל הוא UTF-8, אבל אפשר לשנות אותו.

בואו נבדוק בדוקומנטציה את fs.readFile, מתודה המשמשת לקריאת קובץ. אפשר לחפש אותה בדוקומנטציה. אם תיכנסו לדוקומנטציה
https://nodejs.org/api/fs.html#fs_fs_readfile_path_options_callback ותחפשו אותה, תראו משהו הדומה לזה:



האמת היא שזה די מזכיר את המתודה readdir. גם כאן יש שלושה ארגומנטים. הראשון הוא path שיכול להיות מחרוזת טקסט (ויכול להיות גם סוג אחר), השני הוא אובייקט אפשרויות שהוא לא חובה, והשלישי הוא הקולבק. הקולבק הזה אמור להיות מופעל כשהמתודה readFile מסיימת את תפקידה. היא תפעיל את הפונקציה הזו ותעביר אליה שני ארגומנטים, שגיאה (אם קיימת) ואת המידע.

כתיבה של המתודה הזו גם היא מזכירה מאוד את readdir. אני אצור קובץ בשם app.js, אכניס לתוכו את הקוד הבא:

```
const fs = require('fs');

fs.readFile('./app.js', (err, result) => {
  if (err) throw err;
  console.log(result);
});
```

ואפעיל אותו באמצעות node app.js. מה שהסקריפט עושה הוא בעצם לקרוא את עצמו ולהציג את התוכן. אני אצפה לראות בטרמינל את כל הקובץ, בדיוק כמו שעם readdir אני רואה את רשימת הקבצים.

אבל כשאני מפעיל את התוכנה הזו, אני רואה משהו לא צפוי. במקום לראות את כל הטקסט, אני רואה משהו כזה:

```
<Buffer 63 6f 6e 73 74 20 66 73 20 3d 20 72 65 71 75 69 72 65 28 27
```

למה? אם אני אמשיך לעיין בדוקומנטציה אני אראה שבאופן מאוד מפורש כתוב שם ש:

```
The callback is passed two arguments (err, data), where data is the
contents of the file.
If no encoding is specified, then the raw buffer is returned.
```

זה מסביר על הקולבק. הקולבק מקבל שני ארגומנטים – אובייקט שגיאה, והמידע – המידע אמור להיות תוכן הקובץ. אבל הלאה מוסבר שאם לא מפורט שום encoding, אנו מקבלים Buffer וזה בדיוק מה שקיבלנו. איך אנו בעצם פותרים את הבעיה? גם זה מוסבר בדוקומנטציה (ואפילו יש דוגמה). פשוט להעביר קידוד:

```
const fs = require('fs');

fs.readFile('./app.js', {encoding: 'utf8'}, (err, result) => {
  if (err) throw err;
  console.log(result);
});
```

ההרצה של הקוד הזה כבר תציג לי את תוכן הקובץ כמו שאני מצפה לו:

```
PS C:\Users\barzik\node_projects> node .\app.js
const fs = require('fs');

fs.readFile('./app.js', {encoding: 'utf8'}, (err, result) => {
  if (err) throw err;
  console.log(result);
});
```

יש סיבה שהקדשתי פרק שלם לדוקומנטציה – מומלץ מאוד להכיר אותה ואפילו לבדוק בה לפני שרצים לגוגל כדי לפתור בעיות. בדוקומנטציה יש פירוט של כל המודולים הבסיסיים של Node.js ומומלץ לעבור עליה ולהכיר אותה. אתם משתמשים במודול מסוים ולא מקבלים את התוצאות כפי שרציתם? כדאי לקרוא את הדוקומנטציה.

תרגיל:

אתרו בדוקומנטציה את המתודה fs.mkdir והשתמשו בה בסקריפט של Node.js על מנת ליצור תיקייה במיקום כלשהו במחשב שלכם.

פתרון:

המתודה fs.mkdir נמצאת בדוקומנטציה תחת File System פה:
https://nodejs.org/api/fs.html#fs_fs_mkdir_path_options_callback
אם נסתכל עליה נראה שהיא זהה למתודות של readdir ו-readFile. גם היא מקבלת שלושה ארגומנטים:

fs.mkdir(path[, options], callback)

ארגומנט ראשון — ארגומנט שני [לא חובה] — ארגומנט שלישי קולבק

▶ History

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<Object>` | `<integer>`
  - `recursive` `<boolean>` **Default:** `false`
  - `mode` `<integer>` Not supported on Windows. **Default:** `0o777`.
- `callback` `<Function>`
  - `err` `<Error>`

ארגומנט ראשון הוא המיקום שבו רוצים ליצור את התיקייה החדשה, הארגומנט השני הוא אפשרויות, והארגומנט השלישי הוא הקולבק שנקרא לאחר השלמת הפעולה. הקולבק הזה מקבל אך ורק אובייקט שגיאה אם הפעולה נכשלת. כדאי לשים לב שהוא תמיד מופעל לאחר הצלחת הפעולה ואם אין שם אובייקט שגיאה, אני יכול להניח שהפעולה הצליחה.

כך אני כותב את הסקריפט. בחרתי לכתוב אותו בקובץ app.js בתיקיית העבודה שלי.

```
const fs = require('fs');

fs.mkdir('./Hello', (err) => {
  if (err) throw err;
  console.log('Directory created!');
});
```

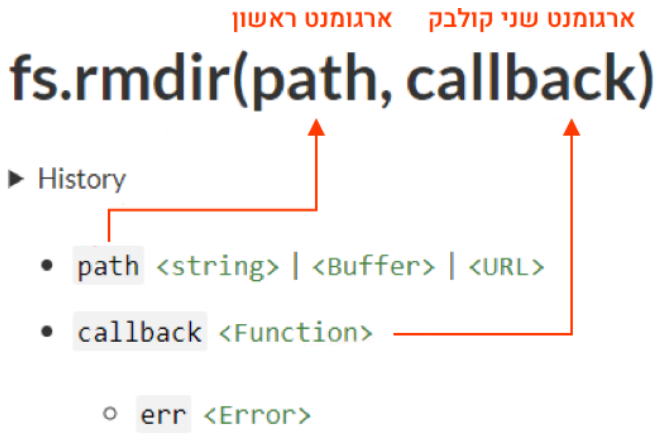
הנתיב שבחרתי הוא ./Hello. זה אומר שאני יוצר את תיקיית Hello כתיקיית בת מהנתיב שבו app.js נמצא. הרצה שלו באמצעות node app.js תיצור את התיקייה הזו.

תרגיל:

אתרו בדוקומנטציה את המתודה fs.rmdir והשתמשו בה כדי למחוק את התיקייה Hello שיצרתם בתרגיל הקודם.

פתרון:

גם כאן קל להשתמש בדוקומנטציה של Node.js על מנת לאתר את המתודה הזו. אנו רואים שיש לה שני ארגומנטים בלבד. הראשון הוא שם התיקייה שאותה אנו רוצים למחוק והשני הוא הקולבק. הקולבק גם הוא פשוט. הוא מעביר אך ורק אובייקט שגיאה אם הפעולה נכשלת.



הקוד שלי ייראה כך:

```
const fs = require('fs');

fs.rmdir('./Hello', (err) => {
  if (err) throw err;
  console.log('Directory deleted!');
});
```

אם הוא שמור ב-app.js, ההרצה שלו תיעשה על ידי node app.js ואם התיקייה קיימת, אני אראה בטרמינל הודעה של Directory Deleted!, והתיקייה שיצרתי קודם תימחק.

פרק 3

גרסאות סינכרוניות למתודות אסינכרוניות

גרסאות סינכרוניות למתודות אסינכרוניות

כתבתי בתחילת הספר ש-Node.js היא אסינכרונית ושזה הכוח שלה. Node.js רצה על הליך אחד במעבד ואם אנו נדרשים לפעולה מסוימת כמו קריאת קובץ, התוכנה לא עומדת וממתינה לקובץ הזה, אלא ממשיכה הלאה. אם אני כותב משהו כזה למשל:

```
const fs = require('fs');

console.log('Before readdir');

fs.readdir('./', (err, result) => {

  if (err) throw err;
  console.log(`readdir is completed. Result: ${result}`);
});

console.log('After readdir');
```

מה שאני אראה בטרמינל הוא זה:

```
Before readdir
After readdir
readdir is completed. Result: app.js,test.txt
```

למה? כיבהתחלה אנו מדפיסים את ה-Before, קוראים ל-readdir. בזמן ש-Node.js רצה לתיקייה, אפשר להמשיך ואכן השורה הבאה, שהיא ההרצה של ה-After, רצה. רק כש-readdir סיימה את העבודה, הקולבק מופעל ומביא את התוצאות.

ואם יש לנו כמה קולבקים שכל אחד מהם מושלם בזמן אחר – כל קולבק ירוץ כשהוא יושלם. אם יש כמה קולבקים שיושלמו, הם ייכנסו לתור. אבל העניין הוא ש-Node.js לא עוצרת ומחכה. אבל היא יכולה לעשות את זה באמצעות מתודות סינכרוניות. לפעמים אנחנו צריכים לעצור את הסקריפט – בדרך כלל כשאנו בונים CLI (כלים לניהול שרתים או סביבות פיתוח) ואין טעם להמשיך את פעולת הסקריפט אם פעולה מסוימת לא מושלמת.

ואיפה מוצאים את הפעולות הסינכרוניות האלו? בדוקומנטציה כמובן! אם חיטטתם בדוקומנטציה, הייתם יכולים לראות שיש מתודות שזהות למתודות שאותן תרגלנו, אבל מוצמד להן Sync לשם. המתודות האלו נמצאות בעיקר במודול File System. כך למשל, יש לנו **readdir** ויש לנו **readdirSync**.

למתודות סינכרוניות אין קולבק והן מחזירות את התוצאה שלהן בדומה לקולבק. כך למשל, readdirSync תיראה כך:

```
const fs = require('fs');

const result = fs.readdirSync('./');
console.log(`readdir is completed. Result: ${result}`);
```

זה אומר שהקוד ממש יעצור ויחכה להשלמת הפעולה. אם אני אשים console.log לפני ואחרי, אני אראה שהקוד רץ לפי הסדר. אין קולבקים, אין אסינכרוניות:

```
const fs = require('fs');

console.log('Before readdir');

const result = fs.readdirSync('./');
console.log(`readdir is completed. Result: ${result}`);

console.log('After readdir');
```

זה מה שאני אקבל בהרצה:

```
Before readdir
readdir is completed. Result: app.js,test.txt
After readdir
```

ואיך אני תופס שגיאות? במקרה הזה אין לי קולבק שמעביר אובייקט שגיאה (אם יש שגיאה). אז פה אני משתמש ב-try-catch רגיל לחלוטין שיפעל אם יש שגיאה. בקוד הבא למשל אני מנסה לקרוא תיקייה שלא קיימת באמצעות readdirSync – הפונקציה הסינכרונית תעיף לי שגיאה שאותה אני יכול לתפוס עם try-catch ולטפל בה כרגיל:

```
const fs = require('fs');

console.log('Before readdir');

try {
  const result = fs.readdirSync('./blahbla');
  console.log(`readdir is completed. Result: ${result}`);
} catch(error) {
  console.log('Error has occured!');
}
console.log('After readdir');
```

התוצאה של הרצת הקוד הזה תהיה:

```
Before readdir
Error has occurred!
After readdir
```

וכמובן הסקריפט לא יתפוצץ עם שגיאה ו-stack trace, אם לא יהיה try-catch.

כמעט לכל מתודה שמטפלת בקבצים יש הגרסה הסינכרונית שלה.

הינה רשימת המתודות שלמדנו עד כה:

תיאור המתודה	גרסה אסינכרונית	גרסה סינכרונית
יצירת תיקייה	fs.mkdir(path[, options], callback)	fs.mkdirSync(path], options])
קריאת תוכן תיקייה	fs.readdir(path[, options], callback)	fs.readdirSync(path], options])
מחיקת תיקייה	fs.rmdir(path, callback)	fs.rmdirSync(path)
קריאת קובץ	fs.readFile(path], options], callback)	fs.readFileSync(path], options])
יצירת קובץ	fs.writeFile(file, data], options], callback)	fs.writeFileSync(file, data[, options])

מאוד לא מומלץ להשתמש בגרסאות סינכרוניות אלא אם כן אתם יודעים מה אתם צריכים – משתמשים בהן בדרך כלל לשימושים ייחודיים. מפתה מאוד, במיוחד אם לא סגורים עד הסוף על האסינכרוניות, להשתמש בקוד הזה. אבל זה עלול להיות הרסני במקומות מסוימים כמו שרתים.

אם אתם לא יודעים אסינכרוניות וקולבקים היטב – זה הזמן לבצע חזרה על כך. קולבקים הם לא ייחודיים ל-Node.js ולא נלמדים בספר זה אלא נלמדים בספרים המלמדים ג'אווהסקריפט מאפס. בהמשך הפרק נלמד דרכים נוחות יותר לכתיבת קוד אסינכרוני, עם פרומיסים או עם Async-Await, שנוחות בדיוק כמו קוד אסינכרוני. אבל כך או כך – Node.js לא נכתב כקוד סינכרוני.

תרגיל:

צרו מחרוזת טקסט רנדומלית עם:

```
const randomString = Math.random().toString(36).substring(7);
```

בעזרת פונקציה סינכרונית, צרו תיקייה עם שם רנדומלי, דווחו על היצירה שלה ואז מחקו אותה.

פתרון:

ראשית אנו צריכים לאתר את המתודות של File System שנשתמש בהן. במקרה הזה, mkdirSync שמשמשת ליצירת תיקייה ו-rmdirSync שמשמשת למחיקת תיקייה. אני יכול להתבסס על הידע המוקדם שלי או לבחון אותן בדוקומנטציה ולראות איך הן עובדות. במקרה הזה הן פשוטות ומקבלות ארגומנט אחד – שם התיקייה. כל מה שאני צריך זה לקבל את שם התיקייה ולהוסיף אותו למיקום היחסי ./ של הקובץ שלי. זה נראה כך:

```
const fs = require('fs');

const randomString = Math.random().toString(36).substring(7);

fs.mkdirSync(`./${randomString}`);

console.log(`${randomString} Directory Created!`);

fs.rmdirSync(`./${randomString}`);

console.log(`${randomString} Directory Deleted!`);
```

כדאי לשים לב שאני משתמש פה בתבנית טקסט (הגרש העקום – backtick) כדי להציב משתנה בתוך מחרוזת טקסט.

תרגיל:

בצעו את התרגיל הקודם בעזרת פונקציות אסינכרוניות.

פתרון:

מציאת הגרסאות האסינכרוניות אמורה להיות פשוטה – פשוט להסיר את ה-Sync משם הפונקציה ולחפש בדוקומנטציה או להיזכר בדוגמאות של הפרקים הקודמים. במקרה הזה אנו משתמשים בקולבקים כי מדובר בפונקציות אסינכרוניות.
כיוון שאנו חייבים לוודא שהתיקייה קיימת לפני שנמחק אותה, נבצע את המחיקה בקולבק של היצירה. כלומר ברגע שהתיקייה נוצרה, הקולבק של היצירה מופעל ורק בו אנו יכולים למחוק את מה שנוצר.

```
const fs = require('fs');

const randomString = Math.random().toString(36).substring(7);

fs.mkdir(`./${randomString}`, (err) => {

  console.log(`${randomString} Directory Created!`);

  fs.rmdir(`./${randomString}`, (err) => {
    console.log(`${randomString} Directory Deleted!`);
  });

});
```

מה שחשוב להבין הוא שבתוך הקולבק הראשון אני יודע בוודאות שהתיקייה נוצרה ואז אני יכול למחוק אותה.

```
const fs = require('fs');

const randomString = Math.random().toString(36).substring(7);

fs.mkdir(`${randomString}`, (err) => {
  console.log(`${randomString} Directory Created`);

  fs.rmdir(`./${randomString}`, (err) => {
    console.log(`${randomString} Directory Deleted`);
  });
});
```

קולבק ראשון
מופעל לאחר יצירת
התיקייה

קולבק שני
מופעל לאחר מחיקת התיקייה

מחיקת התיקייה
נעשית בתוך הקולבק
הראשון