

תודה רבה על הרכישה של הספר!

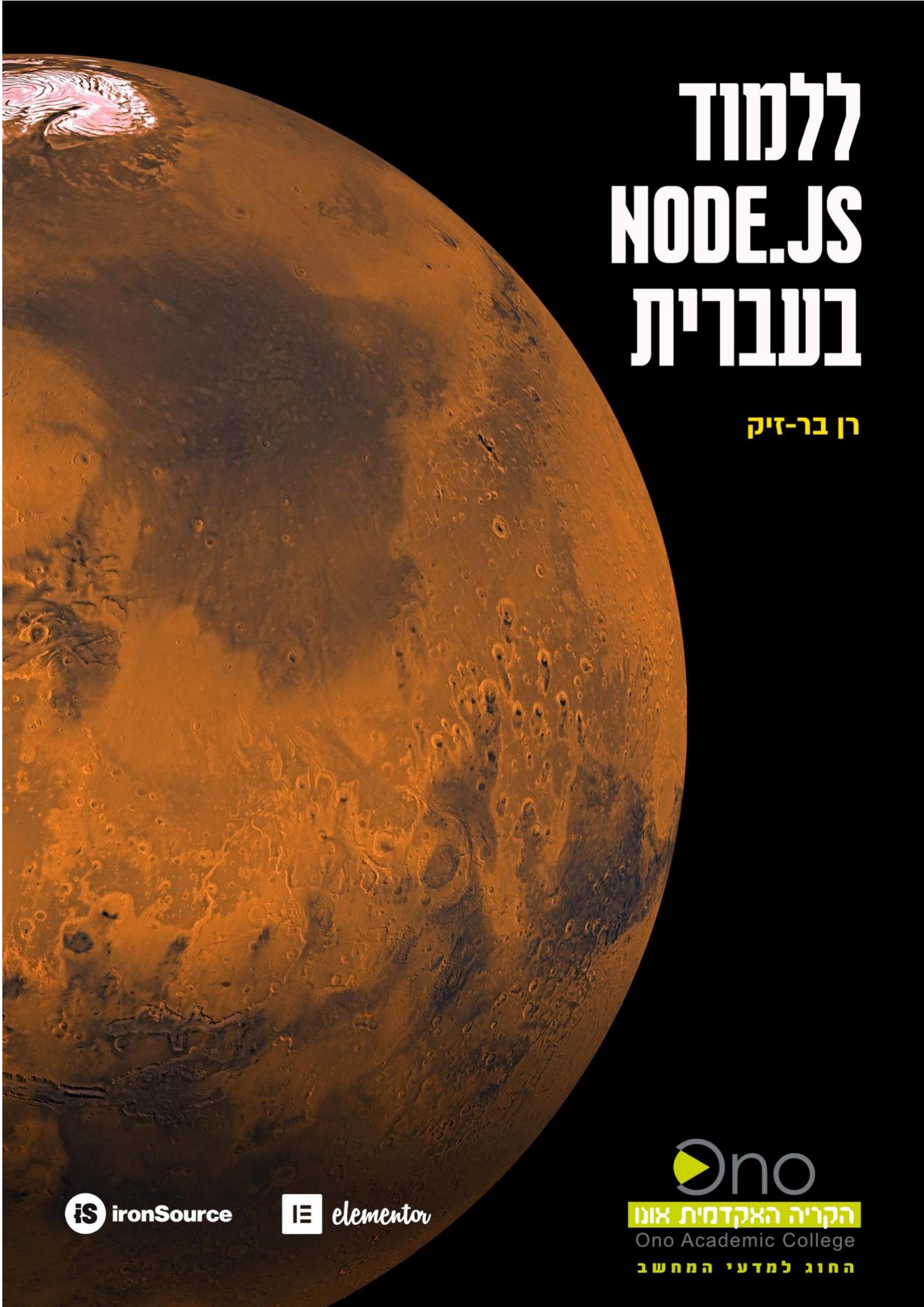
עבדתי מאד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוצריו העריכה. יותר מ-1800 אנשים תמכו בספר זהה ואיפשרו לו לצאת לאור.

הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהקינדל, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתוך תקווה שהרוכש והותמן לא ינצל את האמון שנתתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפצה שלו. אני מאמין שרוב האנשים הוגנים.

העתק זהה נמכר ל:

nivbuskila@icloud.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נჩבים פרטיו הרוכש באופן שקוֹף למשתמש. כדאי מאוד להמנע מהעתקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיהקנו את העתק שנמצא ברשותכם.



ללמוד NODE.JS בנברית

ר' בר-זיק



הקריה האקדמית אונו
Ono Academic College
החוג למדעי המחשב

לلمוד Node.js בעברית

REN BER-ZIK

מהדורה: 1.2.1



כל הזכויות שמורות © רן בר-זיק, 2023.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רישיון לא-בלודי, לא-ייחודי, אישי, בלתוי ניתוח להעברה (למעט על פי דין), ובلتוי ניתן להסבה לעשות שימוש אישי בספר זה לצרכים לימודיים בלבד.

אסור לך להעתיק את הספר, לשכפל אותו, לצור יצירות נגזרות ממנו או לפרסם אותו בכל צורה אחרת.

מותר לך לצלט קטעים קצרים מהספר במסגרת הגנת שימוש הוגן, כלומר פסקה או שתים, כאשר אתה מפנה למקור ומציר את רן בר-זיק כמחבר הספר.

הדוגמאות המובאות בספר זה הן בבעלות של רן בר-זיק, ואסור לך להשתמש בהן בתחום תוכנות שתפתח. אם אתה רוצה להכניס אותן לפרויקט שלך,שלח מייל ונדבר על זה.

עריכה לשונית: יעל ניר
הגהה: חנן קפלן
עיצוב הספר והכricaה: טל סולומון ורדי (tsv.co.il)

הפקה: כריכה – סוכנות לסופרים

www.kricha.co.il



תוכן ענייני

10.....	על הספר
10.....	על המונחים בעברית.....
12.....	על המחבר
13.....	על העורכים הטכניים
13.....	בנג'מין גרייבנברום
13.....	gil pinck.....
14.....	על החברות התומכות.....
14.....	אלמנטור
15.....	ironSource
16.....	הקדמה – מה זה ואיך זה התחיל
18.....	דרך הלמידה
19.....	התקנת סביבת עבודה ועבודה עם טרמינל
22.....	התקנה על חלונות.....
25.....	התקנה על מק
25.....	התקנה על לינוקס
26.....	תקלות נפוצות.....
26.....	כתיבת התוכנה הראשונה
32.....	Require ומודולים
35.....	מודולים של ECMAScript
40.....	היכרות עם הדוקומנטציה של Node.js
48.....	גרסאות סינכרוניות למתחמות אסינכרוניות
55.....	package.json – הכרה ראשונית והפעלה של NPM
56.....	יצירת JSON package לפרויקט שלנו
58.....	התקנת המודול הראשון
60.....	שימוש במודול חיצוני
63.....	יצירת פרויקט npm התומך ב- ECMAScript modules
66.....	עבודה אסינכרונית ומעבר מקבוקים לפורמייסטים ול- async/await

מודולים ב-<i>Node.js</i>	שתוכנים בפרומייסים באופן טבעי
70	מודולים ב- <i>Node.js</i>
73	AIRROUIM
76	כיבוי מאזין
77	הפעלת יותר מAIRROUIM אחד
79	הצמדת כמה פונקציות מאזינות לAIRROUIM אחד
81	העברה נתונים באירועים
85	יצירת שירות HTTP בסיסי
94	ה-<i>loop</i> של <i>Event Loop</i>
94	מתודות הטיימרים
94	תורץ בשאנו אומר לך – setTimeout
95	תורץ מיד עם קולבק – setImmediate
95	הלואה הקבועה setInterval
96	תור הקריאה
105	Streams
107	סוגי הストreamים השונים
108	ストreamים טרנספורמציה
109	אירועים בסטרימים
116	אריזת הקוד שלנו במודול
123	קביעת גרסאות
124	גרסאות סמנטיות
127	קביעת גרסאות סמנטיות ב- <i>nodejs</i>
131	התקינה גלובלית ו-CLI
135	כתיבת <i>bin</i> והتمמשקות עם ה- <i>CLI</i>
145	Sockets
154	קריאה משאבים באמצעות מודול <i>path</i>
159	package.json scripts
161	סקרייפטים עם שמות
162	משתני סביבה
163	קביעת משתנה סביבה דרך הסкриיפט
163	קביעת משתנה סביבה דרך הגדרות מערכת הפעלה
165	קביעת משתנה סביבה דרך קובץ
166	dev dependencies

171.....	אקספרס
173.....	טיפול במתודות של בקשות HTTP
178.....	ראוטינג.....
181.....	MiddleWare
183.....	URL דינמי
186.....	תבניות
190.....	חיבור ל-MySQL
191.....	חיבור ראשוני
193.....	שאילתת בסיסית
194.....	המרת הקוד לעובדה עם פרומיסים ולא עם קולבקים
196.....	Prepared Statement
199.....	עליה לפורודקشن
199.....	עליה לפורודקشن עם שרת
201.....	עליה לפורודקشن בענן
204.....	סיכום
204.....	MITAPIים
205.....	קבוצות דיוון
205.....	גיטהאב
205.....	התנדבות בעמותות ובמידמים
206.....	נספח: בדיקות אוטומטיות ב-<i>Node.js</i>
206.....	מה זה בדיקות אוטומטיות?
209.....	בדיקות אוטומטיות
210.....	Mocha
213	describe
214	it
215	מחדור חיים
218	מבנה בדיקה
219.....	פרימורק בדיקות
220	assert.ok(value)
220	assert.notStrictEqual(actual, expected) assert.strictEqual(actual, expected)
221	assert.notDeepStrictEqual(actual, expected) assert.deepStrictEqual(actual, expected)
222	assert.throws(fn)
222.....	ספריות נוספת

223	ספריות mock
227	mock(obj)
228	בדיקות עם קריאות http
230.....	סוגי בדיקות
230	בדיקות ייחוד
231	בדיקות קומפוננטה
233.....	סוגים נוספים של בדיקות
233	eslint
235	npm audit
237.....	איך יצא לי מזה?
238.....	שינויים בין גרסה 1.1.0 לגרסה 1.2.0

על הספר

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

לימוד `Node.js` מחייב הכרה עמוקה עם שפת ג'אוوهסקריפט. בספר הקודם, "لימוד ג'אוوهסקריפט בעברית", לימדתי ג'אוوهסקריפט ברמה המספיקה להתחלה הקריאה בספר זה. הספר מלמד `Node.js` ומתחילה במבנה סביבת העבודה והתקנת הפלטפורמה. הוא ממשיך בהקנית העקרונות החשובים לפלטפורמה זו: איך בונים מודול בסיסי בשפה, איך משתמשים במודולים אחרים. אנו מסקרים גם אספקטים מתקדמים החשובים להבנה عمוקה של הפלטפורמה: סטרימים, סוקטים ובנייה CLI. בספר יש פרק ארוך ונכבד המלמד על אקספרס, המודול הפופולרי לבניית שרת רשת. אנו לומדים גם על הعلاאת האפליקציה שלנו לענן באמצעות "הרוקו". בסיוםו של כל פרק רלוונטי יש תרגילים והסבירים מפורטים הכוללים גם שרטוטים.

הספר מיועד לכל מתכנת ג'אוوهסקריפט שמעוניין ללימוד על העולם המופלא של `Node.js` ולמתכנתים המכירים את `Node.js` אך זוקים לחיזוק או לתגובה של הידע שלהם באספקטים מסוימים.

על המונחים בעברית

אני כותב בעברית על טכנולוגיה ותוכנות כבר יותר מעשור והדילמה "באילו מונחים בעברית להשתמש" מלואה אותו תמיד. מצד אחד, האקדמיה לשון העברית מספקת לנו מונחים רבים בעברית. מצד שני, בתעשייה ההייטק, שמננה אני מגיע, איש לא משתמש ברבים מהמונחים האלה. אם הגיעו לראיון עבודה ותגידו: "במפגש המתכנתים האחרון שמעתי על דרך חדשה לבצע הידור שבודק הוצאות במנשך הבוחות", סביר להניח שלא תקבלו את העבודה. אבל אם תגידו "במיוחד הוצאות מבוססת הבוחות", סביר להניח שלא תקבלו את העבודה. אבל אם

"פרומיסים" – יבינו על מה אתם מדברים. זו הסיבה שלא תמצאו מילים כמו "הודור", "מחלקה" או "מרשתת" אלא "קמפול", "קלאס" ו"אינטראנט". 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 להכנסו אותו לחיפושים שלכם בגוגל.

חשוב לציין שאיני בכלל לאקדמיה ללשון ושחק מונחים שלא אכן נכנסו לשפה המדוברת במרכז הטכנולוגיה השונים (למשל: קובץ או מסד נתונים), אבל בכל מקום שהיתה לי ברירה בין להיות מובן לבין לעמוד בຄלי הלשון, העדפתה להיות מובן.

על המחבר

REN BAR-ZIK הוא מפתח תוכנה משנת 1996 ב מגוון שפות ופלטפורמות ועובד כמפתח בכיר במרכז פיתוח של חברות רב-לאומיות, HPE ו עד Verizon, שם הוא מפתח בטכניקות מתקדמות הן בצד הלקוח הן בצד השרת, ושם דגש על בניית תשתיות פיתוח נכונה, על שימוש ב-CD\OI וכמו כן על אבטחת מידע.

נוסף על עבודתו כמפתח במשרה מלאה, REN הוא עיתונאי ב"הארץ" במדור המחשבים, שם הוא מסקר נושאים הקשורים לテכנולוגיה ולאבטחת מידע וכן בענין אינטרנט ורשתות.

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

REN הוא מחבר הספרים "לימוד ג'אווה סקריפט בעברית", "לימוד ריאקט בעברית", "לימוד MySQL בעברית" ו "لימוד פיתוח ווב מעשי בעברית".

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

על העורכים הטכניים

בנג'מין גריינבאום

בנג'מין גריינבאום הוא מתכנת מנוסה, מומחה לג'אווהסקריפט בעל רקע עשיר של עבודה במגוון חברות רב-לאומיות ובמגוון תפקידים וborgר 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 מוביילים בתחום בנושא ג'אווהסקריפט, כגון JS Know You Don't Know Exploring ES6 ו-Gil Fink.

gil fink

gil fink הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert . הוא ממייסד Microsoft Developer Technologies MVP . spartXsys. כיום הוא מייעץ לחברות ולארגוני שונים, שם הוא מסייע לפיתוח פתרונות מבוססי אינטרנט ו-SPAs. הוא עורך הרצאות וסדנאות לייחדים ולחברים המעוניינים להתחמות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט Pro Single Page Application (Microsoft Official Course MOC) .angularUP (Apress) "Development לשוטף בארגון הכנס הבינלאומי Development".
לפרטים נוספים על gil:
<http://www.gilfink.net>

על החברות התומכות

אלמנטור

אלמנטור מפתחת פלטפורמת קוד פתוח לבניית אתרים שימושה את הדרך בה בונים אתרים אינטרנט בשוק המkteבי. אלמנטור מעניק למשתמשים את החופש ליצור עמודי אינטרנט ללא צורך בקוד ולפתחים את החירות לדחוף את הגבולות, לרענן ולהרחיב את המערכת בצורה קלה ומהירה באמצעות API ייחודי למפתחים, ובכך לחסוך זמן פיתוח ולהיות יעילים ורוחניים.

עם מיליוןיים אתרים הפעילים על אלמנטור וצמיחה חודשית מדיהימה, התגבשה סביבה של פלטפורמה קהילתית חזקה המונה מאות אלפי חברים, מפתחים, מושוקים ומשתמשים, המקיים מיטאים בכל רחבי העולם. מידי יום האלמנטוריסטים מייצרים וצורכים אלפי שעות של הדרוכות, סרטים השראה ובלוגים עמוקים, ומפתחים תורמים קוד ורעיון נאות באמצעות GitHub. האקויסיטם המkteבי של אלמנטור מתפתח ללא הפסקה והוא אוצר המוסף ומעשיר את יכולות של כל יוצר אינטרנט.

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

ironSource

חברת ironSource, הנחשבת לחברת מובילה בכל הקשור לモンטיזציה באפליקציות ולפלטפורמות פרסומם בויאדו, וחולשת על יותר ממיליארד וחצי שחקנים ברחבי העולם, ה奏פים בפרסומות על גבי תשתיות החברה. החברה עוזרת למפתחים לקחת את האפליקציות שלהם לשלב הבא, זאת גם בזכות רשות הויאדו העצומה שלה - והמספרים מדברים בעד עצמו, עם יותר מ-80,000 אפליקציות המשמשות בטכנולוגיות של החברה כדי לפתח את העסוק שלהן.

הקדמה – מה זה ואיך זה התחיל

Node.js הופיע בשנת 2009. מדובר בסביבת הריצה של ג'אווחסקרייפט בסביבת שרת. סביבת הריצה זו בנויה כולה על מנוע הריצה של כרום V8. מדובר במנוע חזק ומהיר מאוד שימושים בו בכרום. ב-node.js ההרצה היא מחוץ לדפדפן, אך מנוע V8 מאפשר לג'אווחסקרייפט לroz מהר מאוד ויעיל מאוד. מרכיב נוסף של Node.js היא ספריית `uv.lib`, הכתובת ב-C ומאפשרת הריצה של פועלות קלט ופלט במהירות רבה.

מתכנת בשם ריאן דאל רצה לבנות סמן התקדמות של תעינות קובץ. הוא ניסה לעשות זאת בשיטות הקודמים, ובראשם Apache, אך לא הצליח לעשות כן בגלל בעיות ביצועים. הוא החליט לבנות שרת מבוסס על V8 המהיר, עם דרכים פשוטות לבצע קלט ופלט למערכת הפעלה ועם ג'אווחסקרייפט.

בניגוד לסביבות הריצה אחרות, שבהן משתמש נדרש לנצל את התהיליכים של המעבד, ב-node.js הקוד של המשתמש רץ על תהליך אחד של המעבד ואינו חוסם אותו כאשר הוא מחייב נתונים מסוימים. תהיליכים נוספים מנהלים אוטומטית דרך ספריית `uv.lib`. דרך הפעלה זו מאפשרת-node.js לעבוד מהר מאוד עם פלט וקלט, כיוון שם היא מבצעת בקשה כלשהי לשרת אחר, מערכת קבצים או מסד נתונים, התהליך אינו נחסם אלא הבקשת נשלחת ו-node.js ממשיכה לroz. זה מאפשר בಗל האсинכורניות המובנה שיש/node.js והואף את סביבת הריצה זו לטובה מאוד בקלט ופלט.

dal הציג את התוצאה בנובמבר 2009 וכמה חודשים לאחר מכן נוצר וקח, מאגר המודולים החופשיים של Node.js, שבו יש מודולים שככל מתכנת-node.js יכול להשתמש בהם בקלות. סביבת הריצה של node.js יכולה לזרוץ בכל סביבת שרת שהיא, גם בשרת מבוסס על חלונות וגם בשרת מבוסס על לינוקס. זה אומר בעצמך, במילויים אחרים, שאתה רוצה 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 כלם באופן זהה. לא תמיד זה קורה, אבל זו הכוונה ולמרות שיש הבדלים, רובם מטופלים.

node.js פופולרית להדרים. בשעת כתיבת ספר זה (יוני 2019), יש יותר ממאה אלף חבילות תוכנה בקוד פתוח שמיינן למשתמשים-node.js לשימושים שונים. שרתים רבים נכתבו

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

כיום מי שМОBILE את Node.js הוא מוסד ללא כוונת רוח שנקרוא OpenJS Foundation – מוסד שבנוו על פיו עקרון "הממשל הפתוח" וכל אחד שיש לו מספיק רצון יכול להשתתף בדיונים ולהשפע על ההתפתחות העתידית של סביבת הריצה.

Node.js היא לא שפה, השפה היא ג'אוوهסקרייפט. Node.js היא סביבת הריצה. קל לכל מתכנת או מתכנתת ג'אוوهסקרייפט לעבוד היטב עם Node.js. ספר זה אינו מלמד ג'אוوهסקרייפט ואני יוצאת מנקודת הנחה שהקוראים מכירים היטב ג'אוوهסקרייפט ובדגש על ג'אוوهסקרייפט מודרני ואסינכורי. אם איןכם מכירים היטב את השפה הזאת, אני ממליץ לכם לקרוא את ספרי הקודם "למידה ג'אווהסקרייפט בעברית", שייצא בהוצאת הקרייה האקדמית אוננו. 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 שנדאי לזכור ב-node.js הוא שעושר הספריות העצום שלה עצמן חוסך המון זמן הכתיבה. אנו נלמד מה למשל איך מקימים שרת HTTP, אך הסיכון שתctrco לעשות את זה בחיים האמתיים הוא אפסי, כיון שהמודול הפופולרי Express משמש את רוב המתכנתים ליצור שרת HTTP. כמו כן תוכלנו להשתמש בידע שתלמידו בספר הזה כדי להוסיף למודולים קיימים או לכתב אפליקציות של ממש או שירותים של ממש שימושים במודולים של Node.js. ברגע שתבינו איך עובדים עם ג'אווהסקרייפט על סביבת הריצה הזאת – השמיים הם הגבול. כאמור, משתמשים ב-node.js בכל מקום. גם במקומות שבהם כתבים בעיקר בשפות תכנות אחרות, כיון שהכח של Node.js הוא ביכולת שלה לפעול בכל מקום, גם במשימות תחזקה וגם במשימות של אבטחת מידע.

דרך הלמידה

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

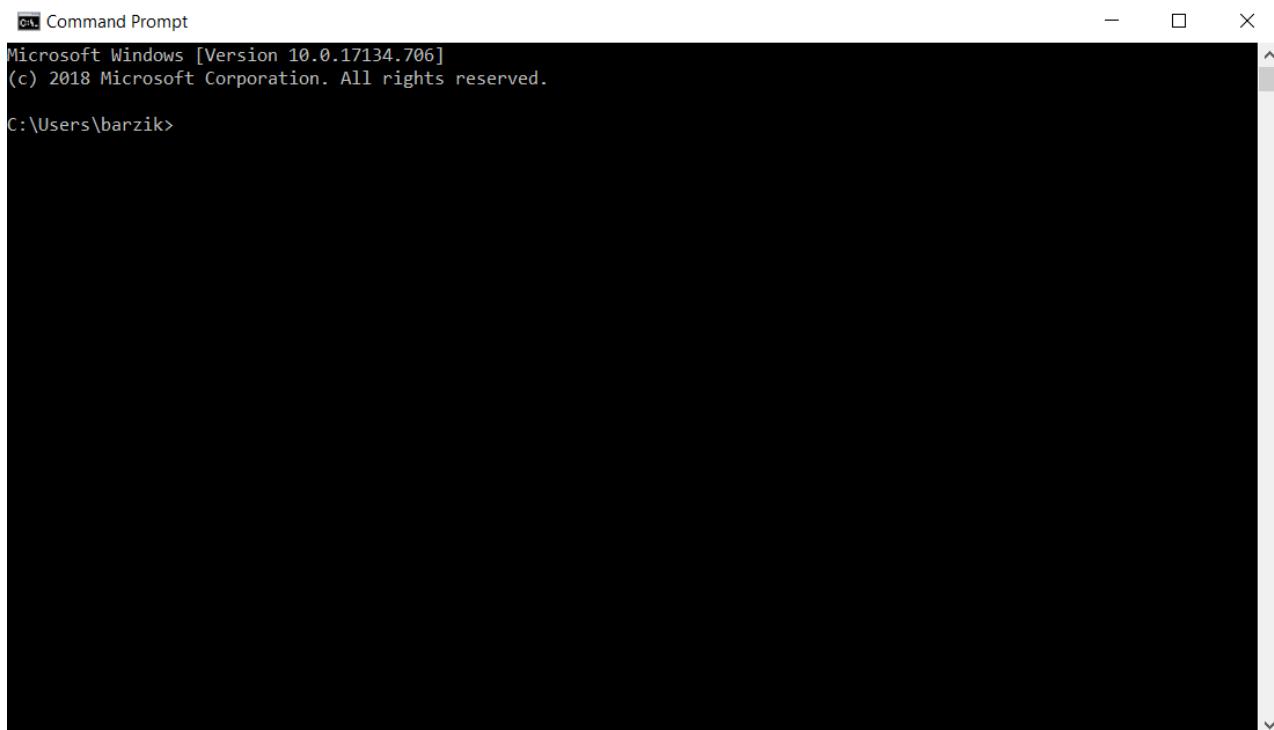
שפת תוכנה או אפילו סביבת הריצה לומדים דרך הידים. בעבודה קשה. לא תוכלו ללימוד Node.js ללא לנוכח הידים וכתיבת אמיתי. הספר הוא כלי עזר, הוא לא יחליף את הקלדה שלכם. בדרך כלל הקושי האמיתי הוא במבנה סביבת עבודה יציבה וטובה, لكن הפרק הראשון שעוסק בהתקנת סביבת עבודה הוא קריטי.

לא תמיד כל הסבר המופיע בספר הוא כולל או מתאים. אם קראתם את הפרק פעם ופעמיים ושלוש פעמים ועודין לא הבנתם – הבעיה לא בכם אלא בהסביר. לא להתייחס – פה נדי להתייעץ בקהילות של ג'אווהסקריפט ויש לא מעט כאלה בפייסבוק ובמקומות אחרים. גם חיפוש בגוגל לפעמים יכול להוציא אתכם מבוץ אמיתי. נתקעתם? אל תהייאשו – 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 ישב. 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, כתעי קוד או שגיאות שונות ולקבל פתרון כמעט מיידי. כדאי להזהר עם חלק מהתשובות וגם לא להסתמך על הבינה המלאכותית יותר מדי, כיון שהמטרה היא ללימוד Node.js. עם הידע שלכם ב-node.js תוכלו לudy 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 הריצה, ונכדי שהיא תוכל לתוכנן אותה על המחשב, ממש כמו כל תוכנה אחרת. מה שההתקנה הזאת עשוה הוא פשוט למדי – היא מאפשרת לנו להפעיל את Node.js כמו כל תוכנה אחרת. זה הכל. אנו רגילים לפתח תוכנות באמצעות איקונונים, אבל חלק מהתוכנות עובדות באמצעות הטרמינל. מה זה טרמינל? מקום שבו אתם יכולים להקליד פקודות. הוא קיים בכל מערכת הפעלה. בחלונות מגעאים לטרמינל באמצעות לחיצה על הזוכנית המגדלת (בחולונות 10) והקלדה של cmd – ראשית תיבות של command. נגעים לחלון שנראה כך:

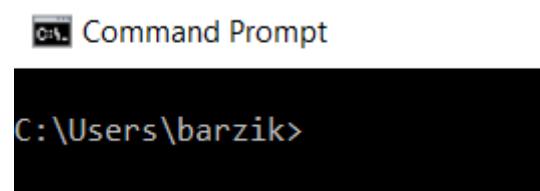


פתחו את החלון הזה, הקלידו notepad ולחצו על enter. ייפתח ה-notepad של חלונות. ברכותי! הפעלתם תוכנה באמצעות שורת הפקודה. הקלידו calc ולחצו על enter, תוכנת המחשבון תיפתח. זו עוד תוכנה שהפעלתם באמצעות מסך הפקודה. הממשק הזה, או הטרמינל בשפת העם, הוא סביבת העבודה של Node.js. מפעילים את Node.js באמצעות הטרמינל. זה בדיק מה שקרה בשורת "אמת". זכרו שרשרת בסופה של דבר הוא מחשב – ייתכן שמחשב לא מערכת הפעלה גרפית אלא רק עם טרמינל – אבל מחשב שմבוסס על חלונות או על לינוקס. ייתכן שהשרת חזק בהרבה מהמחשב הביתי שלכם – אבל עדין מדובר במחשב לכל דבר. כאמור, Node.js רצה היטב

על שירותים מבוססי חלונות ועל שירותי מבוססי לינוקס. לצורך העניין, המחשב שלכם עכשו הוא שרת.

צריך להכיר מעט את משק הפקודה של הטרמינל ולהתמצא בו. כיוון שהטרמינלים שונים בין חלונות לlinpus, יש שוני קטן בין הפקודות. כיוון שחלונות היא מערכת הפעלה הנפוצה, ומשתמשי linpus בדרך כלל מ信נים 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 במקום של המשתמש שלי. כך למשל, אם אני נכנס ל-pwd במחשב שלי – אני רואה את המיקום שלי:



```
C:\ Command Prompt
C:\Users\barzik>
```

אפשר לראות שאנו נמצא בדיסק C, בתיקיית Users ובתת התיקייה barzik, שהוא שם המשתמש שלי בחלונות. אם אני אפתח את סיר הקבצים, אני יוכל לנוט לתיקיה זו. הטרמינל הוא פשוט דרך נוספת לשוטט במחשב ולפעול בו – דרך שהיא לא גרפית, אבל כל מה שאנו יכול לעשות במשק 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 זו תהיה זהה לחנותין לרשימת הקבצים שאנו רואה בסיר הקבצים כשהיינו נכנס לאותו מקום. אם אצור קובץ או תיקייה בסיר הקבצים ואקליד שוב dir בטרמינל כשאני באותו המיקום של סיר הקבצים, יוכל לראות את הקובץ או את התיקייה בטרמינל.

כדי להכנס 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 להקייף אותו במירכאות. אם אני משתמש ב-TAB הוא יעשה זאת זה עבורו.

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

אם אני רוצה לחזור לאחור, אני אכתוב `cd ..` שני הנקודות יעלו אותי לתיקיית האב. אם אכתוב `cd ../../` אני יוכל 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 זאת, אבל אנו לא נתיחס לכך בספר זהה.

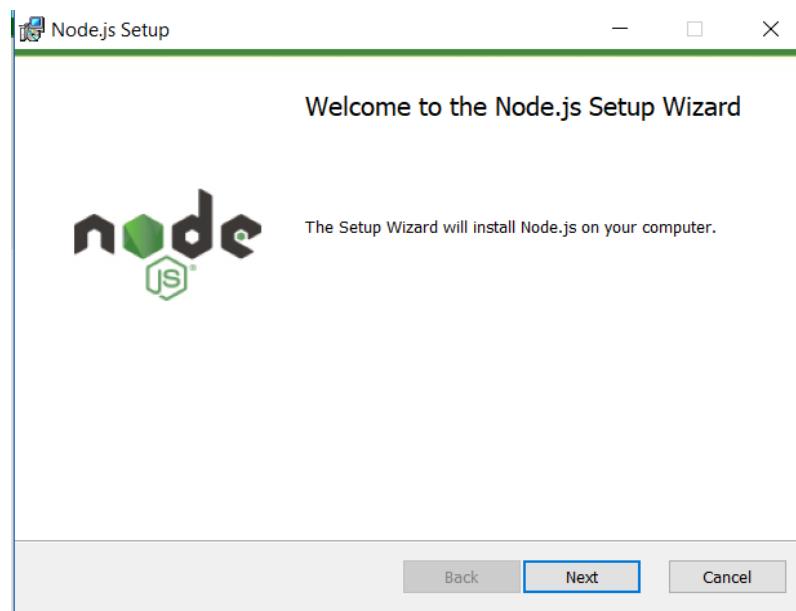
פקודה בLINQ/S/מק	פקודה בחלונות	פקודה
<code>ls -al</code>	<code>dir</code>	הציג את רשימת הקבצים והתיקיות בתיקייה
<code>cd</code>	<code>cd</code>	עברו לתיקייה אחרת
<code>exit</code>	<code>exit</code>	יציאה מהטרמינל
<code>clear</code>	<code>cls</code>	ניקוי המסך

לאחר שנאנו יודעים איך לעבוד עם הטרמינל, נתקן את `jsNode`. ההתקנה שונה במערכות הפעלה שונות אבל בכלל היא קלה למד. בחרו את מערכת הפעלה שלכם ותתקינו את `jsNode` לפי ההוראות.

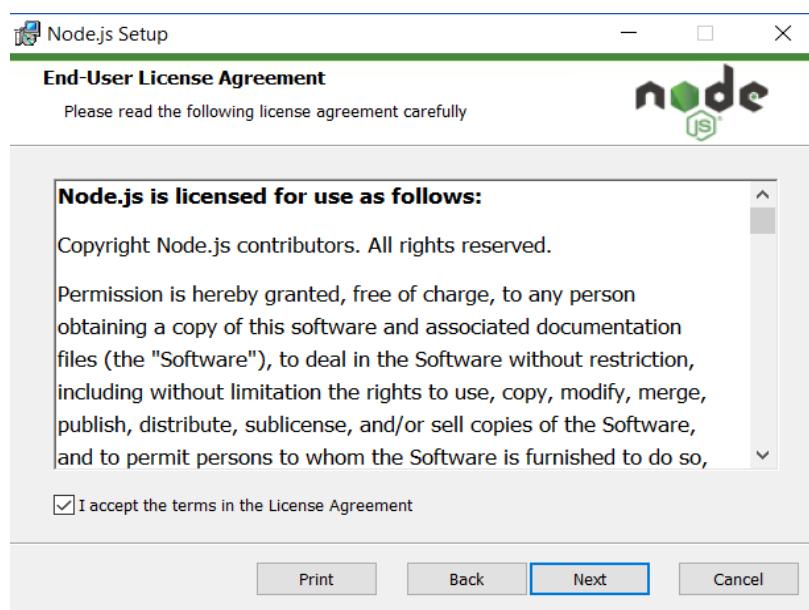
התקנה על חלונות

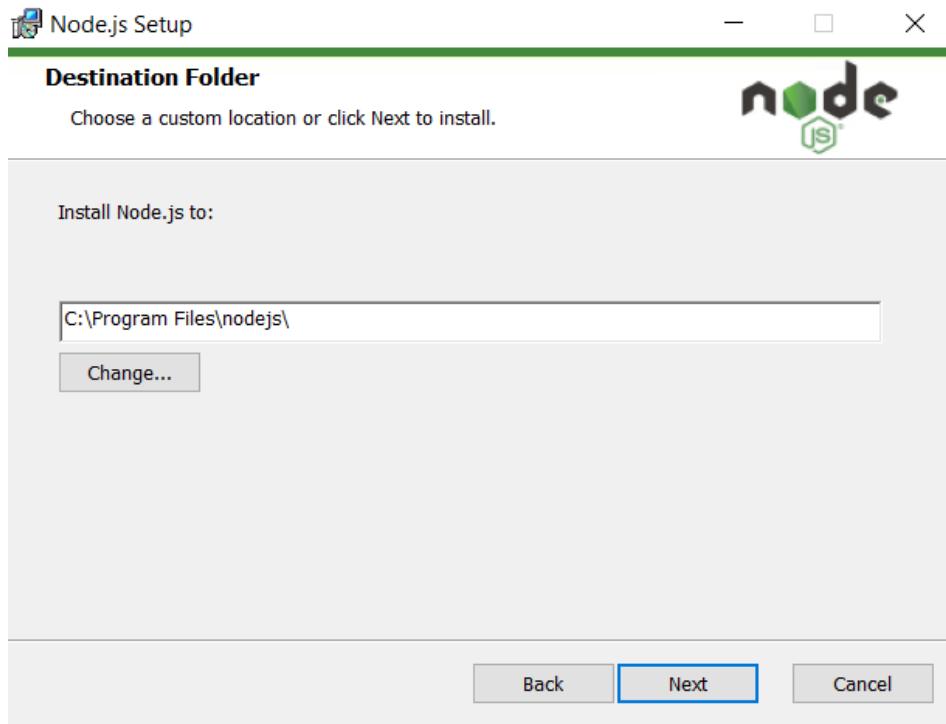
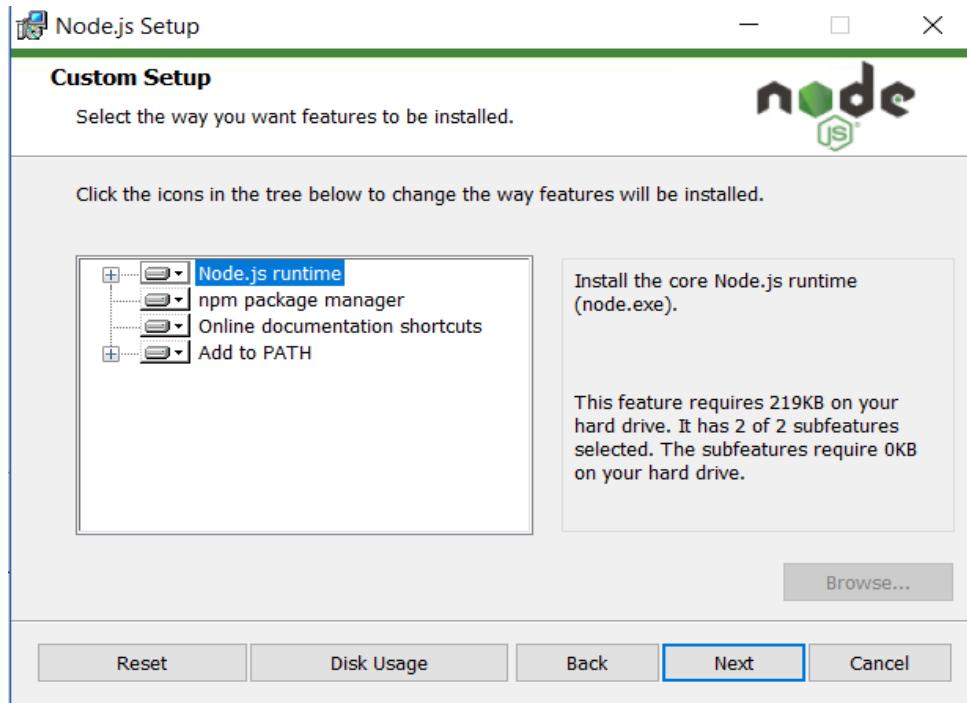
ההתקנה של Node.js על Windows היא פשוטה מאוד. נקליד בוגול <https://nodejs.org/en/download/>

- אנו נבחר בגרסת LTS – ראש תיבות של "גראף לטוח אורך", ונבחר במערכת הפעלה שלנו – אם מדובר בחלונות, יש לנו installer נוח. מורידים, לוחצים על התוכנה שיורדת:

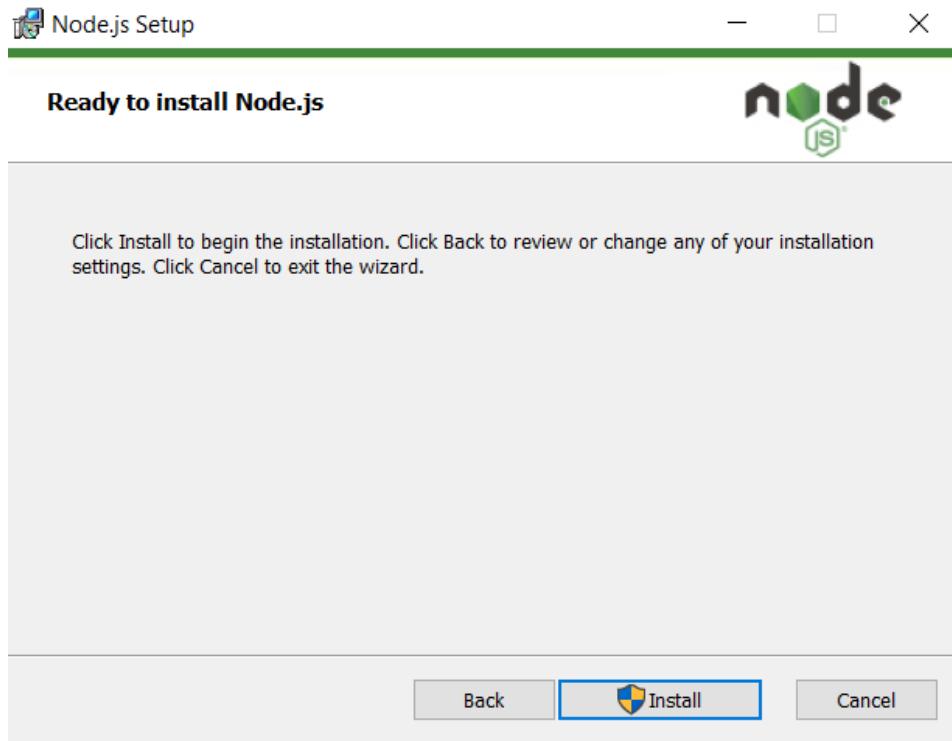


מקבלים את התנאים:

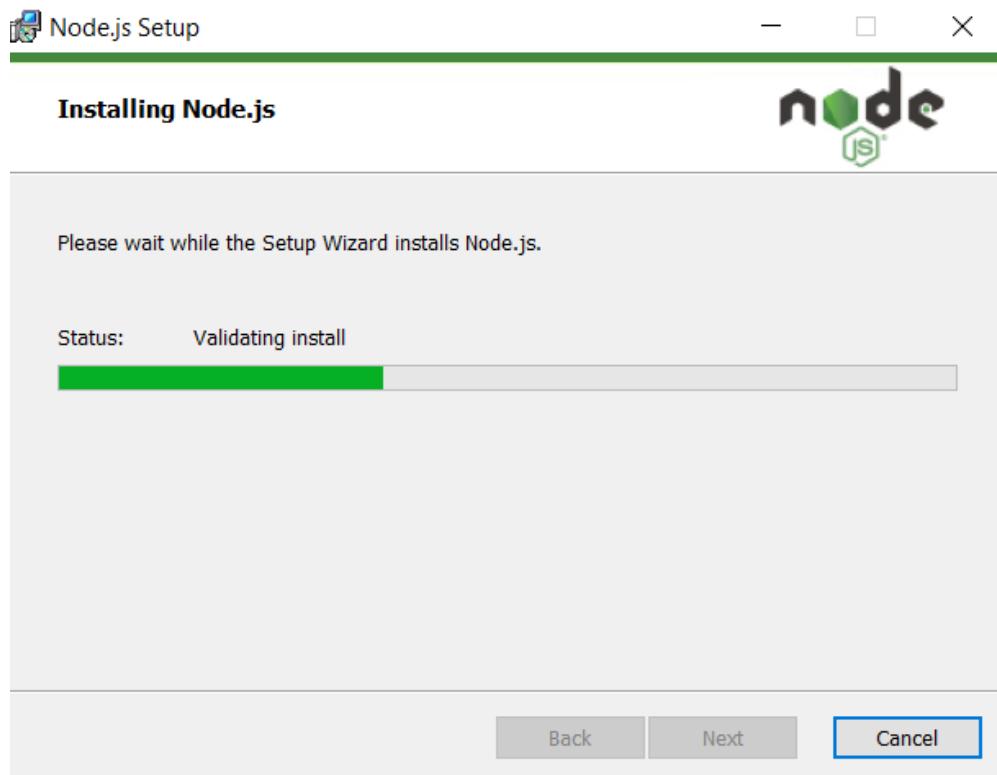


לוחצים על **:next**לשוב על **:next**

לחיצה על Install תתקין לבסוף את התוכנה:



כל מה שנוצר הוא לחכות לסיום ההתקינה:



אחרי שההתקנה הושלמה, נפתח את הטרמינל שלנו (אם היה לנו טרמינל לפני ש-node.js הותקנה, נדרש לסגור ולפתחו אותו מחדש) ונקליד `v - node`. אם הכל 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 בוגול `Node.js Download` או ניכנס אל:

<https://nodejs.org/en/download/>

אנו נבחר בגרסה LTS – ראשיתיבות של "גרסה לטוווח ארוך", ונבחר במק – ייד קובץ `npm` שהוא אפשר להתקין כמו כל תוכנה אחרת בהנחה שהמחשב שלכם הוא לא מחשב ארגוני שמנוע 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, וכך ישיה מותקנת):

`brew install node`

כך או אחרית, לאחר ההתקנה, כניסה לטרמינל והקלדה של `v - node` תראה לכם את מספר הגרסה בדיקן כמו בחולנות.

התקנה על לינוקס

אם אתם משתמשים בדبيان, אז בדרך כלל ברוב ההפצאות `sudo apt-get install node` יטפל בהתקנה, אך אתם עלולים להתקין גרסה ישנה של `Node.js` וזה עלול להוות בעיה. למורת הפיתוי, הייכנסו אל הקישור וקראו לפניו ההתקנה את המדריך המלא לכל ההפצאות של לינוקס, שמסביר על ההתקנות.

<https://nodejs.org/en/download/package-manager/>

אני יוצא מנקודת הנחה שמשתמשים בlienoks הם מיומנים בהרבה ממשתמשי חלונות וידעים להתקין חבילת תוכנה ללא הסברים נוספים. כך או אחרת – לאחר ההתקנה, כניסה לטרמינל והקלדה של - node תראה לכם את מספר הגרסה בדיקן 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 וליאוש 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 מהודעתה השגיאה וחפשו בראשת – סביר מאוד להניח שהם אחר נתקל באותה בעיה. בדרך כלל מדובר בבעיית אינטרנט של מחשבים ארגוניים שעובדים מאחורי רשת ארגונית. בדף זה יש הסבר על תקלות נפוצות ופתרונות:

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

אל תתיאשו אם זה קורה, נסו שוב ושוב והתעקשו עד שהזה יצליח. אני מבטיח לכם Sh-Node.js שווה את זה.

כתיבת התוכנה הראשונה

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

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

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

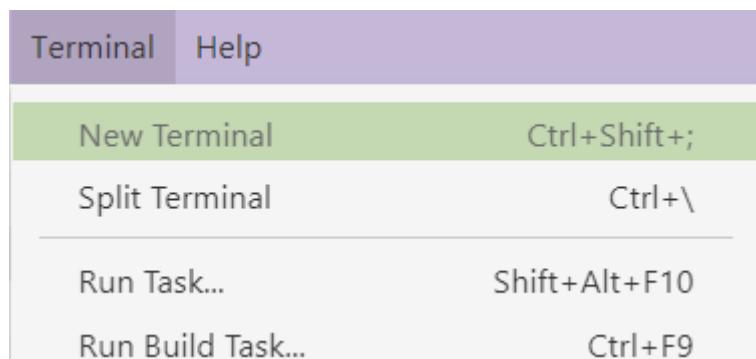
```
console.log('Hello World!');
```

נשמר ואז נחזיר לטרמינל ונכתב node hello.js או נראה שמודפס לנו המשפט Hello World!

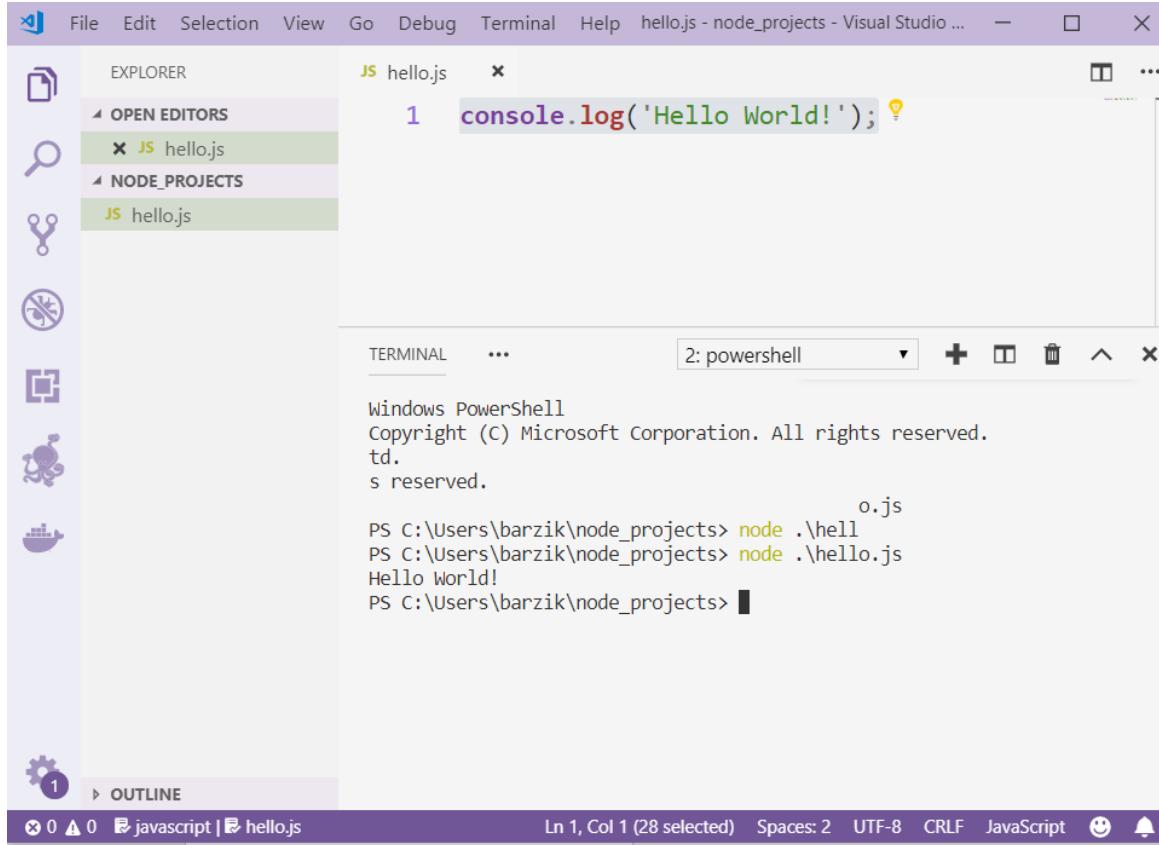
```
C:\Users\barzik\node_projects>node hello.js
Hello World!
```

איזהEIF! כתבתם את תוכנית ה-`Node.js` הראשונה שלכם!
 חשוב! אני יוצא מנקודת הנחה שאתם יודעים ג'אויסקייפ וידעים מה זה `console.log` ומה זה IDE ואפיו נזכר מותקן לכם `Visual Studio Code`, `Atom`, `WebStorm` או כל IDE אחר על המחשב. אם זה נשמע לכם כמו סיינט – אתם חייבים ללמידה ג'אויסקייפ על מנת להתקדם בספר זה.

הערה חשובה נוספת: ב-`Visual Studio Code` וגם בסביבות העבודה אחרת הטרמינל מובנה ב-IDE. חפשו בתפריט העליון `Terminal` ולחצו עליו. בחרו `New Terminal`. יפתח לכם בתחום המסך טרמינל במיקום של הקבצים שלכם.



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

**תרגיל:**

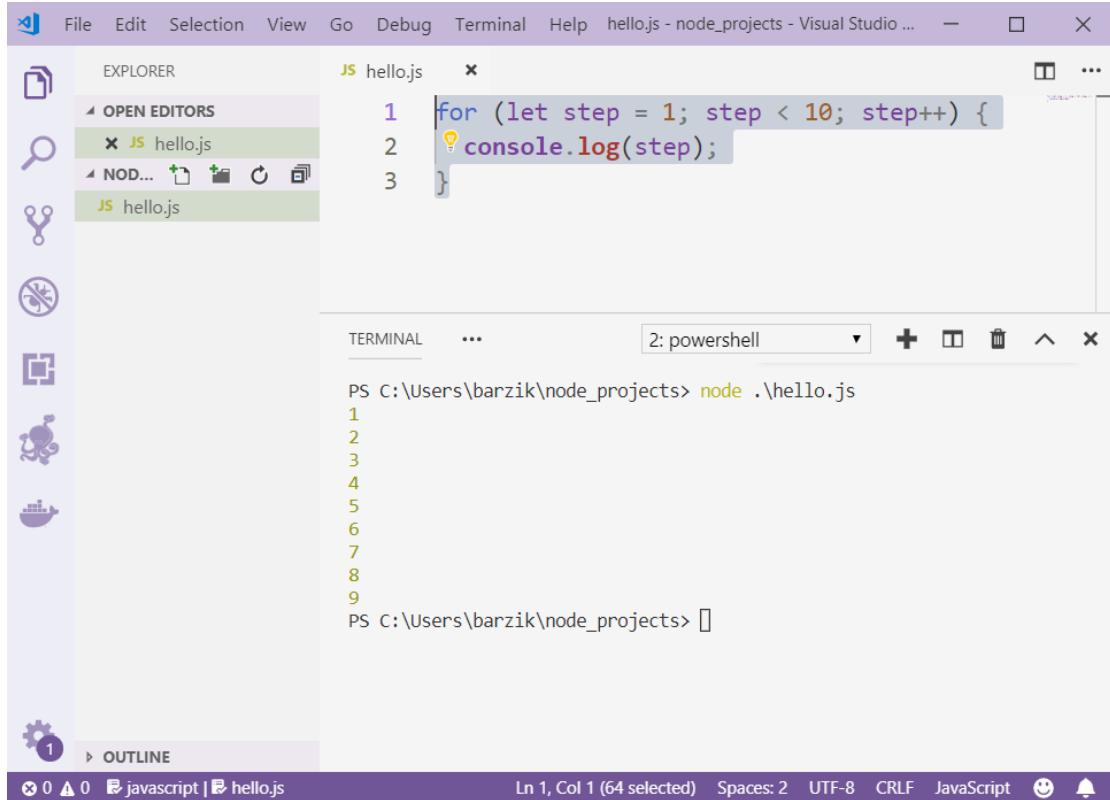
כתבו לולאה שרצה מ-1 עד 10 ומדפיסה את המספר של הלולאה. הריצו אותה ב-node.js.

פתרונות:

בתיקיית העבודה שלי אני יוצר קובץ בשם hello.js. בתוכו אני כותב ג'אווהסקריפט רגיל לחולוטין של לולאה.

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

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



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

תרגיל:

כתבו קוד שזרוק הערת שגיאה. הריצו את הקוד.

פתרון:

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

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

אני נכנס למקום התיקייה, באמצעות הטרמינל במערכת הפעלה שלי או באמצעות הטרמינל ב-IDE שלי. אני מקפיד לוודא שאני בתיקייה שבה נמצא הקובץ וכותב node hello.js. אני רואה שגיאה וגם את ה-stack trace – השרשרת של פקודות שהובילה לשגיאת. בראשיתה אני בעצם רואה את הסיבה לבעה – 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:722:10)
    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)
D:\...\Users\barzik\node_projects\
```

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

פרק 1

REQUESTS וМОודולים REQUIRE



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 דרכיהם עיקריות בהתאם לשיטת המודולים שהמתקנת בוחר. אנו נסביר על השיטה שנקראת CommonJS – כיוון שהיא בירית המחדל של Node.js. על השיטה השנייה, ECMAScript Modules אנו נלמד בהמשך.

שיטת בירית המחדל, CommonJS עובדת באמצעות **require**. אחד ההבדלים הגדולים בין Node.js לבין ג'אווהסקריפט בסביבת דף-דף הוא ה-require. הוא כמעט ייחודי ל-node.js (יש ספירות נוספת שימושה בו, אך ללא ספק הוא סימן היכר משמעוני של node.js) ומשמש ליבור ושימוש בחבילות תוכנה. ל-node.js יש חבילות תוכנה שבאות איתו כבירית מיוחד ואנו משתמש בהן בהתחלה.

חבילת התוכנה הראשונה שאנו משתמש בה היא `fs`, שידועה גם כ-`file system`. זהה חבילת תוכנה שבאה תמיד עם Node.js (אי-אפשר להתקין את node.js בלבד) ומסייעת לנו לטפל במערכות הקבצים של המחשב. יש לה מתודות רבות שמתפעלות את מערכת הקבצים. מוגדרת אחת שאבחן בה היא `readdir` – מוגדרת שמקבלת שני פרמטרים. הראשון הוא הנתיב של התיקייה שאני רוצה לבדוק והשני הוא פונקציית קולבק. פונקציית הקולbak נקראת על ידי `readdir` בגמר הפעולה ומחזירה שני ארגומנטים – אובייקט שגייה (אם היא מתקיימת) או אובייקט תוצאה שמציג את הקבצים והתיקיות שיש לנו בנתיב.

אוצר קובץ בשם `app.js` ואנכים בו את הקוד הבא:

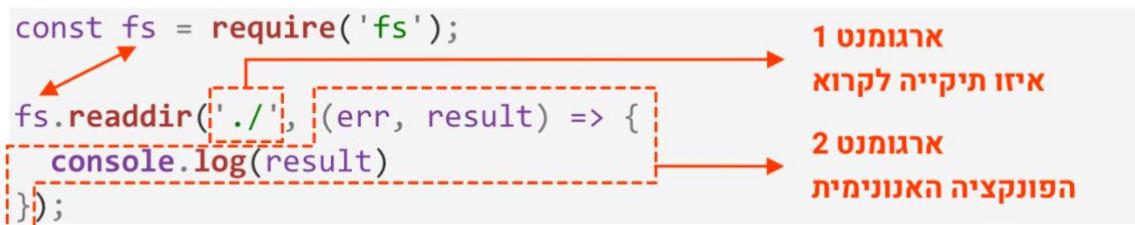
```
const fs = require('fs');

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

מה הקוד הזה בעצם אומר? הדבר שלא אמרו להיות ברור לתוכנת ג'אווהסקריפט מן השורה הוא `require`. כאן אני בעצם קורא לחבילת התוכנה או למודול שנקרא `fs`. מדובר בעצם באובייקט כמו כל אובייקט שאנו מכירים, שיש לו מוגדרת שנקראת `readdir`. זהה מוגדרת אסינכרונית שמקבלת ארגומנט ראשון של התיקייה שבה מחפשים וארוגמנט שני של קולbak. בקולbak יש שני

ארגומנטים – אובייקט שגייה וובייקט תוצאה. זהו פורמט סטנדרטי של קולבקים ב-node.js.

ואני ארכחיב על כך בפרק על פרומיםים 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.

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

```

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

באו ניצור קובץ באמצעות fs. יצירת הקובץ נעשית באמצעות המתודה writeFile. המתודה זו מקבלת שלושה ארגומנטים. הראשון הוא שם הקובץ, השני הוא תוכן הקובץ והשלישי הוא קולבק שבו מועבר אובייקט שגייה. אם אין שגייה, 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-node.js.app 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, שנונן מידע על מערכת הפעלה. מתודה אחת מתוך os היא metadat homedir – המתודה זו לא

מקבלת ארגומנטים, ומחזירה את תוקיות ה"בית" של מערכת הפעלה. אם אני למשל בחלונות, תוקיות הבית שלי היא `Users\barzik\c:`. אם אני בlionos, תוקית הבית שלי היא `/home/barzik`. אם אני כותב סקריפט של `Node.js`, אני רוצה שהוא יעבד בלי קשר למערכת הפעלה ואני לא מעוניין לדעת מה היא. שימוש ב-`os` הוא הדרכן.

כך נכתב 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 שפנות פשוט עשויה `require('os')` והשתמשתי במתודה `homedir()`. מדובר במתודה סינכרונית שלא מקבלת קולבק, אז אין בעיה מהותית להשתמש בה. `require('os')` הוא לא קסם או וודו אפל. מדובר בקבלה של מודול, וברגע שקיבנתי אותו אני יכול להשתמש בו בדיקן כמו שאנו משתמש בכל מודול אחר בג'אווהסקריפט. כך למשל, אם אני רוצה ליעיל את הקוד הנוכחי ולהוסיף שורה, אני יכול לבצע `require('os')` ומיד לקרוא למתודה, וכך להוסיף 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 שמדובר במקרה שלעיל, אבל היא אמורה להבהיר לכם שלא מדובר במקרה. באחד מהפרקם הבאים נראה מקרוב איך ה-`require` עובד כאשר נכתב מודול מסוינו.

מודולים של ECMAScript

שיטה נוספת לצריכת מודולים היא זו שנקראת ECMAScript Modules והוא תופסת תאוצה בשנים האחרונות באkosיסטם של ג'אווהסקרייפט וגם js.js הולכת ומטמיעה אותה, למורות שעדין בירית המחדל היא Common.js require-1. אבל רואים יותר ויותר שימוש ב-ECMAScript modules.

בשיטה זו, אנו צריכים מודולים באמצעות המילה השמורה import. אני מקlid import ואז את שם המשתנה שאני בוחר ואז את המילה השמורה from ואז את שם החבילה. אני אתן דוגמה לקוד של ECMAScript modules 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 לchlוטין לדוגמה האחורונה שנטנו עם require, רק שפה אנו משתמשים ב-import. זה כל ההבדל. במקום הפקודה זו, המיבאת את fs במאזעות require:

אנו משתמשים בפקודה זו, המיבאת את fs במאזעות import:

על מנת להשתמש ב-import, אנו צריכים לשמור את שם הקובץ בסימת .js ולא בסימת .mjs. כלומר הקוד שלעיל יעבד אך ורק אם הוא יהיה בקובץ שהסימת שלו היא .mjs כמו app.mjs.

גם פה לא מדובר במקרה אף אלא בדרך פשוטה להשתמש במודולים. ישנם מודולים שמייצאים כמה פונקציות או מחלקות ואז אנו צריכים לייבא את המשתנים שלhn באמצעות סוגרים מסולסים סביר שם הפונקציה. Homedir שיש בקובץ os. הדגמן בפרק זה אין משתמשים בפונקציית homedir שהיא אחת מהפונקציות שנחקרו במודול os של Node.js. כיצד אני משתמש בה עם import? באופן זה:

```
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 סוגרים מסולסלים וכתובן אשמור את הקובץ בשם עם סיוםת `.mjs`.

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

תרגיל:

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

פתרונות:

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

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

הדבר הראשון שאנו עושה הוא `require('fs')`. אני יוצר את הקובץ עם מетодת `writeFile` ואני מעביר לה שלושה ארגומנטים. ארגומנט ראשון הוא שם הקובץ שהוא שואתו 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`, לקרוא 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
הfonktsiyah nmzachat batuk kolbak

תרגיל:

חזרו על התרגיל הקודם אך ביצעו `import` ולא `require`.

פתרונות:

```
import fs from 'fs';

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

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

פרק 2

היכרות עם הדוקומנטציה של NODE.JS



היכרות עם הדוקומנטציה של Node.js

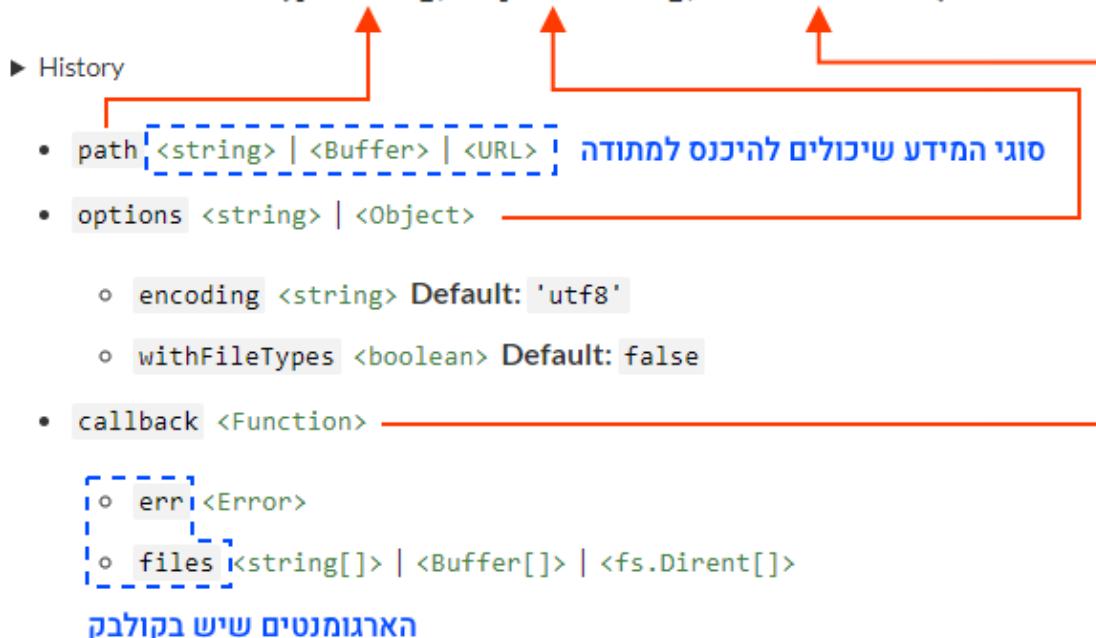
בפרק הקודם הסברתי על המודולים של ברירת המחדל `fs` וגם `os`. מהיכן הכרתי אותם? הידע זה לא בא לי בחלום אלא הוא כתוב בדוקומנטציה המפורת של `Node.js`. בדוקומנטציה זהו יש פירוט של כל המודולים שבאים כברירת מחדל עם `Node.js`.

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 עשויה והארוגמנטים שהיא מקבלת. אם יש קולבק – הפקת הנקראות לאחר השלמת הפעולה, יהיה מידע על שמה של הפקת.

בואו נסתכל על הדוקומנטציה של `readdir` על מנת לנסות להבין:

סוגרים מרובעים - אפשרי ולא חובה

`fs.readdir(path[, options], callback)`



ראשית אני רואה את שם המתודה, `fs.readdir`, ואני רואה שאין יכול להעיבר לה שלושה ארגומנטים. הראשון הוא `path`, השני הוא `options` שבו יש סוגרים מרובעים כדי לرمז על כך שהוא אפשרי ולא חובה. השלישי הוא הקולבק.

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

השני, `options` הוא לא חובה. אנו יודעים את זה בגלל הסוגרים המרובעים סביב הארגומנט הזה בכוורת. אני יכול להעיבר לו מחרוזת טקסט או אובייקט המכיל את כל האפשרויות.

השלישי, הוא הקולבק שלנו. זה הינו פונקציה (מקובל להעיבר פונקציית Hz) שמשופעת לאחר שהפעולה מסתיימת. ככלומר ברגע ש-`fs.readdir` מסיימת לקרוא את תוכן התיקייה, היא לוקחת את ההפונקציה שהעבכנו כארוגומנט שלישי ומפעילה אותה. כשהיא מפעילה אותה היא מאכלסת את שני הארגומנטים בקולבק. אני יכול לטפל בקולבק בארגומנטים אלו 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 `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 תראו מההו הדומה לזה:

ARGUMENTS AVAILABLE fs.readFile(path[, options], callback)

▶ History

- `path` `<string> | <Buffer> | <URL> | <integer>` filename or file descriptor
- `options` `<Object> | <string>`
 - `encoding` `<string> | <null>` Default: `null`
 - `flag` `<string>` See support of file system flags. Default: `'r'`.
- `callback` `<Function>`
 - `err` `<Error>`
 - `data` `<string> | <Buffer>`

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.

כתבת של המתודה הזו גם היא מזכירה מאוד את `readdir`. אני אציג קובץ בשם `app.js`, אכנס לתוכו את הקוד הבא:

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

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

ואפ漂流וento במציאות `app.js` `node`. מה שהסקריפט עושה הוא בעצם לקרוא את עצמו ולהציג את התוכן. אני אצפה לראות בטרמינל את כל הקובץ, בדוק כמה שעם `readdir` אני רואה את רשימת הקבצים.

אבל כשאנו מפעיל את התוכנה זו, אני רואה שהוא לא צפוי. במקום לראות את כל הטקסט, אני רואה שהוא 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 שבאupon 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, והميدע – המידע אמרור להיות תוכן הקובץ. אבל הלאה מוסבר שם לא מפורט שום `encoding`, אנו מקבלים Buffer וזה בדיקת מה שקיבלנו. איך אנו בעצם פותרים את הבעיה? גם זה מוסבר בדוקומנטציה (ואפיו 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 שהוא 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 מאד להכיר אותה ואפיו לבודק בה לפני שרצים לגוגל כדי לפטור 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 שרצו? כדאי לקרוא את הדוקומנטציה.

תרגיל:

אתרו בדוקומנטציה את המתודה `fs.mkdir` והשתמשו בה בסקריפט של Node.js על מנת ליצור תיקייה במקומם 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 עלייה נראה שהיא למתודות של `readFile` ו-`readdir`. גם היא מקבלת שלושה ארגומנטים:

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

► History **ארגומנט שלישי** **ארגומנט שני** **ארגומנט ראשון**
[לא חובה] **קולבק**

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

ารגומנט ראשון הוא המיקום שבו רוצים ליצור את התיקייה החדשה, הארגומנט השני הוא אפשרויות, והargonment השלישי הוא הקולבק שנקרא לאחר השלמת הפעולה. הקולבק זהה מקבל אר וرك אובייקט שגיאת אם הפעולה נכשלה. כדאי לשימוש בו תמיד מופעל לאחר הצלחת הפעולה ואם אין שם אובייקט שגיאת, אני יכול להניח שהפעולה הצליחה.

כך אני כותב את הסקריפט. בחרתי לנתחו אותו בקובץ `js.app` בתיקיית העבודה שלו.

```
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. הריצה שלו באמצעות `node app.js` תיצור את התיקייה זו.

תרגיל:

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

פתרונות:

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



הקוד שלי ייראה כך:

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

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

אם הוא שומר ב-`app.js`, הרצה שלו תיעשה על ידי `node app.js` ואם התיקייה קיימת, אני אראה בטרמינל הודעה של `Directory Deleted!`, והתיקייה שיצרתתי קודם תימחק.

פרק 3

גורסאות סינכרוניות למוחוזות אסינכרוניות



grossאות סינכרוניות למתודות אסינכרוניות

כתבתי בתחילת הספר ש-Node.js היא אסינכרונית וזו הכוח שלה. Node.js רצה על הליך אחד בmund ואם אנו נדרשים לפעולת מסוימת כמו קוריאת קובץ, התוכנה לא עומדת וממתינה לקובץ זה, אלא ממשיכת הלהה. 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 רואה בטרמינל הוא זה:

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

למה? כי בהתחלה אנו מדפיסים את ה-Before, קוראים ל-readdir. בזמן ש-Node.js רצה לתקן, אפשר להמשיך ואכן השורה הבאה, שהיא ההרצה של ה-After, רצה. רק כשה-readdir סימנה את העבודה, הקולבק מופעל ו מביא את התוצאות.

ואם יש לנו כמה קולבקים שככל אחד מהם מושלם בזמן אחר – כל קולבק יירוץ כשהוא יושלם. אם יש כמה קולבקים שיושלמו, הם ייכנסו לתוך. אבל העניין הוא ש-Node.js לא עוזרת ומחכה. אבל היא יכולה לעשות את זה באמצעות מתודות סינכרוניות. לעיתים אנחנו צריכים לעזור את הסкриיפט – בדרך כלל כשאנו בונים CLI (כלים לניהול שירותים או סביבות פיתוח) ואין טעם להמשיך את פעולה הסкриיפט אם פעולה מסוימת לא מושלמת.

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

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

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

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

זה אומר שהקוד ממש יעזור ויחכה להשלמת הפעולה. אם אני אשימוש `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 קיבל 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 שייפעל אם יש שגיאה. בקוד הבא למשל אני מנסה לקרוא תיקיה שלא קיימת באמצעות `readdirSync` – הפקצייה הסינכרונית תעיף לי שגיאה שאוותה אני יכול לתפוס עם try-catch ולטפל בה כרגע:

```

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 occurred!');
}
console.log('After readdir');

```

התוצאה של הריצת הקוד זהה תהיה:

```

Before readdir
Error has occurred!
After readdir

```

וכמובן הסקריפט לא יתפוצץ עם שנייה ו-stack trace, אם לא יהיה .try-catch, כמו שקיים(stack trace).
כמעט לכל מתודה שמתפלת בקבצים יש גרסה 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])
קריאה תוכן תיקייה	fs.readdir(path[, options], callback)	fs.readdirSync(path[, options])
מחיקת תיקייה	fs.rmdir(path, callback)	fs.rmdirSync(path)
קריאה קובץ	fs.readFile(path[, options], callback)	fs.readFileSync(path[, options])
יצירת קובץ	fs.writeFile(file, data[, options], callback)	fs.writeFileSync(file, data[, options])

מואוד לא מומלץ להשתמש בגרסאות סינכרוניות אלא אם כן אתם יודעים מה אתם צריכים – משתמשים בהן בדרך כלל לשימושים ייחודיים. מפתחה מואוד, במיוחד אם לא סגורים עד הסוף על האсинכרוניות, להשתמש בקוד זהה. אבל זה עלול להיות הרטני במקומות מסוימים כמו שרתים.

אם אתם לא יודעים אסינכרוניות וקולבקים היטב – זה הזמן לבצע חזרה על כן. קולבקים הם לא ייחודיים ל-Node.js ולא נלמדים בספר זה אלא נלמדים בספרים המלמדים ג'אווהסקריפט מאפס. בהמשך הפרק נלמד דרכים נוספות יותר לכתיבת קוד אסינכרוני, עם פורמייסים או עם Async-Await, שנותות בדיקת כמו קוד אסינכרוני. אבל כן או כן – 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, דוחחו על הייצירה שלה ואז מחקו אותה.

פתרון:

ראשית אנו צריכים לאתר את המתוודות של File System שנשתמש בהן. במקרה זהה, mkdirSync שמשמשת ליצור תיקייה ו-rmdirSync שמשמשת למחיקת תיקייה. אני יכול להתבסס על הידע המוקדם שלי או לבחון אותו בדוקומנטציה ולראות איך הן עובדות. במקרה זהה הן פשוטות ומקבלות ארגומנט אחד – שם התיקייה. כל מה שאנו צריך זה לקבל את שם התיקייה ולהוסיף אותו למיקום היחסוי. 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!`);
```

כדי לשים לב שאנו משתמש פה בתבנית טקסט (הגרש העקום – backtick) כדי להציג משתנה בתוך מחרוזת טקסט.

תרגיל:

בצעו את התרגיל הקודם בעזרת פונקציות אסינכרניות.

פתרון:

מציאת הגרסאות האсинכרניות אמורה להיות פשוטה – פשוט להסיר את ה-Sync משם הפונקציה ולהפץ בדוקומנטציה או להזכיר דוגמאות של הפרקים הקודמים. במקרה זה אנו משתמשים בקולבקים כי מדובר בפונקציות אסינכרניות.

כיוון שאנו חיבים לוודא שהתקייה קיימת לפני שنمחק אותה, נבצע את המחיקה בקורסוק של היצירה. למשל ברגע שהתקייה נוצרה, הקולבק של היזירה 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 שהתקייה נוצרה ואז אני יכול למחוק אותה.

מחיקת התיקייה
נעשית בתוך הקולבך
הראשון

```
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`);
  });
});
```

קולבך ראשון
מופעל לאחר יצירת
התיקייה

קולבך שני
מופעל לאחר מחיקת התיקייה

פרק 4

PACKAGE.JSON הנדסה ראשונית והפעלה של NPM



הכרה ראשונית ופעולת package.json של NPM

עד כה למדנו על מודולים בסיסיים שיש ב-`js.Node`, אלו שבאים כברירת המחדל. התמקדנו יותר ב-`File System` אבל ראיינו שיש עוד – כמו `os` למשל. מי שטרח לקרוא בדוקומנטציה של `js.Node` ראה שיש הרבה יותר אבל כולם מאוד בסיסיים, או יותר נכון `level` או, עושים דברים שהם פשוט וקלט אבל לא מעבר.

כל האציגו של `js.Node` הוא שמוסיפים דברים ללבת המערכת רק כשהם מוסיפים יכולת לפלאטפורמה. אם אני רוצה להשתמש ב-`js.Node` לדברים מתקדמים יותר, אני יכול להשתמש במודולים של ברירת המחדל כדי לבנות את הפקציונליות הזו, אבל זה ייקח לי הרבה זמן.

במקום זה, אני יכול להיעזר במודולים מורכבים שאנשים אחרים יצרו על בסיס המודולים הבסיסיים וכך לחסוך זמן רב. זה הרעיון שעומד בבסיס תרבות הקוד פתוח או המערכת האקולוגית, האkosystem, של `js.Node`. יש המון מודולים בקוד פתוח שככל אחד יכול להשתמש בהם למערכת שלו.

כך למשל, אם אני צריך לבנות מערכת מאפס, אני לא נדרש לפתח כל דבר ודבר אלא יכול להשתמש במודולים מתקדמים שאנשים או ארגונים אחרים בנו וסחררו בקוד פתוח. זה אפשרי לי ולכל אחד לבנות דברים באופן מאד מהיר וגם יעיל – במקרה כתוב בעצמי את הקוד, אני אשתמש במודולים שהושקעו בהם כבר אלף שנות אדם, הוכיחו את עצמן בהמוני פרויקטים אחרים והם טובים בהרבה, בדרך כלל, مما שمفתח בודד יכול ליצור. זה הכוח האמתי מאחורי `Node.js`.

כל המודולים בקוד פתוח נמצאים במאגר מודולים מיוחד בשם **NPM**, ראשי תיבות של `Node Package Manager`.

כתובת האתר היא npmjs.com ואם תיכנסו אליו תוכלו לראות שיש שם יותר ממאה אלף חבילות תוכנה. אבל מתוכן יש רק כמה אלפי חבילות תוכנה פופולריות. אפשר לחפש באתר ולבוחן את המודולים השונים.

NPM אינו רק מאגר של תוכנות אלא גם כלי שימושי לנו לנוהל את החבילות שאנו משתמשים בהן בפרויקטים שלנו.

הכלי זהה מותקן אוטומטית יחד עם Node.js. אם התקנתם אותו כמו שהסבירתי בפרק הראשון, תוכלו להשתמש בו כעת. היכנסו לטרמינל שלכם (למدى עליו באחד הפרקים הקודמים) וננווטו אל המיקום שבו נמצאת תיקיית הפרויקט שלכם. אפשר לעשות את זה מתוך Visual Studio Code כתבו `-v` וקח ותוכלו לראות את מספר הגרסה.

```
C:\Users\barzik\node_projects>npm -v
6.4.1
```

אפשר להשתמש ב-NPM על מנת להתקין את המודולים השונים. יותר מכך – NPM מאפשר לנו ליצור סוג של "הוראות התקינה" לתוכנה שלנו – כך שמי שימוש בה יוכל גם הוא להתקין את כל המודולים שבחרנו להשתמש בהם בקלות רבה באמצעות פקודה אחת. הוראות ההתקינה הללו נמצאות בקובץ שנקרא `package.json`, שהואונו נדרש ליצור בתיקייה הראשית של כל פרויקט Node.js משמש ב-NPM.

בקובץ הזה יש לא מעט דברים אבל בראש ובראשונה יש בו רשימות של המודולים השונים המשמשים בהם והגרסאות שלהם. כך שאם מתכוון אחר ירצה להשתמש בפרויקט שלנו, הוא לא צריך לנחש באילו מודולים של NPM השתמשנו אלא תהיה לו רשימה מסודרת שלהם (עם מספר הגרסה שלהם, כפי שנראה בהמשך) ודרך להתקין אותם.

יצירת `package.json` לפרויקט שלנו

inicnes לטרמינל שלנו וננווט אל התיקייה הראשית של הפרויקט שלנו. נקליך `init` וקח ואז נKİSH על אנטר. מיד תופיע לנו רשימה של שאלות. אנו יכולים ללחוץ אנטר כדי לתמוך בתשובות ברירת המחדל. לשאלות האלו יש משמעות שנדון בה מאוחר יותר.

```

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (node_projects)
version: (1.0.0)
description: My first project
entry point: (app.js)
test command:
git repository:
keywords:
author: Ran Bar-Zik
license: (ISC)
About to write to C:\Users\barzik\node_projects\package.json:

{
  "name": "node_projects",
  "version": "1.0.0",
  "description": "My first project",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC"
}

Is this OK? (yes)

```

לאחר שנאשר הכל, יוצר לנו בתיקייה הראשית קובץ בשם `package.json`. אפשר לפתוח אותו כדי להסתכל עליו. הוא פשוט למדי והוא בפורמט JSON.

כך הקובץ נראה他自己:

```
{
  "name": "node_projects",
  "version": "1.0.0",
  "description": "My first project",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC"
}
```

אפשר לראות שאין בו כמעט כלום. הפורמט שלו הוא פורמט שנקרא JSON או JavaScript Object Notation. מדובר בפורמט אחסן מייד שמצויר את הפורמט של אובייקט בג'אוوهסקרייפט, למעט כמה הבדלים פשוטים (למשל, שדות של אובייקטים חייבים להיות בין גרשימים). משתמשים בו בדרך כלל לקובצי קונפיגורציה. בספר הלימוד "למידה ג'אוوهסקרייפט בעברית" יש הסבר על פורמט הנתונים זהה ומתכנתית ג'אוوهסקרייפט אמוראים להכיר אותו. הוא בניו כמו אובייקט ג'אוوهסקרייפט אבל התכונות שלו (למשל ה-name, ה-version וה-description) מוקפות במירכאות נפולות. גם מחרוזות הטקסט. כמו בג'אוوهסקרייפט, גם ב-JSON תכונה יכולה להכיל אובייקט או מערך. פורמט הנתונים זהה קל לקרוא ומשתמשים בו בלי כמעט מקרים, הן לקונפיגורציה של מערכת והן לצרכים אחרים.

כרגע אנו לא רואים שום רשימה של מודולים, אבל זה די קל – אין מודולים חיצוניים שהותקנו, אז אין לנו רשימה. מה שיש לנו הוא קובץ הגדרות פשוט שבפושוטים – זה הכל. מה שצריך לזכור זה שקל ליצור את הקובץ הזה ובכל פרויקט Node.js הוא קיים.

התקנת המודול הראשון

אנו נבחר במודול הפופולרי ביותר ב-NPM, הלא הוא lodash. הוא נמצא בכתבوبة זו: <https://www.npmjs.com/package/lodash> אם תיכנסו אליו, תוכלו לראות הוראות התקנה, אבל הוראות התקנה של מודול הן פשוטות. מקלידיים npm install ואז את שם המודול. במקרה שלנו:

```
npm install lodash
```

אחד הקיצורים המקבילים של install הוא i ואפשר להקליד:

```
npm i lodash
```

אם תקלידו את השורה זו בטרמינל במקומם שבו יצרתם את ה-package.json, שזו היא התקيبة הראשית של הפרויקט, NPM תוריד את המודול זהה.

```
C:\Users\barzik\node_projects>npm i lodash
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN node_projects@1.0.0 No repository field.

+ lodash@4.17.11
added 1 package from 2 contributors and audited 1 package in 1.539s
found 0 vulnerabilities
```

אם תיכנסו לפרויקט, תוכלו לראות שנוצרה לכם תיקייה בשם `node_modules` ובתוכה יש תיקייה נוספת שנקראת `lodash`. בתיקיית `node_modules` אמורים להיות המודולים החיצוניים שאתם משתמשים בהם, והם בלבד. זו תיקייה שモורה לך. מאוד לא מקובל לבצע שינויים ידנית על התוכן שנמצא בתוך התיקייה זו. מי שמבצע שום שינוי זו אך ורק NPM, באמצעות סט פקודות שקיים בה.

לעתים המודולים החיצוניים שלכם משתמשים במודולים חיצוניים אחרים. NPM מטפלת בהורדה שלהם וגם הם יופיעו בתיקייה זו.

שינוי נוסף שהתרחש הוא ב-`package.json` – שם תוכלו לראות תוספת שנקראת `dependencies` ומכליה מידע על `lodash`:

```
"dependencies": {
  "lodash": "^4.17.11"
}
```

מה שיש ב-`dependencies` הוא רשימה של מודולים חיצוניים שבהם אנו משתמשים בפרויקט. במקרה זה – מודול `lodash` בלבד. ליד המודול מופיעעה הגרסה שלו עם כובע. נדון על כך בהמשך.

אם אתם מעלים את התוכנה שלכם לאנשו, מעבירים אותה לחבר או שומרים אותה בגיטהאב, לא מעלים יחד עם התוכנה את תיקיית `node_modules` אלא רק את `package.json`. כל מי שירצה להתקין את המודולים שאתם מעוניינים להשתמש בהם, יוכל לעשות זאת בקלות באמצעות ניוט עם הטרמינל אל מקום הפרויקט שלכם והקלדה של:

`npm i`

הפקודה זו מפעילה את kommand שninger ל-`package.json` ומוריד את כל המודולים שמפורטים ב-`dependencies` היישר מ-NPM. זו הסיבה שמאוד חשוב להකפיד שככל המודולים החיצוניים שאתה משתמש בהם מופיעים ב-`package.json`.

שימוש במודול חיצוני

אחרי שהתקנו את המודול החיצוני שלנו, הגיעו העת להשתמש בו. איך משתמשים בו? ממש כמו במודול פנימי. יש הסברים בדוקומנטציה של **lodash** אבל הינה פונקציה פשוטה שיש במודול זה – **reverse**. מה שהמتدה עושה הוא פשוט להפוך מערך. אם יש לנו מערך שהוא [1,2,3] היא תהפוך אותו ל [3,2,1] – די פשוט ונחמד.

איך נשתמש בו? נבצע לו `require` בדיק כmo מודול בריית מחדל. עם השם שמופיע ב-`:lodash`, במקורה זהה `package.json`

```
const _ = require('lodash');

let array = [1, 2, 3];

_.reverse(array);

console.log(array); // [3, 2, 1]
```

את ה-`require` העברתי למשתנה שהשם שלו הוא `_` – שם תקני לחולוטין של משתנה. כיוון ש-`lodash` הוא מודול פופולרי מאוד בתעשייה, זו הקונבנצייה לשימוש במודול הספציפי הזה. חשוב לציין שברוב הhabilitות לא מקובל להשתמש בסימנים מיוחדים בודדים. אם זה מאוד מפריע לכם אתם יכולים להשתמש בשם אחר. איך משתמשים במודול זהה? בדיק כmo במודול של `fs` שעליו למדנו קודם. ממש קל.

בואו נדגים עם מודול אחר. הפעם בחרתי במודול `chalk` שנמצא בכתובת צריים אותן. לפי כתובת המודול אני רואה שהוא שמו הוא `chalk`. אנווט לתקינה של הפרויקט שלי באמצעות הטרמינל (או באמצעות הטרמינל של `Visual studio code`) ואנכתוב:

```
npm i chalk@4
```

```
C:\Users\barzik\node_projects>npm i chalk
npm WARN node_projects@1.0.0 No repository field.

+ chalk@2.4.2
added 7 packages from 3 contributors in 111.827s
```

על מנת להקליד את גרסה 4 של המודול. אחרי שאלה אנטר ומתין שהפעולה תושלם, אני אראה שני דברים. ראשית, שבקובץ `package.json` מתווסף המודול `chalk` ומספר הגרסה שלו. באחד הפרקים הבאים אסביר על הגרסאות, אבל כרגע, קבלו את הנתון זהה כפי שהוא – מספר הגרסה עם הסימן `^`.

```
"dependencies": {
  "chalk": "^2.4.2",
  "lodash": "^4.17.11"
}
```

שנית, בתיקיית `node_modules` נוספה תיקייה של `chalk` שבה נמצא הקוד של המודול.
אני יכול כעת להשתמש ב-`chalk`.

אפתח את הקובץ `app.js` ואכנס אליו את הקוד הבא:

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

console.log(chalk.blue('Hello world!'));
```

הקוד הזה נלקח ישרירות מהדוקומנטציה של `chalk`. על מנת לראות מה הוא עושה אחזר לטרמינל
ואכטוב `app.js` וראה>Hello World! אבל בכחול.

```
C:\Users\barzik\node_projects>node app.js
Hello world!
```

אפשר כמובן לשלב כמה מודולים בלי בעיה, בדיק כמו מודולים של ברירת המחדל של Node.js:

```
const _ = require('lodash');
const chalk = require('chalk');

let array = [1, 2, 3];

_.reverse(array);

console.log(chalk.red(array)); // [3, 2, 1] BUT IN RED
```

מה שקרה פה הוא שgem lodash וgem chalk יעבדו במקביל, בלי בעיה.

שימוש לב: מקובל להציב את כל ה-require בתחילת הקובץ שעליו עובדים.

בסוף דבר מודולים הם לא קסם, הם לא וודו – גם הם קוד של Node.js שנכתב על ידי מתכנתיםبشر ודם שהשתמשו במודולים אחרים או במודולים שבאים עם Node.js. אנחנו גם יכולים לנתח, מאפס, מודולים שצובעים את הפלט של הקונסולה בצבעים מרהייבים או הופכים מערך. בסופה של דבר זה ג'אוوهסקרייפט. אבל למה לנו? למה לי לנתח שוב ושוב אותם מודולים שעושים אותם דברים כשהאני יכול להוריד אותם בקלות ולנהל אותם עם NPM?

ככה עובדים עם מודולים חיצוניים וזה הכלוח של NPM ושל האkosיסטם של Node.js – שפע של מודולים חיצוניים שמאפשרים המון דברים ויכולת מדיה של ניהול שלהם.

יש מנהל מודולים נוסף `ljs`, שנקרא `yarn`, והוא עובד בצורה זהה. אני לא מלמד אותו בספר זה. אפשר לקרוא עליו באתר שלו: yarnpkg.com.

יצירת פרויקט עם התומך ב-ECMAScript modules

אם אתם רוצים לבנות את הפרויקט שלכם עם מודולים התומכים ב-ECMAScript modules כלומר ב-`import`, כאשר מרכיבים את `init` ו-`npm` לראשונה על מנת ליצור את הפרויקט, יש להקליד גם `es6` באופן זהה: `npm init es6`.

הקלדת הסкриיפט באופן זהה גורמת לשורה זו להופיע בקובץ `package.json`:

```
"type": "module",
```

ואז אין צורך לשמור את הקבצים המשתמשים ב-`import` עם הסיומת `.js` ונitin להשתמש ב-`import` ללא בעיה (אך לא ב-`require`). אני נדרש כמובן לוודא שהמודולים החיצוניים תומכים ב-ECMAScript module (נקרא גם ESM). כך למשל, `chalk` מגרסת 5 תומך אך ורק ב-ESM, ככלומר אפשר להשתמש בו ב-`import` בלבד ולא ב-`require`.

תרגיל:

השתמשו במודול החיצוני `request` בכתובות `request` על <https://www.npmjs.com/package/request>. מנת לבצע קרייה לעמוד הראשי של מנוע החיפוש גוגל ולקבל את ה-HTML שלו בקונסולה.

פתרונות:

1. באמצעות הטרמינל שלנו ניכנס אל התקינה שלuproject שלנו. נודא שבתיקייה יש `package.json`.
2. ניכנס אל האתר של NPM ונבחן את המודול `request` – נמצאות שם הוראות ההתקנה והשימוש באתר NPM עצמו.
3. נתקין את המודול באמצעות הקלדה של `npm i request` ונטר. נמתין עד סוף ההתקנה.
4. נודא שב-`package.json` יש פירוט של `request`.
5. נעתיק את הדוגמה מהדוקומנטציה של המודול אל תיק `app.js`:

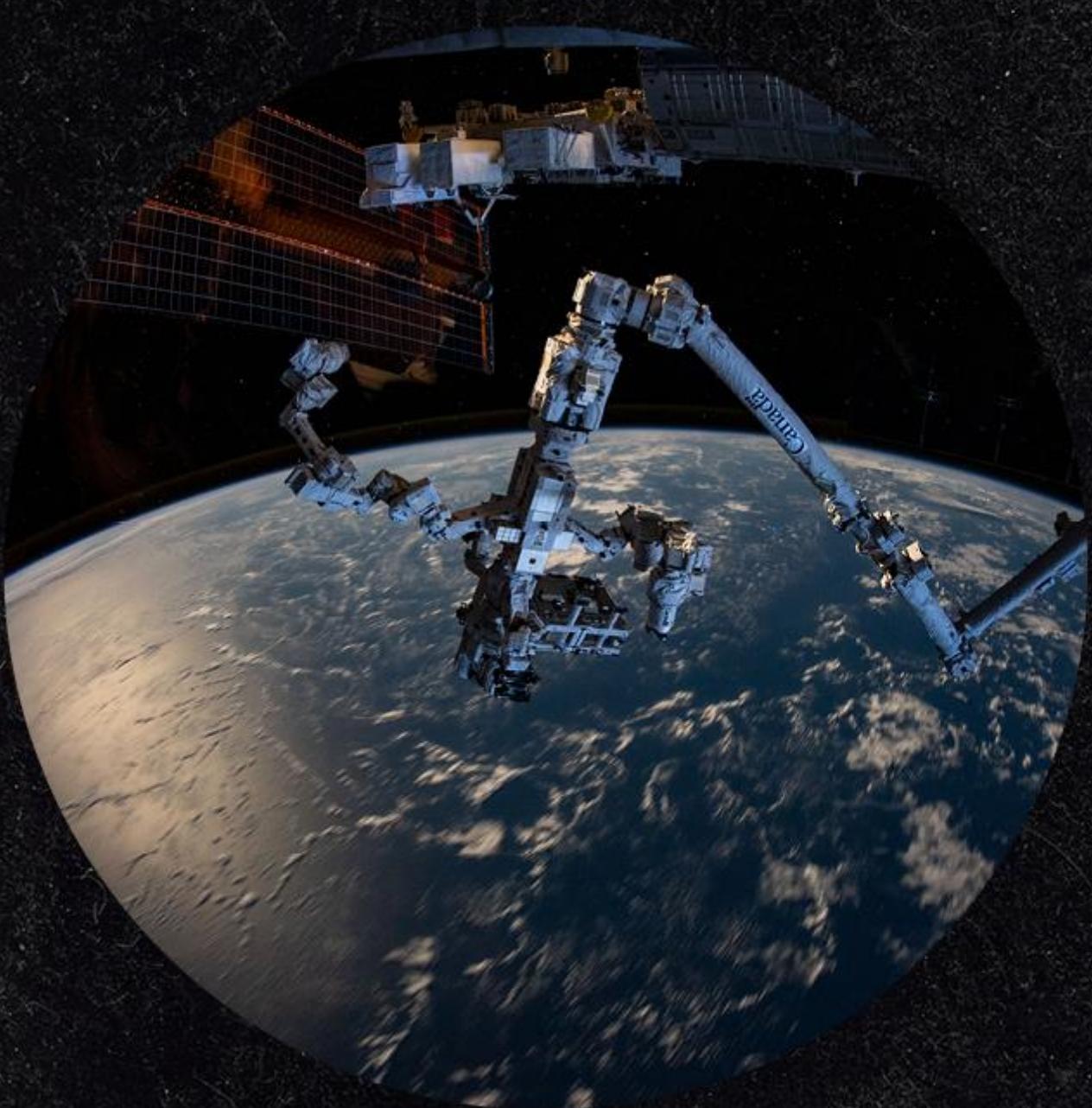
```
const request = require('request');

request('http://www.google.com', function (error, response, body) {
  console.log('error:', error); // Print the error if one occurred
  console.log('statusCode:', response && response.statusCode); // Print the response status code if a response was received
  console.log('body:', body); // Print the HTML for the Google homepage.
});
```

6. נפעיל את `app.js` באמצעות הקלדה של `node app.js` בטרמינל (כאשר אנו נמצאים במיקום שלuproject שלנו כמובן) ונראה את התוכנה עובדת.

פרק 5

עבודה אסינכרונית ומעבר מהולבקים לפומxisים ול-AWAIT



עבודה אסינכרונית ומעבר מ콜בקים לפרומייסים ול-**async/await**

עד כה רأינו שענו לעבודים עם קולבקים פשוטים ב-`js.Node`. אבל כל מתכנת ג'אווהסקריפט יודע שקולבקים זו דרך מסורבלת לעובדה עם קוד אסינכרוני. ב-`js.Node` יש דרכיהם טובות יותר מאשר קולבקים ואני נלמד בפרק זהו כיצד להסביר את הקוד שלנו והמודולים שלנו לעבודה עם פרומייסים ועם `async/await`.

בספר זה אני לא מלמד על קולבקים, פרומייסים או `async/await`. זה ידע שנלמד במסגרת הספר "לימוד ג'אווהסקריפט בעברית" ואני לא חוזר עליו כאן. אם המונחים האלה לא מוכרים לכם, אני ממליץ בחום לקרוא עליהם, בין אם בספר ובין אם במקומות אחרים כמו developer.mozilla.org.
המודולים של `js.Node` עובדים כרגע באופן אסינכרוני עם קולבקים בלבד. כ-6 ECMAScript (שידוע גם בתור תקן ES2015) יצא עם היכולת האסינכרונית של פרומייסים, רבים רצו לעבוד איתם ועם המודולים שבאים עם `js.Node` כבירית מחדל, אבל לא היה אפשר. זו הסיבה שמתכנתים רבים עבדו עם מודולים חיצוניים כמו `bluebird`, שהמירו את המודולים הסטנדרטיים של `js.Node` לעבודה באמצעות פרומייסים. מ-`js.Node` גרסה 8 יש לנו דרך לעשות את זה עם מודול של `js.Node` וזה הדרך הפешטה והטובה לעבוד עם המודולים של `js.Node` ופרומייסים כרגע.

המודול שבו משתמשים ב-`js.Node` על מנת להסביר את המודולים שלו למודולים שתומכים בפרומייסים הוא `util`. מזכיר במודול שכשמו כן הוא – מכיל המון כליעזר. אחד מכלי העזר הוא מתודה בשם `promisify` שמקבלת כารוגמנט מתודות אחרות של `js.Node`. המתודה זו מסייעת לנו ללקח מודול שמשתמש בקולבקים ולהמיר אותו למודול שמשתמש בפרומייסים.

המודול הזה נקרא `carigil` באמצעות `require`. עם מתודת `promisify` עוטפים את המתודה האסינכרונית הרגילה, מחזירים אותה למשתנה וקוראים למשתנה זהה.

הינה דוגמה עם `readdir`. אפשר לראות שההディ פשוט:

```
const util = require('util');
const fs = require('fs');
const readdir = util.promisify(fs.readdir);

readdir('.').then((results) => {
  console.log(results);
});
```

אני קורא לモduל `util` ולモduל `fs` עם `require` כרגיל. בשורה השלישי קורה הנס – אני בעצם עוטף את המתודה שאני רוצה להשתמש בה, כמו `fs.readdir`, במתודה `util.promisify` ומוחזיר את התוצאה לקבוע בשם `readdir` (אני יכול לקבוע איזה שם משתנה שאני רוצה, כמובן). מעכשיו הקבוע הזה הוא בעצם ה-`readdir` של `fs` והוא יוכל להשתמש בו לכל דבר ועניין, בהבדל פשוט אחד – אני לא מעביר לו קולבק אלא מקבל ממנו פורםיס ועובד אליו כמו כל שירות שמחזיר לי פורםיסים.

זה יעבוד רק על מתודות של מודולים שמקבלים קולבק בהתאם הארגומנט האחרון ובkolbak יש `error` ו-`value` של חזרה, כמו ב-`readdir`. להזכירכם, `readdir` נראה בדוק כהה:

```
fs.readdir(path, [options], (error, results) => {});
```

ולפיכך הוא ממש מתאים ל-`utils.promisify`. הסטנדרט של קולבק אחרון נחשב כסטנדרט מקובל מאוד בתעשייה ולפיכך `utils.promisify` יתאים לרוב המודולים שיש ב-NPM.

ובמקרה שיש פורםיסים, יש גם `async/await`.

```
const util = require('util');
const fs = require('fs');
const readdir = util.promisify(fs.readdir);

async function init() {
  const result = await readdir('./');
  console.log(result);
}

init();
```

ככה `js.Node` סטנדרטי ומודרני נראה וכך בדיק אני כתוב אותו מהנקודה הזו והלאה. אם הקוד הזה לא מוכר לכם, זה הזמן לעבור על `async-await` – דרך מומלצת ומודרנית לכתוב קוד אסינכרוני בג'אווהסקרייפט.

מה קורה אם הקולבק מחזיר כמה ערכים ולא רק ערך אחד? אdegim בעזרת המודול DNS שהוא מודול ביריתech מחדל ב-`js.Node` ויש לו מתודה שנקראת `lookup`. המתודה זו מבצעת פעולה שנקראת `dns lookup` – פעולה שמחזירה את כתובות ה-IP מהחורי שם מתוך מסויים ואת הסטטוס שלו. הקולבק שהמתודה `dns.lookup` מפעילהמחזיר שני ארגומנטים: הראשון הוא `?promisify` והשני הוא `family`. מה קורה כשאני עושה לה `address`

```
const util = require('util');
const dns = require('dns');
const lookup = util.promisify(dns.lookup);

async function init() {
  const result = await lookup('nodejs.org');
  console.log(result); // { address: '104.20.22.46', family: 4 }
}

init();
```

לא קורה הרבה – פשוט במקום ערך, חוזר אובייקט שבו יש מפתחות עם כל הערכים. זה הכל.

באו נראה איך הכל מתחבר יחד. אני אקח את מודול `fs`, את מודול `dns`, ואוצר תוכנה אסינכרונית שעובדת בלי קולבקים. התוכנה שלי תעשה משהו פשוט – היא תיקח כתובות ה-IP של `js.Node`, תיצור תיקייה המכילה כתובות ה-IP הללו ותדפיס את כל תוכן תיקיית האב.

זו פעולה שהייתי צריך בשביבה שלושה קולבקים מקוונים. עם `async/await` זה הרבה יותר קל ונעiem:

```
const util = require('util');
const dns = require('dns');
const fs = require('fs');
const lookup = util.promisify(dns.lookup);
const mkdir = util.promisify(fs.mkdir);
const readdir = util.promisify(fs.readdir);

async function init() {
  const result = await lookup('nodejs.org'); // { address:
'104.20.22.46', family: 4 }
  const address = result.address;
  await mkdir(address);
  const directories = await readdir('.')
  console.log(directories);
}

init();
```

אני לוקח את שלוש המethodות שאני רוצה: `fs.readdir`-`dns.lookup`, `fs.mkdir` ועוטף אותן ב-`util.promisify` ואז אני יכול לעבוד איתן עם פרומיסים או עם `async-await`. במקרה הזה אני בוחר לעבוד איתן באמצעות `async-await`. הקוד הרבה יותר קרייא ונוח.

שיםו לב שהוא נכתב כמו קוד סינכרוני אבל מתנהג כמו קוד אסינכראוני אמיתי. במקרה הזה כל מה שיש בפונקציית `itou` תלוי זה בזה, אבל אם היו פונקציות אחרות, הן היו רצויות גם כן בזמן שהוא שיש ב-`itou` היה מחייב לרשום ממערכת הקבצים או מהרשף.

מודולים ב-`js`-Node שתומכים בפרומייסים באופן טבעי

ב-`js`-Node החלו להכניס תמיינה טבעית בפרומייסים במודולים שלהם גם ללא `promisify` או תוספות מלאכותיות. איך? באמצעות תכונת `promises` שנמצאת במודולים התומכים בפרומייסים. כאשרנו עושים `require` למודול התומך באופן טבעי בפרומייסים, נעשה אותו כך:

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

ואז אני אוכל להשתמש בו ממש וכךילו עשויתי לו `util.promisify`. לשם הדוגמה, הקוד הבא יאפשר הרבה יותר פשוט:

```
const dns = require('dns').promises;
const fs = require('fs').promises;

async function init() {
  const result = await dns.lookup('nodejs.org'); // { address:
'104.20.22.46', family: 4 }
  const address = result.address;
  await fs.mkdir(address);
  const directories = await fs.readdir('.')
  console.log(directories);
}

init();
```

בשעת כתיבת שורות אלו עדין מזכיר בפייצ'ר ניסיוני.

תרגיל:

מודול DNS מכיל מתודה בשם reverse שנמצאת במקום זהה בדוקומנטציה: https://nodejs.org/dist/latest-v8.x/docs/api/dns.html#dns_dns_reverse_ip_callback המתודה זו מקבלת כתובת IP ומחזירה את שם המתחם הקשור אליה. יש ליצור פונקציה אסינכרונית שמקבלת IP כפרמטר ומחזירה את כל שם המתחם. כדי לבדוק, נסו את 8.8.8.8.

פתרון:

```
const util = require('util');
const dns = require('dns');
const reverse = util.promisify(dns.reverse);

async function reverseIPLookup(ip) {
  const domains = await reverse(ip);
  console.log(domains);
}

reverseIPLookup('8.8.8.8');
```

ראשתי משכתי את המודול `utils` ואת המודול `dns`. עכשו אני יכול לעתוף את `dns.reverse` ב-`Promise` ואני יכול להשתמש בו ב-`await`.>Create a promise-based asynchronous function named `reverseIPLookup` that takes an IP address as an argument and returns all domain names associated with it. You can use the code above as a starting point.

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

dns.reverse('8.8.8.8', (err, domains) => {
  console.log(domains);
});
```

פרק 9

איזוניים



אירועים

אחרי שלמדנו איך `js.Node` עובדת מבחינה בסיסית, הגיע הזמן לצלול יותר עמוק אל האירועים. אירועים הם מנגנון חשוב ביותר ב-`js.Node` שמאפשר למודולים שונים ולבויקטים שונים לתקשר זה עם זה בקלות. בעזרת אירועים אנו מפעילים קטעים שונים בקוד שלנו. כל אירועים ב-`js.Node` מבוססים על המודול `events` וחשוב מאוד להכיר אותו כיון שהוא עומד בסיס של לא מעט מודולים ומאפשר לנו גם לבנות מודולים שמתקשרים עם מודולים אחרים.

יש מוצרים שימושיים בפרומייסים כדי לתקשר ויש ככלו שימושים באירועים. כל מערכת והצרcisים שלה ושל מי שתכנן אותה. מערכות מבזירות יותר, שיש להן תוספים או מודולים חיצוניים, נוטות להשתמש באירועים. מערכות אחרות, במיוחד לקריאה ולכתיבה, נוטות להשתמש בפרומייסים.

המודול `events` עובד באופן שונה מהמודולים שאנו מכירים. הוא קלאס שמכיל מתודות. אנו יכולים להשתמש במתודות האלו ישירות או לרשת ממנו – קלומר לבצע לו `extend`. אנו נלמד על הדרך השנייה – ירושה. אם אתם מכירים את ג'אויהסקריפט לעומק, אתם בוודאי יודעים שהירושה זו היא לא י魯שה "אמתית" אלא פשוט סוכר סינטקטי מעל אובייקט של ג'אויהסקריפט. כך או כן, בנגדול מודול רגיל שבו אנו עושים `require` ומשתמשים במתודות שלו באופן סטטי, כאן אנו אמורים ליצור את הקלאס שלנו ואז לרשת את ה-`event`. ברגע שהוא עושים את זה, הקלאס שלנו, או המודול שלנו אם תרצו – מקבל את כל המתודות של מודול ה-`events` ויכול לעבוד עם אירועים. בואו ניצור קלאס משלנו, נתן לו לרשת מ-`events` ואז נראה איך מצדדים לו אירועים:

```
const EventEmitter = require('events');

class MyClass extends EventEmitter { };

const myClass = new MyClass();

myClass.on('someEvent', () => {
  console.log('someEvent occurred!');
});

myClass.emit('someEvent');
```

הדבר הראשון שאני עושה הוא לבצע require ל-`events`. מקובל לעשות require לתוך `EventEmitter`, למרות שאפשר לעשות לכל שם משתנה שנבחר. בኒגוד למודולים אחרים, אני לא משתמש ישירות ב-`events` אלא יורש ממנו בקלאס שלי, במקרה זהה `MyClass`. זהו קלאס פשוט מאוד שאין בו כלום. אבל ברגע שאני יורש מ-`events`, אני מקבל את כל המתודות שיש בו. המתודות מופיעות בדוקומנטציה:

https://nodejs.org/api/events.html#events_class_eventemitter

איך מתקדמים מכאן? עכשו אני יכול לבצע אימפלמנטציה להازנה לאירוע פשוט. איך? אני משתמש במתודת `on`, שכאמר נמצאת בקלאס שלי אחורי שירשתי מ-`events`, ו"נרשם" לאירוע. אני בעצם אומר – שמע נא, `js` חמוד, ברגע שיש אירוע בשם `someEvent` הוא `someEvent`, תפעיל את הקולבק הזה. מה יש בקולבק? בקוד שלעיל סתם `console.log`. אבל זה יכול להיות הרבה יותר.

בהמשך הקוד אני פשוט מפעיל את האירוע באמצעות הפונקציה `emit`. הפעלת אירוע גורמת ל科尔בק להתקיים:

```
const EventEmitter = require('events');
class MyClass extends EventEmitter { };

const myClass = new MyClass();
myClass.on('someEvent', () => {
  console.log('someEvent occurred!');
});
myClass.emit('someEvent');
```

מתודת `on` מקבל שני ארגומנטים: הראשון הוא שם האירוע שANI נרשם אליו והשני הוא הקולבק המופעל.

האירוע יכול להיות מופעל מכל מקום, לא רק מבחוץ! בדוגמה זו אני מכניס את `this` בתוך מתודה של הקלאס ומפעיל אותה באמצעות המילה השמורה `this` – שהוא רפרנס לאובייקט עצמו. אני מבצע הפעלה של אירוע `this.emit`. בתוך הקלאס הוא שווה ערך ל-`MyClass`. פשוט אחד מבפנים והשני מבחוץ:

```
const EventEmitter = require('events');

class MyClass extends EventEmitter {
  callMe() {
    this.emit('someEvent');
  }
}

const myClass = new MyClass();

myClass.on('someEvent', () => {
  console.log('someEvent occurred!');
});

myClass.callMe();
```

כאן למשל אני מפעיל את האירוע מתוך ה-constructor עם setTimeout עם constructor. גם פה אני משתמש ב-`this` כי אני מפעיל את emit מתוך הקלאס ולא מבחוץ. אבל הדוגמה צריכה להיות ברורה:

```
const EventEmitter = require('events');
class MyClass extends EventEmitter {
  constructor() {
    super();
    setTimeout(() => {
      this.emit('someEvent');
    }, 1000);
  }
}

const myClass = new MyClass();

myClass.on('someEvent', () => {
  console.log('someEvent occurred!');
});
```

אפשר להאזין לאירועים עם `on` מכל מקום ולהפעיל אותם מכל מקום שיש בו רפרנס, מבפנים או מבחוץ. אירועים(events) דומים באופן כמעט זהה לאירועים בג'אוויסקייפ בדף, אבל השינוי העיקריפה – כיוון שאין דף – הוא שהוא יוצרים את האירועים בעצמם. קלומר לנו יוצרים את המאזינים לאירועים שעובדים כשם קוראים וגם, במידת הצורך, את האירועים עצם.

לסיכום אני אציג דוגמה נוספת, שבה גם הפונקציה המטפלת באירוע ושהאותה אני מצמיד למאזין באמצעות `on` וגם הפעלת האירוע נעשות מתוך הקלאס ולא מבוחץ:

```
const EventEmitter = require('events');

class MyClass extends EventEmitter {
  constructor() {
    ();
    this.on('someEvent', this.someEventHandler);
    this.emit('someEvent');
  }

  someEventHandler() {
    console.log('someEvent occurred!');
  }
};

const myClass = new MyClass();
```

וכך בדרך כלל תראו מאזינים בתוך מודול. מובן שככל אחד מחוץ למודול יוכל להתחבר לאירוע. פונקציות שמטפלות באירוע נקבעות בדרך כלל `handler` ומואוד מקובל להצמיד את המילה `handler` לשם פונקציה שמטפלת באירוע.

כיבוי מאzin

כמו שהצמידנו קולבק לאירוע, ככלומר גרמנו לו להיות מאzin באמצעות `on`, אנחנו יכולים לנכבות אותו באמצעות `off`. זה חשוב אם אנחנו לא צריכים את המאזין הזה יותר. עושים את זה באמצעות `off` ביותר:

```
const EventEmitter = require('events');

class MyClass extends EventEmitter {
  constructor() {
    super();
    this.on('someEvent', this.someEventHandler);
    this.off('someEvent', this.someEventHandler);
    this.emit('someEvent');
```

```

}

someEventHandler() {
  console.log('someEvent occurred!');
}

};

const myClass = new MyClass();

```

בקוד זהה אני יוצר מאזין. בדיקן כמו בדוגמה הקודמת – הכל נמצא בתוך הקלאס. מייד אחרי היצירה של המאזין עם חס, אני מכבה את המאזין הספציפי זהה באמצעות שימוש בפקודה off והעברת שני ארגומנטים: שם האירוע ופרנס לפונקציה שטפלת בו. בדיקן כמו ב-חס. אם תרצו את הקוד הזה, תראו שדבר לא קורה למרות שעשיתי emit.

הפעלת יותר מאירוע אחד

אנו יכולים להפעיל את אותו אירוע שוב ושוב (ושוב), כמה פעמים שנרצה והפונקציה המאזינה תפעיל כמה פעמים. כאן למשל אני מבצע emit שלוש פעמים ובכל פעם someEventHandler :

```

const EventEmitter = require('events');

class MyClass extends EventEmitter {

  constructor() {
    super();
    this.on('someEvent', this.someEventHandler);
    this.emit('someEvent');
    this.emit('someEvent');
    this.emit('someEvent');
  }

  someEventHandler() {
    console.log('someEvent occurred!');
  }
};

const myClass = new MyClass();

```

אבל אני יכול גם לבצע האזנה פעם אחת בלבד ואז להרוג את המאזין באמצעות הפקודה `.once`. אם אני מצמיד את ההאזנה באמצעות `once` במקום `on`, הפונקציה המאזינה תפעל רק פעם אחת:

```
const EventEmitter = require('events');

class MyClass extends EventEmitter {

  constructor() {
    super();
    this.once('someEvent', this.someEventHandler);
    this.emit('someEvent');
    this.emit('someEvent');
    this.emit('someEvent');
  }

  someEventHandler() {
    console.log('someEvent occurred!');
  }
};

const myClass = new MyClass();
```

אם אני ארים את הפונקציה הזו, אני אראה הדפסה אחת בלבד בקונסולה – כי השתמשתי ב-`.once`

הצמדת כמה פונקציות מאזיניות לאירוע אחד

אין שום בעיה להצמיד כמה וכמה פונקציות ומאזינים שונים רוצחים לאירוע אחד. זה כל העניין באירועים. מתרחש emit אחד – אבל לאירוע הזה שמתקיים יכולות להיות אלף מאזינים שיביצעו אלפי פעולות.

```
const EventEmitter = require('events');

class MyClass extends EventEmitter {

  constructor() {
    super();
    this.on('someEvent', this.someEventHandler);
    this.on('someEvent', this.someEventOtherHandler);
    this.emit('someEvent');
  }

  someEventHandler() {
    console.log('someEvent occurred! This is #1 listener');
  }
  someEventOtherHandler() {
    console.log('someEvent occurred! This is #2 listener');
  }
}

const myClass = new MyClass();
```

איך קובעים איזה מאzin מתבצע ראשון? לפי סדר ההצמדה. במקרה הזה הצמדתי את someEventOtherHandler לפני someEventHandler והוא יירוץ ראשון. אם אנו רוצחיםprependListener someEventOtherHandler ירוץ קודם, אנו יכולים להשתמש במתודה prependListener (למרות שנדאי שהקוד לא שלוקחת את הפונקציה המאזינה שמתقبلת וגורמת לה לירוץ קודם (למרות שנדאי שהקוד לא يستמך על זה). אם נסתכל על הקוד הזה ונريץ אותו:

```
const EventEmitter = require('events');

class MyClass extends EventEmitter {

  constructor() {
    super();
  }
```

```

    this.on('someEvent', this.someEventHandler);
    this.prependListener('someEvent', this.someEventOtherHandler);
    this.emit('someEvent');
}

someEventHandler() {
    console.log('someEvent occurred! This is #1 listener');
}
someEventOtherHandler() {
    console.log('someEvent occurred! This is #2 listener');
}

};

const MyClass = new MyClass();

```

נראה את הפלט הבא:

```

someEvent occurred! This is #2 listener
someEvent occurred! This is #1 listener

```

בגלל שהשתמשנו ב-`prependListener`, אז למרות שהצמדנו את `prependOnceListener` לאחרונה, היא תרוץ ראשונה. יש לנו גם את `prependOnceListener` שהוא מקבלת של `once`. ככלומר היא מעבירה את הפונקציה המאזינה לראש התור, אבל היא תרוץ פעם אחת.

וכמובן יש לנו את **`removeAllListeners`** שבאמצעותה אנו יכולים להסיר את כל המאזינים לאירוע מסוים:

```

const EventEmitter = require('events');

class MyClass extends EventEmitter {

    constructor() {
        super();
        this.on('someEvent', this.someEventHandler);
        this.on('someEvent', this.someEventOtherHandler);
        this.removeAllListeners('someEvent');
        this.emit('someEvent');
    }
}

```

```

someEventHandler() {
  console.log('someEvent occurred! This is #1 listener');
}
someEventOtherHandler() {
  console.log('someEvent occurred! This is #2 listener');
}
};

const MyClass = new MyClass();

```

בדוגמה שלעיל שום פקודה לא תודפס לקונסולה למרות שהצמדנו כמה מאזינים לאיורע .removeAllListeners, בגלל שהסרנו אותו באמצעות someEvent

העברת נתונים באירועים

אנו יכולים לקבל נתונים בפונקציה המאזינה ולהעביר נתונים בהפעלת האירוע. זה די פשוט. כל מה שאנו צריכים לעשות הוא להעביר ארגומנט -`emit` ולקבל ארגומנט בקורסוק של הפונקציה המאזינה, והารגומנט יוכל להיות כל טיפוס מידע, כמו כן, כולל אובייקטים ומערכות:

```

const EventEmitter = require('events');

class MyClass extends EventEmitter {

  constructor() {
    super();
    this.on('someEvent', this.someEventHandler);
    this.emit('someEvent', 'Helllllooooo');
  }

  someEventHandler(arg) {
    console.log(`someEvent occurred! with ${arg}`);
  }
};

const MyClass = new MyClass();

```

מה שיודפס כאן כתוצאה מההרצה הוא:
 someEvent occurred! with Helllllloooo

ה-oooooo Helllllloooo כМОבן הנג'ע מהairoו עצמו.

תרגיל:

צרו קלאס שירש מ-`events` ונקרא כלב. בכל פעם שמופעל אירוע של `food`, הקלאס כותב "נבייה" אל הקונסולה. הפעילו את האירוע מתוך הקלאס ומוחצתה לו.

פתרון:

```
const EventEmitter = require('events');

class Dog extends EventEmitter {

  constructor() {
    super();
    this.on('food', this.foodHandler);
    this.emit('food');
  }

  foodHandler() {
    console.log(`bark!`);
  }
};

const dog = new Dog();

dog.emit('food');
```

הדבר הראשון שעשיתי הוא `require('events')`. מהרגע זהה אני יכול לבצע `extends` מהקלאס שלי, שקרأت לו `Dog`, מ-`EventEmitter`. מהנקודה הזאת לקלאס שלי יש את כל המתודות של `EventEmitter` ואני יכול לעבוד.

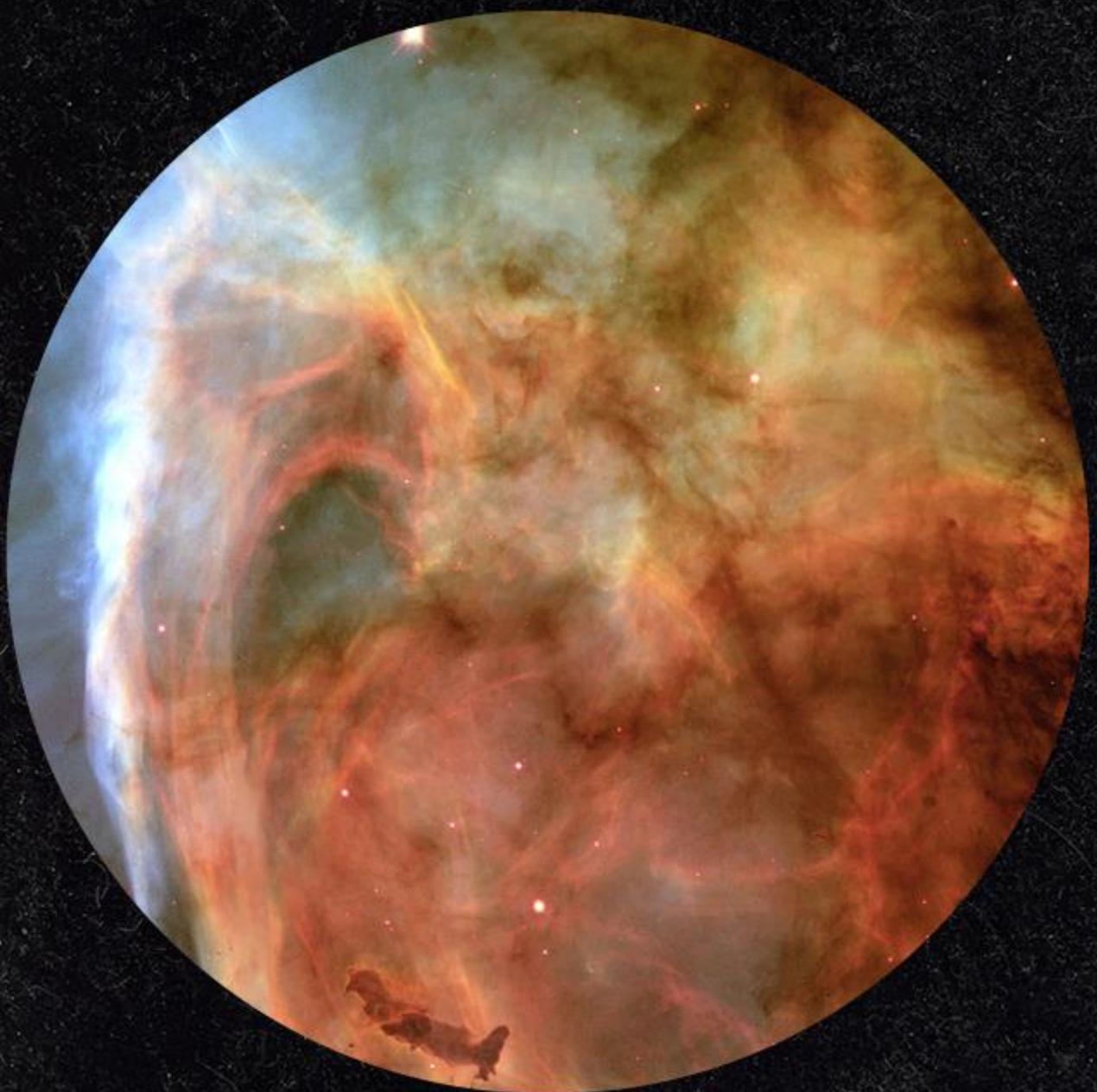
יצרתי `constructor` והכנסתי בו `super` כיון שהוא יורש (זו קונבנצייה של ג'אווהסקריפט). יצרתי גם פונקציה מסוימת בשם `foodHandler` שנובחת אל תוך הקונסולה.

השלב הבא הוא להציג את הפונקציה המאזינה, `foodHandler`, אל אירוע `food` באמצעות `on`. מאייפה היא הגיעה? זוכרים שאני יורש מ-`EventEmitter`? משם.

עכשו כל מה שנותר לי זה "להפעיל" את האירוע, ואת זה אני עושה באמצעות `emit`. פעם אחת מתוך `constructor` ופעם שנייה מבחוץ. בפלט אני אראה שתי "نبיחות".

פרק 7

יצירת שורת-HTTR א-בסיסי



יצירת שרת HTTP בסיסי

אחרי שבפרק הקודם למדנו על אירועים, אנו יכולים להשתמש במודולים שימושיים את events וירושים ממנה. המודול המפורסם ביותר הוא מודול `http`. המודול זהה מאפשר לנו בעצם לבצע פעולות מול הרשת, הן כיוון התקשרות (מה שנקרא קליננט או לקוח) והן כמקבל התקשרות (מה שנקרא שרת).

משמעות ראשי התיבות HTTP הוא Hyper Text Transfer Protocol והוא מדבר בעצם בפרוטוקול התקשרות הבסיסי של הרשת. כאשרנו מפעילים דפדפן ונותנים לו פקודה להיכנס לאתר וצופים בתוצאה – הפקודה והתוצאה (זהה דף האינטרנט) מועברים על גבי הפרוטוקול הזה. מודול `http` יכול לשגר בקשות HTTP כמו דפדפן ולהציג תשובה ממש כמו אתר או שרת. שרת הוא בעצם מחשב המחבר לרשת ובניגוד למחשב שלנו, הוא מטפל בבקשתות שנשלחות אליו. אנו נתמקד בפרק הזה בשרת וניצור שרת צזה, שיזע לבקשת דפדפן ולהציג תשובה. כאמור, מודול השרת הוא `http`.

איך אני יודע שמודול `http` משמש באירועים? פשוט מאד. זה כתוב בדוקומנטציה! אם תפתחו את הדוקומנטציה של `http` ותגלו אל `http.Server` תונלו לראות באופן מאד מפורש שהקלאס הזה יורש מ-

Class: `http.Server`

Added in: v0.1.17

This class inherits from `net.Server` and has the following additional events:

אם נלחץ על הקישור נראה שבאמצע הקלאס המרכזי שמן `https.server` יורש הוא `.EventEmitter`.

`net.Server` is an `EventEmitter` with the following events:

גם בדוקומנטציה של המתודות השונות של `http.Server` נראה שיש לנו אירועים, ואנו כבר אמרו
לדעת לעבוד היטב עם אירועים ב-`Node.js`.

- `Class: http.Server`

- `Event: 'checkContinue'`
- `Event: 'checkExpectation'`
- `Event: 'clientError'`
- `Event: 'close'`
- `Event: 'connect'`
- `Event: 'connection'`
- `Event: 'request'`

ازבעצם `http` הוא מודול מורכב יחסית. הוא יורש ממודול אחר שנקרא `net`, ומודול `net` מכיל
בתוכו את היכולת ליצור אירועים, יכולת שנייתה לו על ידי `events`. אבל זה לא מאד מורכב. אוטנו
פחות מעניין, כאשר אנו כותבים, ממי כל אחד יורש. אנחנו צריכים את הידע הזה כאשר אנו
מסתכלים בדוקומנטציה. מה שקרה הוא שלכל מודול יש המתודות והתכונות שלו.



נאמו, לא להיבהל. חמושים במידע זה, ניגש למלאכת בניהת השרת שלנו. תראו שזה יגמר בתוך
כמה דקotas.

ראשית ניצור `http.Server`. איך אנו יודעים שדוקא אותו? כיון שהוא יורש מ-`net.Server` וכותב במפורש שהוא מיועד לייצור שירותים שיודעים לקבל תנועת TCP, שזו התנועה הרגילה. איך אנו מבצעים `require`? כמו כל מודול אחר, אבל כיון שב-`http` יש כמה מודולים, אנו ניאלץ לבצע `require` מ-`http`. לא מהו מיוחד במיוחד, זה מצוי במפורש בדוקומנטציה: `Class: http.Server`

ועכשיו? עכשו נבחן את המתודות שיש במודול `http` ונראה שיש מתודה `listen` שמקבלת פורט. מה זה פורט? פורט הוא בעצם סוג של "עוז" שדרכו אנו מנהלים תקשורת. אנחנו רואים את זה לעיתים גם בכתובות אינטרנט. אם פעם יצא לכם לראות כתובות בסגנון: [sites.com:3456](https://some-site.com:3456)

אנו תמיד משתמשים בפורט בכל תקשורת שהיא, אך כיון שברוב הפעמים אנו משתמשים בפורט ברירת מחדל, אנו לא רואים אותו בשורת הכתובות. הפורט הדיפולטיבי של הדפדפן ושל השירות אינטרנט הוא 80. אם היינו רואים את הפורט בעיניהם והדפדפן לא היה מסתיר את פורט ברירת המחדל מאיתנו, היינו רואים למשל <https://google.com:80>

אנו יכולים להשתמש בכל פורט שבא לנו. הבה נבחר ב-3000. זה אומר שאם נרצה להיכנס לשרת שלנו, נctrיך להקליד נקודתיים ואז 3000. נראה את זה בהמשך.

אבל זה לא מספיק. מה יקרה כשה邏輯 יתחבר לשרת שלנו? אנו צריכים שבכל מקרה של "חיבור", יוצג מהهو משתמש. עיון בדוקומנטציה של `http` ייתן את התשובה. אנו יכולים להשתמש באירוע `request`: https://nodejs.org/api/http.html#http_event_request – האירוע הזה מפעיל את הקולבק עם שני ארגומנטים – הבקשה של הלקוח, `request`, והותצאה שהשרת אמרה להחזיר, `response`. הוא גם קלאס משל עצמו ואני יכולים לבדוק את המתודות שלו. https://nodejs.org/api/http.html#http_class_http_serverresponse

מתודת אחת, שהוא `end`, היא מה שאנו מחפשים – היא סגרת את החיבור ומחזירה טקסט ללקוח.
הקוד ייראה כך:

```
const http = require('http').Server;

const myServer = new http();

myServer.listen(3000);

myServer.on('request', (request, response) => {
  response.end('Hello World!')
});
```

אם תשמרו אותו ב-`app.js` בתיקייה שלכם, תפעילו אותו באמצעות כניסה עם הטרמינל למיקום
של `app.js` והקלדה של `node app.js` ואז אנטר – תראו שהסקריפט של `node.js` לא עוצר. ה-
`listen` משאיר אותו פתוח. אם אתם רוצים להרוג את התהילה, תלחצו על כונטרול+C.

אם תנוווטו עם הדפדפן אל [Hello World!](http://localhost:3000/). בניתם את השרת
הראשון שלכם!

אפשר לבנות את זה באופן אלגנטי יותר, כמובן. אנחנו כבר ידעים שאירועים אפשר להפעיל
מתוך קלאס:

```
const httpServer = require('http').Server;

class MyServer extends httpServer {
  constructor() {
    super();
    this.listen(3000);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {
    response.end('Hello World!')
  }
}

const myServer = new MyServer();
```

מה שעשינו כאן הוא לקרוא ל-`http.Server` ליצור קלאס משלנו עם `super constructor`. ה-`super` נדרש כאן כי אנו יורשים מקלאס אחר. אנו מבצעים האזנה לפורט 3000. במקביל, אנו מצמידים את הפונקציה `requestHandler` לאירוע `request`. עשינו את זה בפרק הקודם. מה שנותר לנו לעשות הוא לאתחל את הקלאס שלנו. פשוט ונעiem.

טוב, האמת שזה לא כל כך פשוט – הקוד הזה הוא בן 14 שורות, אבל בשבייל ליצור אותו היינו צריכים לצולל לדוקומנטציות לא מעטות: לדוקומנטציה של תיאור `http.Server`, תיאור `net`, תיאור `response class` וCompanyIdן לעדעת מעולה מה אנחנו עושים. זה באמת מסובך! אבל מה שיפה ב-`Node.js` זה שאפשר להעביר קריירה שלמה מבלי לקרוא את הדוקומנטציה או לכתוב שורת שמשתמשות במודולים הבסיסיים של `Node.js`, כיוון שם תצטרכו למש שרת לא כתבו את השורות האלה אלא תשתמשו במודול קיים שנקרא `express` – מודול לבניית שרת אינטרנט שנלמד בהרחבה בהמשך הספר. מבלי לשבור את הראש ולהיכנס לעומקם של דברים. אנו לומדים את זה כאן כדי להבין יותר לעומק איך אירועים עובדים ואיך לקרוא לדוקומנטציה.

תרגיל:

אירוע `connection` שמתקיים בklass מסוג `http.Server` פועל בכל פעם שיש חיבור לשרת. צרו שרת, התחברו לאירוע זהה והדפיסו לקונסולה עדכון על חיבור בכל פעם שמיישמו מתחבר לשרת שלנו.

פתרונות:

```
const httpServer = require('http').Server;

class MyServer extends httpServer {
  constructor() {
    super();
    this.on('request', this.requestHandler);
    this.on('connection', this.connectionHandler);
    this.listen(3000);
  }
  requestHandler(request, response) {
    response.end('Hello World!')
  }
  connectionHandler() {
    console.log('Connection created!');
  }
}
```

```

    }
}

const myServer = new MyServer();

```

את השרת קל ליצור, יש לנו את זה בדוגמה בפרק. מה שאנחנו צריכים לעשות הוא ליצור פונקציה שתפעל בכל פעם שיש לנו אירוע. ניצור אותה כחלק מהklass שלנו:

```

connectionHandler() {
  console.log('Connection created!');
}

```

אחרי כן, ניצור פונקציה מסוימת עם `on` לאירוע `connection` ונחבר אותה ל-`socket`:

```

this.on('connection', this.connectionHandler);

```

בגלל זהה בתוך הקlass שלנו, ולא יושב בחוץ, אני משתמש ב-`this` כרפרנס לklass שלנו. קצת מסובך להבנה, אבל הקוד הרבה יותר קרייא.

אשמור את הקוד הזה ב-`app.js` וапעל אותו באמצעות ניווט בטרמינל אל מיקום הקובץ, הקלהה של `node app.js` וAz הקשה על אנטר. בכל פעם שאנווט אל: <http://localhost:3000/> אני אראה בקונסולה `.Connection created`

כדי לשים לב שבודגמה זו, כמו גם בשאר הדוגמאות, אנו מאזינים לפורט אחרי האירועים.

תרגיל:

צרו בתיקיית הפרויקט שלכם קובץ txt בשם test.txt עם תוכן מסוים. אפשר גם תוכן ארוך במיוחד (כמו למשל הטקסט הזה, המכיל את כל הקומדייה האלוהית מאת דנטה אליגיורי: <http://www.gutenberg.org/cache/epub/8795/pg8795.txt>). צרו קובץ app.js שבתוכו יש שרת http שמАЗין לפורט 3000. השרת שלכם יטען את הקובץ ויחזיר את תוכנו אל המשתמש. רמז: עשו זאת באמצעות fs.readFile שיימצא ב-requestHandler.

פתרון:

```
const HttpServer = require('http').Server;
const fs = require('fs');

class MyServer extends HttpServer {
  constructor() {
    super();
    this.on('request', this.requestHandler);
    this.listen(3000);
  }
  requestHandler(request, response) {

    fs.readFile('./test.txt', (err, data) => {
      response.end(data)
    });
  }
}

const myServer = new MyServer();
```

בדוק כמו בפתרון הקוד, אני יוצר קלאס שירוש את כל המתודות של httpServer. ב-constructor אני שם super, כי ג'אווהסקריפט דורשת זאת ממני. אחרי כן אני יוצר את השרת באמצעות методת listen ויצור מאזין לאיורע request כפי שיש בדוקומנטציה – בהזנה, במקום להציג מחוזות טקסט פשוטה, אני מכניס קריאה ל-fs.readFile כפי שלמדנו. התוכן שהוזר משם הוא התוכן שמזרם למethodת end.

אשמור את הקוד זהה ב-app.js ואפיעיל אותו באמצעות ניווט בטרמינל אל מיקום הקובץ, הקלדה של node app.js וاز הקשה על אונטר. אני אוכל לראות את תוכן הקובץ.

אם אתם רוצים לעשות זאת באופן אסינכרי ללא קולבקים – עושים זאת כך:

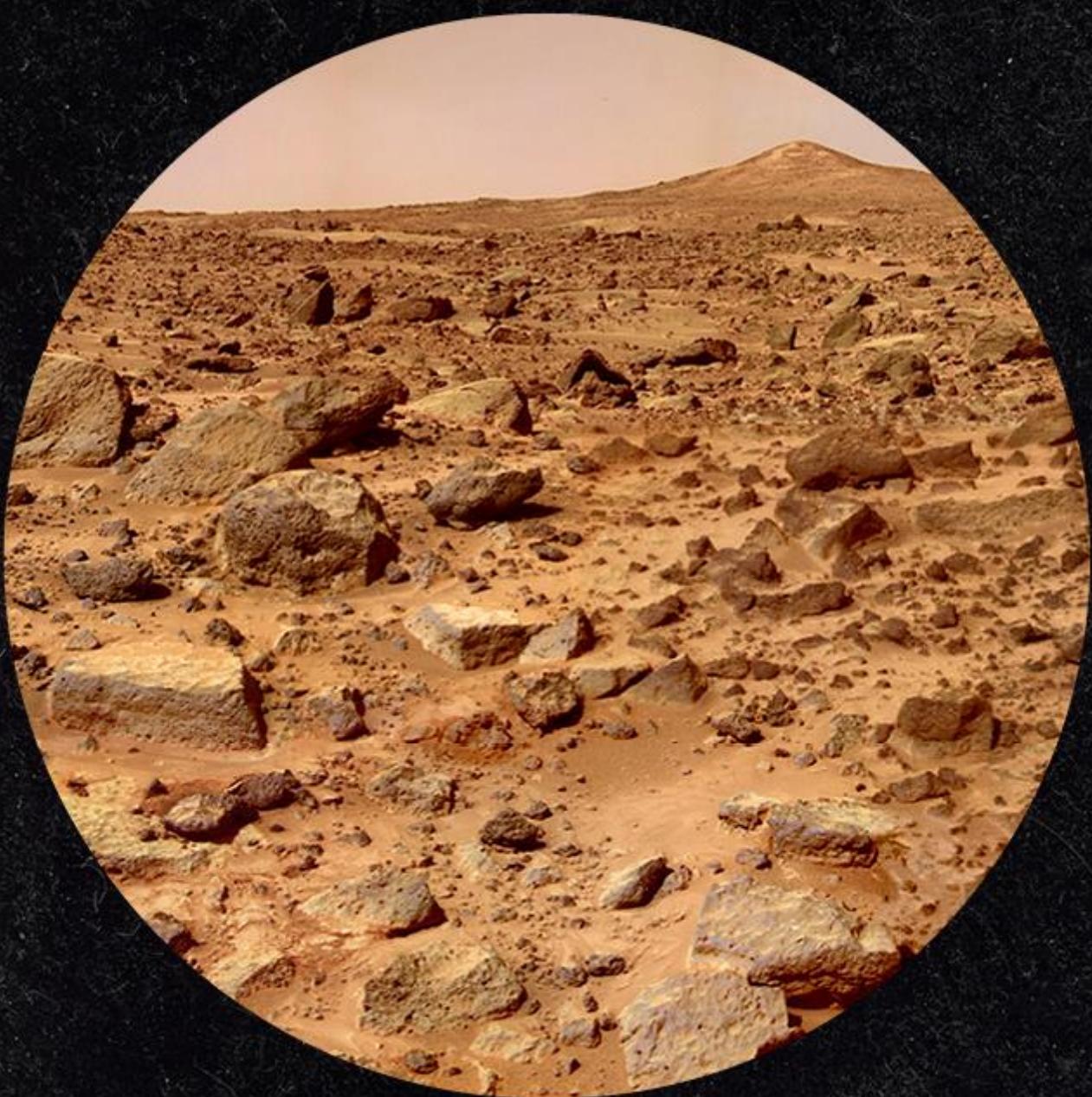
```
const HttpServer = require('http').Server;
const fs = require('fs');

class MyServer extends HttpServer {
  constructor() {
    super();
    this.on('request', (request, response) => {
      this.requestHandler(request, response) });
    this.listen(3000);
  }
  async requestHandler(request, response) {
    const data = await fs.promises.readFile('./test.txt');
    response.end(data);
  }
}

const myServer = new MyServer();
```

פרק 8

NODE.JS של EVENT LOOP ה-



ה-Node.js של Event Loop

מדובר באחד החלקים החשובים אך הוא עלול להיות קשה להבנה. חשוב מאוד להבין לעומק את הפרק המדובר על קולבקים ועל אירועים לפני שאתם מעמיקים לתוך פרק זה.

ראינו עד כה ש-Node.js עובדת בדרך כלל כאסינכרונית, אבל ג'אוوهסקריפט היא סינכרונית (כפי שŁמדנו מוקדם יותר בספר – מرتיצה את הפקודות לפי הסדר שבו הן נכתבו). הסיבה שבטעיה Node.js היא אסינכרונית היא בגלל הפורמייסטים והפלטפורמה של Node.js שמאפשרת לה לעבוד כך. הבנה של הפלטפורמה והדרך שבה היא מ滿מת את האסינכרוניות זו היא קריטית.

מתודות הטיימרים

ב-Node.js יש לנו כמה טיימרים – פונקציות שמסיימות לנו בתזמן הקוד שלנו. משתמשים בהם בכל מיני שימושים ש廣告ים בהמשך, אבל בינהם אסביר עליהם כאן.

תrox כשאני אומר לך – `setTimeout`

מדובר במתודה שמקבלת שלושה ארגומנטים. הראשון הוא קולבק שרצ – הקוד שאמור לרוץ כשאני אומר לו. השני הוא מספר המילישניות שייעברו עד שהקוד רץ, והשלישי הוא ארגומנטים שנייתן להעביר לקולבק זהה:

```
setTimeout((arg1) => {console.log(`Callback with ${arg1}`);}, 1000,
'arg 1');
```

זה לא קוד שאמור להפעיל אתכם מהכיסא בשלב זה. אם תרצו אותו עם Node.js תוכלו לראות את הפלט מופיע אחרי שנייה. 1,000 מילישניות = שנייה אחת. הפלט יהיה כמובן:

Callback with arg 1

ת្រוץ מייד עם קולבק – **setImmediate**

הfonקציה `setImmediate` זהה ל-`setTimeout` מלבד הבדל אחד – היא רצתה מייד ללא מילישניות. יש לה שני ארגומנטים: הקוד שרצץ והארגומנטים. ככה היא נראה:

```
setImmediate((arg1) => {console.log(`Callback with ${arg1}`);},  
'IMMEDIATE');
```

למה צריך פונקציה כזו? בהמשך נבין.

הlolאה קבועה **setInterval**

`setInterval` מקבלת גם היא שלושה ארגומנטים. הראשון הוא הקולבק, השני הוא מספר מילישניות והשלישי הוא ארגומנט. מה שהוא עושה הוא להריץ את הקולבק (והארגומנטים המתאימים) מדי כמה מילישניות. הלולאה זו תחזיר על עצמה לנצח כל עוד לא עצור אותה (אפשר לעצור אותה בכמה דרכים שלא אפרט כאן).

אם תרצוו את הקוד הזה, תוכלו לגלוות שהוא לא עוצר עד שתאתם לא הורגים את התהיליך באמצעות `kontrol + C`.

```
setInterval((arg1) => {console.log(`Callback with ${arg1}`);}, 500,  
'every 0.5 sec');
```

טור הקריאה

לא צריך להיות גאון גדול כדי להבין מה יודפס בטרמינל אם אני ארים את הקוד הבא:

```
console.log('First');
console.log('Second');
console.log('Last');
```

בפלט אני רואה:

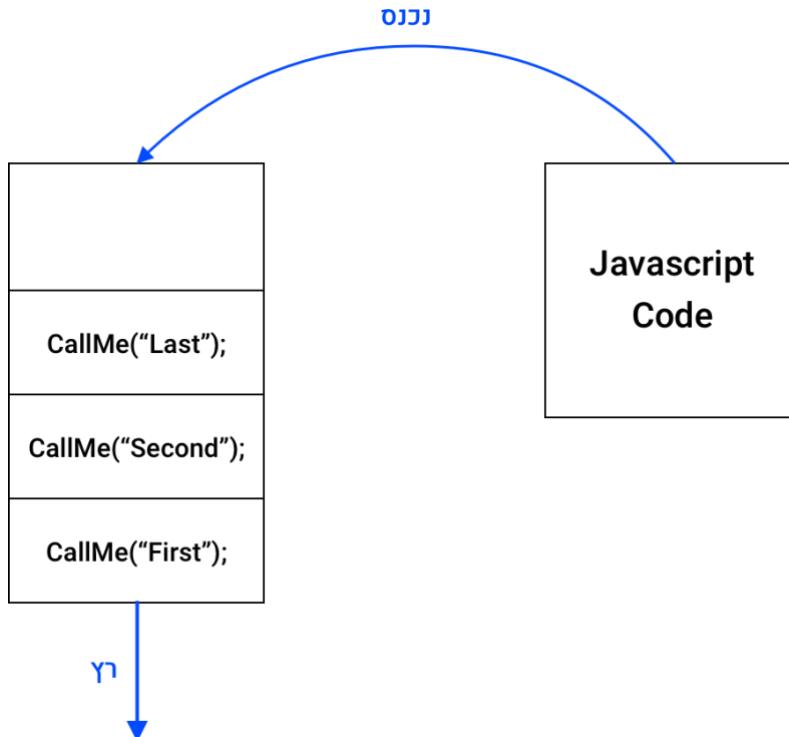
```
First
Second
Last
```

מה קורה מאחרוי הקלעים? איך ג'אווהסקריפט יודעת להרים את הקוד? מדובר בקוד סינכרוני, קוד שרצץ לפי הסדר מלמעלה למטה והוא בולם את המשך. ג'אווהסקריפט לא תמשיך לשורה השנייה
לפניהם שהשורה הראשונה מסתירה לroz' ולא תמשיך לשורה השלישית עד שהשנייה תושלם.

התור הזה נקרא "טור הקריאה" והוא יעבד גם עם פונקציות כמו זו:

```
function callMe (arg1) {
  console.log(arg1);
}

callMe('First');
callMe('Second');
callMe('Last');
```



זה ממש קל בקוד סינכרוני. אבל מה קורה כשיש לנו קטעי קוד אסינכרוניים? כמו טיימרים למשל (שלוש הפונקציות שראינו לעיל) או קטעי קוד אסינכרוניים שמטפלים בפלט ובקלט (O\A) כפי שלמדנו קודם עם File Server? כאן נכנסת לפולה לולאת האירועים של Node.js.

מה קורה כשיש לנו קוד מעורב זהה?
ראשית, הפקודות הסינכרניות מอรוצות לפי הסדר. אם יש קריאה למשק חיצוני, כמו קובץ או רשות – הן מורוצות והקובלים שלהן נכנסים לתוך הקובלקים של הפלט/קלט. אם יש פונקציות של טיימרים, הן נכנסות לתוך הטיימרים. אחרי שנכל הפקודות הסינכרניות מורוצות, רץ ה-loop על כל חלקיו:

בחלק הראשון רצים כל הטיימרים שזמן הגיע. בחלק השני – הקובלקים. החלק השלישי הוא פנימי ובחלק הרביעי רצות כל הבקשות שמנויות מבחוֹז (למשל שרת http://).

כדי להמחיש את זה נבחן את הקוד הבא:

צרו קובץ בשם `test.txt` בפרויקט שלכם, פתחו את `app.js` והדיביקו את הקוד הזה:

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

console.log('First');

//File I/O operation
fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 1');
});

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 2');
});

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 3');
});

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 4');
});

setTimeout(() => { console.log('setTimeout Finished'); }, 0);

console.log('Last');
```

הרכזו את הקוד. הפלט שיופיע הוא:

```
First
Last
setTimeout Finished
Reading data 1
Reading data 3
Reading data 4
Reading data 2
```

למה? ה-`First` וה-`Last` מגיעים מ-`console.log` סינכרוני שנמצא בקוד. הם לא נמצאים בתוך קולבקים, לא נמצאים בתוך טיימרים. סתם זרוקים בקוד. אז המנווע של `Node.js` מריצ' אותם מיד. את הקריאה לקובץ הוא גם מריצ', אך לא מחייבת תוצאה (מדובר בפעולה אסינכורונית). הוא מכניס

את הקולבקים ואת התוצאות שלהם כשיגיעו, לתוך תור הקולבקים ב-loop. Event Loop. הוא מכניס את הטיימרים שיש לנו – וווש לנו אחד – אל התור של הטיימר.

ועכשיו הלולאה נכנסת. קודם כולם Node.js בודקת את ענייני הטיימרים. מה יש בתור? רק setTimeout אחד – אם הזמן הגיע, הוא רץ. מה זה הזמן הגיע? האם עברו מספר המילישניות מרגע הרצת הקוד? במקרה הזה ציינתי אפס, אז הטיימר הזה יורץ. וכך? עכשיו לטור הקולבקים. אם יש קולבקים הם יירצו לפי הסדר שבו הם הושלמו. זו הסיבה שלא תמיד נראה 1,2,3,4 בפלט. אלא זה תלוי בסדר שבו מערכת הקבצים השלים את הקריאה השונות. כל קולבק נכנס לתור לפי הסדר שבו הוא מושלם ולא לפי הסדר שבו הוא נקרא.

הכל קורה באופן אסינכרי, כלומר אין שהוא שבולם את מעגל האירועים של Node.js אלא אם כן בקולבקים שמננו קוד כבד במיוחד – וזה אחת הסכנות הגדולות שבגלאן אני מלמד את זה בספר – בקולבקים לא-Amor להיות קוד כבד כי אחרת הוא ייחסם את המעגל וכל המעגל יתעכב בגלל קוד כבד. קוד כבד לא-Amor להתקיים ב-node.js ובמיוחד בשורותים המבוססים על Node.js. אם מנפחים יותר מדי את הקוד, המعالج ייתקע, התור של הקולבקים יתנפח ומתיישהו הסקריפט יקרום מחוסר זיכרון.

יש שלב במעגל האירועים שנקרא check – במעגל הזה setTimeout מופעלת. בואו נמחיש עם קוד נוסף. אם תרצו את הקוד הזה, למשל, שהוא קוד מורכב יותר, תוכלו לראות בפלט של הטרמינל איך הכל רץ:

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

console.log('First');

//File I/O operation
fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 1');
});

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 2');
});

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 3');
});
```

```
fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 4');
});

setImmediate(() => { console.log(`setImmediate Finished`); });

setTimeout(() => { console.log('setTimeout #1 Finished'); }, 0);

setTimeout(() => { console.log('setTimeout #2 Finished'); }, 5);

console.log('Last');
```

ראשית יירוץ הפלט הסינכרוני:

First

Last

וירוצו כל הבקשות אל ה-`FileSystems` – אבל כמובן Node.js לא מחייב שהן יושלמו כשה콜בקים יושלמו הם יגיעו לתוך ה-`Pending callbacks`. הקוד ממשיך ונכנס אל תוך הלולאה של האירועים, שם התחנה הראשונה היא שלב הטיימרים. יש שם שניים. רץ הטיימר הראשון:

`setTimeout #1 Finished`

כיוון שעברו 0 מילישניות מתחילת הריצת הקוד, הטימר השני לא ירוץ – עדין לא עברו 5 שניות מתחילת הריצת הקוד. אנו עוברים אל הקולבקים:

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

console.log('first');

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 1');
});

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 2');
});

fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 3');
});

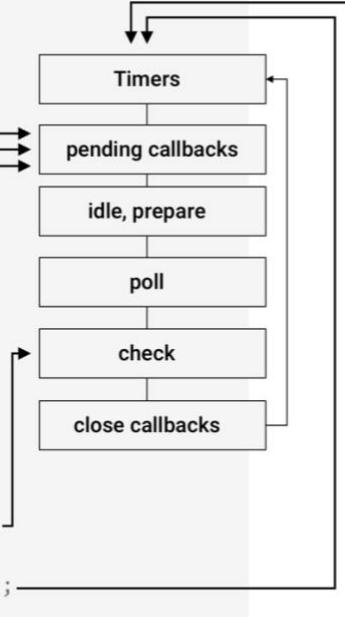
fs.readFile('test.txt', 'utf-8', (data) => {
  console.log('Reading data 4');
});

setImmediate(() => { console.log('setImmediate Finished'); });

setTimeout(() => { console.log('setTimeout #1 Finished') }, 0);

setTimeout(() => { console.log('setTimeout #2 Finished') }, 5);

console.log('last');
```



כיוון שבשלב זהה עוד שום קולבק לא הושלם, אנו עוברים אל שלב ה-**poll**. שם אין קוד נוסף ואני מגיעים לשלב ה-**check**, שבו ה-**setImmediate** עובד ונראה בפלט את:

setImmediate Finished

הלוואה סוגרת את מה שנוצר לסגורה ומתחילה מההתחלת. שלב הטימרים מגיע. אם בשלב זהה עברו 5 מילישניות (ה-**setTimeout** השני אמור לרוץ 5 מילישניות לאחר הפעלה שלו) אנו נראה:

setTimeout #2 Finished

אם לא, הקוד ירוץ מיד אל Pending callbacks וירץ את מה שהזמין מערכת הקבצים לפי סדר החזרה. ככלומר נוכל לראות שם:

Reading data 3
Reading data 1

אחריו שהתוර התרוקן, הלולאה ממשיכה לפעול, check ריק אז חוזרים לטיימרים. אם ה-setTimeout השני עדין לא רץ, הוא ירוץ עכשו כי בטח עברו 5 מילישניות. ואז הלואה אל ה-Pending callbacks יש שם מהו? מעולה. Node.js תריץ את הכל. אין? מצוין. חזרה לlolaha עד שכך אין שום קולבק שמחכה לחזור ואז הסקריפט ימות. הבדיקה הסתיימה והlolaha רצתה במלואה.

תרגיל:

מה יהיה לפניכם הפלט שתראו אם תרצו את הקוד זהה?

```
console.log('First');

setTimeout((arg1) => {console.log(`Callback with ${arg1}`);}, 1000,
'arg 1');

console.log('Last');
```

פתרונות:

First
Last
Callback with arg

התשובה היא ברורה למדי – ראשית הקוד הסינכרוני רץ, ורק אחריו כל מה שנכנס לlolaha האירועים של Node.js. במקרה זה רק setTimeout.

תרגיל:

מה יהיה לפि דעתכם הפלט שתראו אם תריצו את הקוד זהה?

```
console.log('First');

setImmediate(() => { console.log(`setImmediate Finished` ) });

setTimeout(() => { console.log('setTimeout Finished'); }, 0);

console.log('Last');
```

פתרונות:

```
First
Last
setTimeout Finished
setImmediate Finished
```

ראשית הקוד סינכרוני רצ, שתי הפקודות האחרות עוברות ללולאת האירועים. setTimeout נכנס לחלק של הטימרים ו-setImmediate לחלק של ה-check. הלולאה רצה בפעם הראשונה. כיוון ש-setTimeout אמור לפעול 0 מילישניות לאחר תחילת הסקריפט, הוא יעבד מיד. הלולאה ממשיכה לחלק של ה-check ומricane את setImmediate.

פרק 9

STREAMS



Streams

בפרק על ייצירת שרת http בסיסי הרأיתי את הקוד הבא:

```
const httpServer = require('http').Server;
const fs = require('fs');
const util = require('util');

class MyServer extends httpServer {
  constructor() {
    super();
    this.listen(3000);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {

    fs.readFile('./test.txt', (err, data) => {
      response.end(data)
    });
  }
}

const myServer = new MyServer();
```

הקוד הזה בעצם מזמן לארוע `request`, וברגע שהוא מתקיים הוא מבצע `fs.readFile`. הבעה עם `fs.readFile` שאמ `fs.readFile` לוקחת הרבה זמן, אם הקובץ גדול מזמן זה ייקח זמן. בעצם מה שיקרה פה זה שה-`fs.readFile` תיכנס לולאת האירועים של `Node.js` ועד שהקובץ לא יישלח, הלולאה תרוץ וזה עלול להושיב את הלוקוח מול מסך לבן במשך הרבה זמן.

זה בדוק כמו להוריד סרט לחלווטין ורק אז לראות אותו. דבר אפשרי בהחלט, אבל מייאש מאוד ועלול לקחת הרבה זמן. במצבות אלו מורידים בכל פעם חלק קטן של הסרט ומתחלים לראות אותו לפניו שכל הסרט ירד. כך אנחנו לא צריכים להמתין. אותה התנהגות יכולה לקרות פה בעקבות Streams. במקרה הזה אנו מונעים מהлокוח לבקש זמן יקר ולחייב לקובץ `txt` שייקרא במלואו מערכת הקבצים. אז אנו קוראים בכל פעם חלק קטן שנקרא `chunk` ואז מעבירים אותו אל

הלקוח, והלקוח מקבל אותו בחלקים. הלקוח יכול להיות משתמש במקרה שצופה בקובץ או איזושהי תוכנה אחרת.

סטריםים הם לא המצאיה ייחודית ל-node.js. למעשה, כל פלטפורמת צד שרת עשו את זה, אבל זה מהשו שעובד מעולה ב-node.js היישר מה קופסה. הנה נדגים ונראה איך זה עובד. הקוד הבא הוא של שרת שקורא קובץ בעזרת סטרים:

```
const httpServer = require('http').Server;
const fs = require('fs');

class MyServer extends httpServer {
  constructor() {
    super();
    this.listen(3000);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {
    const src = fs.createReadStream('./test.txt');
    src.pipe(response);
  }
}

const myServer = new MyServer();
```

שתי השורות היחידות שונות מהדוגמה ללא סטרים הן:

```
const src = fs.createReadStream('./test.txt');
src.pipe(response);
```

אני משתמש במתודה `createReadStream`. זו היא מתודה שמקבלת בדוגמה ארגומנט אחד בלבד – שם הקובץ שהוא אנו קוראים. המתודה הזאת נמצאת באופן טבעי ב-File Server, ממש כמו `readFile`. גם הארגומנטים שהיא מקבלת זהים לחולוטין `readFile` שעליו כבר למדנו, אבל בניגוד `readFile` אני לא משתמש בקובלבק אלא מעביר את התוצאה באמצעות `pipe` אל ה-`response` שהוא תגבות השרת. זה הכל!

כשאני יוצר סטרים אני בעצם יוצר בrz שمزרים מים, שאני חייב לחבר אותו לאנשו עם `pipe`. ממש כמו אינסטטטור. אני יוצר את בrz הנתונים המזרים נתוניים מ-`test.txt` וublisher אותם אל ה-`response`. במקרה ה-`response` יש `end`.

אבל מה שקרה מאחורי הקלעים הוא שהקובץ לא נטען במלואו לזיכרון ואז נשלח במלואו אל הלוקה דרך השרת, אלא הוא נטען בחלקים, וזה אומר שלולאת האירועים שלנו לא תיתקע כל כך מהר.

בואו נדגים שוב. הפעם עם שני סטרים. סטרים אחד שקורא – קלומר מספק את זרם הנתונים, וסטרים אחר שcribes – קלומר מתבב את זרם הנתונים החוצה:

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

const readStream = fs.createReadStream('./test.txt');
const writeStream = fs.createWriteStream('./out.txt');

readStream.pipe(writeStream);
```

כאן אני יוצר באמצעות מודול `File System` שני סטרים. הראשון הוא סטרים שקורא את הנתונים, סטרים לקריאה, והשני הוא סטרים שcribes את הנתונים, סטרים לכתיבה. בדוק כמו אינסטטטור אני לחבר את היצור שמצויא נתונים אל היצור שמצויה להם. למעשה מתבצעת העתקה של קובץ אחד לקובץ שני, אבל באופן שמקל על הזיכרון. במקרה לטען קובץ שלם אל הזיכרון ואז להעתיק אותו, אני עושה זאת בחלקים באופן שקויף וכל ביותר.

סוגי הסטרים השונים

יש כמה סוגים סטרים ב-`Node.js`, שניים מהם כבר ראיינו:
 הראשון הוא סטרים לקריאת נתונים, זה שהוא מקור הנתונים. הוא נקרא `Readable Stream`. הדוגמה שהשתמשנו בה היא `fs.createReadStream`.

השני הוא סטרים לכתיבה הנתונים, זה שידוע מה לעשות עם הנתונים שמנגנים אליו. הוא נקרא `Writable Stream`. ראיינו שני סוגי כאלה, `httpServer.response` ו-`fs.createWriteStream`.

נוסף על הסטרים שכבר ראיינו יש סטרים נוספים: סטרום דו-כיווני (Duplex) לכתיבה וקריאה כמו סוקט, שעליו נרჩיב בפרק על סוקטים, וסטרים שמבצע טרנספורמציה, קלומר הוא מקבל מידע, משנה אותו וublisher אותו להלה.

סטרים טרנספורמציה

כמו אינסטלטור שמחבר כמה צינורות, אני יכול לחבר שלושה צינורות: אחד שקורא את המידע ושולח אותו להלה, השני שמקבל את המידע, מבצע עיבוד שלו ושולח אותו להלה, והשלישי שמשתמש במידע. למשל – סטרים שקורא קובץ טקסט, סטרים טרנספורמציה שלוקח את הקובץ, מכועז אותו ב-kz וublisher אותו להלה וסטרים שכותב אותו. המודול הטבעי של Node.js שעסק בכיווץ נקרא `gzip` והוא מכועז בפורמט `kz` ואני יכולם להשתמש בו כדי לבצע כיווץ של קבצים, עם סטרים. איך זה נראה? בפשטות כך:

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

const readStream = fs.createReadStream('./test.txt');
const writeStream = fs.createWriteStream('./out.gz');
const gzip = zlib.createGzip();

readStream.pipe(gzip).pipe(writeStream);
```

אם תרצו את הקוד הזה תראו שנוצר لكم קובץ בשם `out.gz` שבו הוא קובץ מכועז. מה קורה פה? פשוט מאד – סטרום אחד קורא, סטרים שני מבצע כיווץ והשלישי כותב. הדבק ביניהם נעשה באמצעות `pipe`. זכרו את מטפורת האינסטלטור. זה בדיקות מה שאתם עושים.

אירועים בסטרים

סטרים מכליים אירועים שאפשר להשתמש בהם בקהלות כדי לבצע פעולות שונות בתחילת העברה, תוך כדי העברת הנתונים ובסיום העברת הנתונים. האירועים משתנים מסטרים קריאה לסטרים כתיבה.

סטרים כתיבה (כמו <code>fs.createWriteStream</code>)	סטרים קריאה (כמו <code>fs.createReadStream</code>)
תוקן כדי מעבר מידע drain	תוקן כדי מעבר מידע data
סיום כתיבה Finish	סיום קריאה end
שגיאה error	שגיאה error
סגירת הסטרים close	סגירת הסטרים close
מודיען עדין נמצא בסטרים readable	כאשר התחברו אל הסטרים עם <code>pipe</code>
	כאשר פונקציה אחרת ביצעה <code>unpipe</code> לסטרים

האירועים המשמעותיים ביותר הם בדרך כלל `end\finish` ו-`data\drain`.
 איך משתמשים באירועים? בדיקן כמה בכל מקום. יצרתי סטרים? עם המתודה `on` ושם האירוע אני יכול להאזין למידע. יש אירועים שמעבירים מידע (כמו `drain`) ויש כאלה שלא. איך אנו יודעים אילו? בדוקו מונטזיה מפורט לגבי כל אירוע ואירוע.

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

const readStream = fs.createReadStream('./test.txt');
const writeStream = fs.createWriteStream('./out.txt');

readStream.on('end', () => {
  console.log(`end!`);
});

writeStream.on('pipe', (data) => {
  console.log(data);
});

readStream.pipe(writeStream);
```

הינה דוגמה קצרה יותר מעניינת:

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

const readStream = fs.createReadStream('./test.txt');
const writeStream = fs.createWriteStream('./out.txt');

let dataLength = 0;

readStream.on('data', (chunk) => {
  dataLength += chunk.length;
})
readStream.on('end', () => { // done
  console.log(`The length was: ${dataLength} bytes`);
});

readStream.pipe(writeStream);
```

הכוח של סטרימים גדול מאוד, כי הוא מאפשר לי לטפל במידע ביבנאי או לא ביבנאי ממש בקלות ובלוי להתאמץ יותר מדי, מבלתי לחשב באפרים (Buffer). באפר זה בעצם "אזור הרמתנה" של הסטרים. כשהאנו יוצרים סטרים, הנתונים שלא מספיקים להיכנס אליו "מחכים" בעצם בבאפר. אבל כאמור אנו לא נדרשים להבין לעומק את עניין הבאפרים. יוצרים סטרימים ומחברים אותם עם פיפויים, ובאמצעות אירוחים אפשר לנטר או לבצע פעולות נוספות על המידע הזה. פשוט וקל.

תרגיל:

צרו קובץ קצר בשם test.txt בתיקייה שלכם. כתבו קובץ המעתיק את test.txt ל-out.txt באתרי סטרימים והכניסו אותו לקובץ בשם js.app. הפעילו אותו באמצעות js.node. מהטרמינל ובדקו שנוצר קובץ out.txt שה תוכן שלו הוא בדוק כמו test.txt.

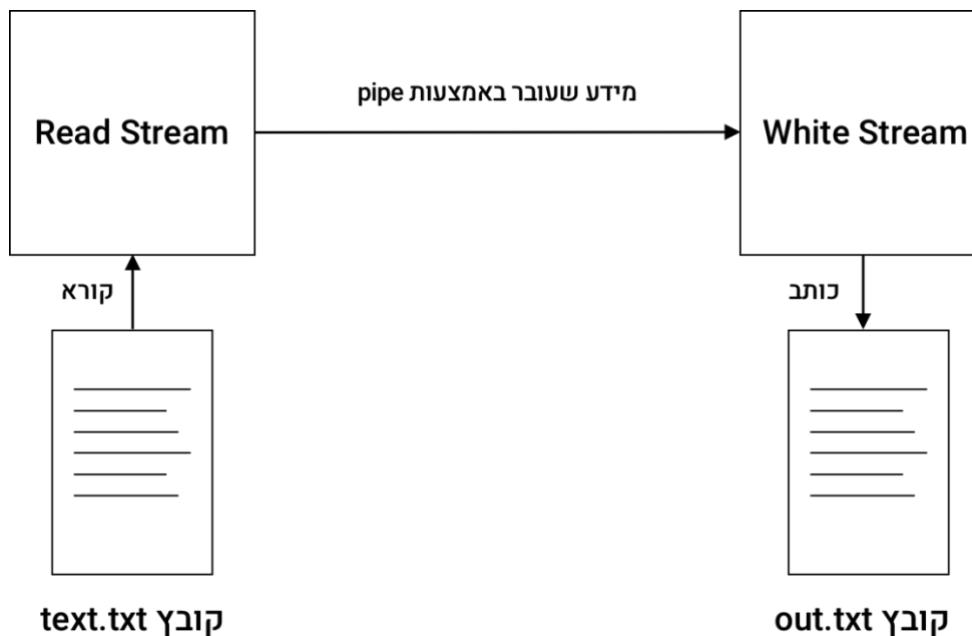
פתרונות:

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

const readStream = fs.createReadStream('./test.txt');
const writeStream = fs.createWriteStream('./out.txt');

readStream.pipe(writeStream);
```

הסבר: ראשית קראתי למודול fs. ל-fs, כפי שלמדנו בפרק ואנו רואים בדוקומנטציה, יש שתי מתודות: createReadStream ו-createWriteStream. המתודה הראשונה מקבלת את שמו של הקובץ ששמנו אנו קוראים. המתודה השנייה מקבלת את שמו של הקובץ שאליו אנו כותבים. כל מה שנוטר לעשות הוא לחבר את המתודות השונות באמצעות方法 pipe. הסטרים הקורא מתחבר לסטרים הכותב.



תרגיל:

יש צורך בהצפנה הקובץ. מנהל האבטחה נתן לך את ההגדרות הבאות:

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

const algorithm = 'aes-256-ctr';
const password = 'Password used to generate key';
const key = crypto.scryptSync(password, 'SomeSalt', 32);
const iv = 'myIVstringisnice1'.toString('hex').slice(0, 16);
```

צרו סטרים עם `.encrypted.txt` `test.txt` אל `createCipheriv` והצפינו את הקובץ מ-`test.txt`

פתרונות:

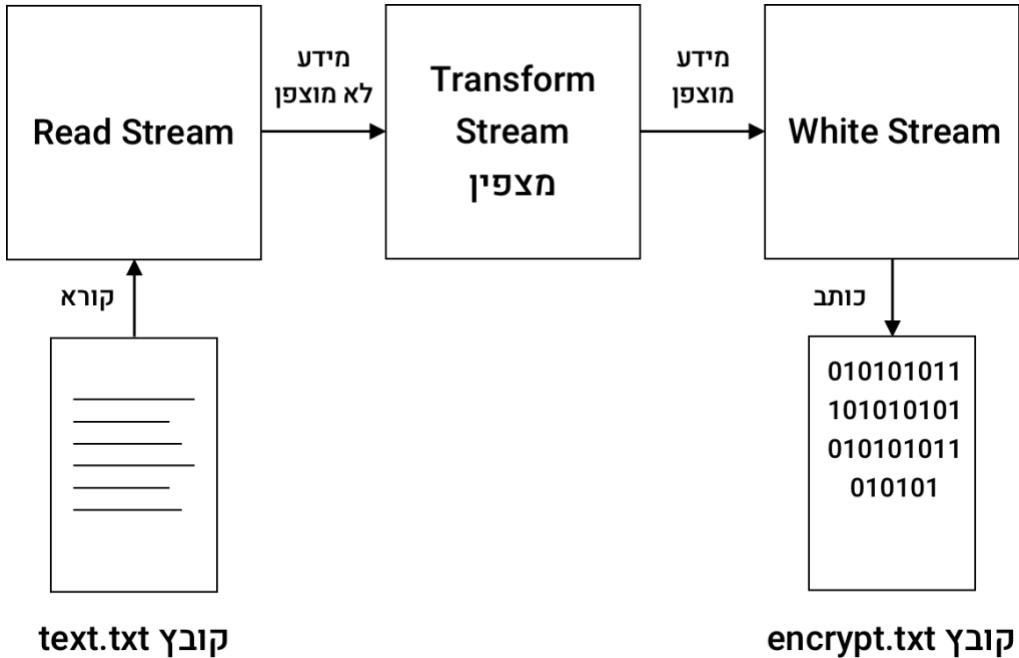
```
const fs = require('fs');
const crypto = require('crypto');

const algorithm = 'aes-256-ctr';
const password = 'Password used to generate key';
const key = crypto.scryptSync(password, 'SomeSalt', 32);
const iv = 'myIVstringisnice1'.toString('hex').slice(0, 16);

const readStream = fs.createReadStream('./test.txt');
const writeStream = fs.createWriteStream('./encrypted.txt');
const encryptStream = crypto.createCipheriv(algorithm, key, iv);

readStream.pipe(encryptStream).pipe(writeStream);
```

אם תבדקו בדוקומנטציה, תוכלו לראות ש-`createCipheriv` מקבל שלושה פרמטרים שיש בתרגיל: `algorithm`, `key` ו-`iv`. אם תעמיקו בקריאה תוכלו לראות ש-`createCipheriv` יוצר cipher שעובד בסטרים. כל מה שנותר לעשות הוא ליצור סטרים בדומה לדוגמה של ה-`blob` שיצרנו ולחבר את הכל עם `pipe`.



כדי לשימוש לב `sh-cryptSync` היא פונקציה סינכרונית חוסמת. אם נzieיב פונקציה כזו בחיקום האמיתיות, נסתכן בביטחון מהירות ויציבות.

תרגיל:

קחו את הקובץ המוצפן שיצרתם, קראו אותו באמצעות `readStream` וAz פענחו את הצופן באמצעות סטרים `createDecipheriv` אל קובץ `.decrypted.txt`

פתרונות:

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

const algorithm = 'aes-256-ctr';
const password = 'Password used to generate key';
const key = crypto.scryptSync(password, 'SomeSalt', 32);
const iv = 'myIVstringisnice1'.toString('hex').slice(0, 16);

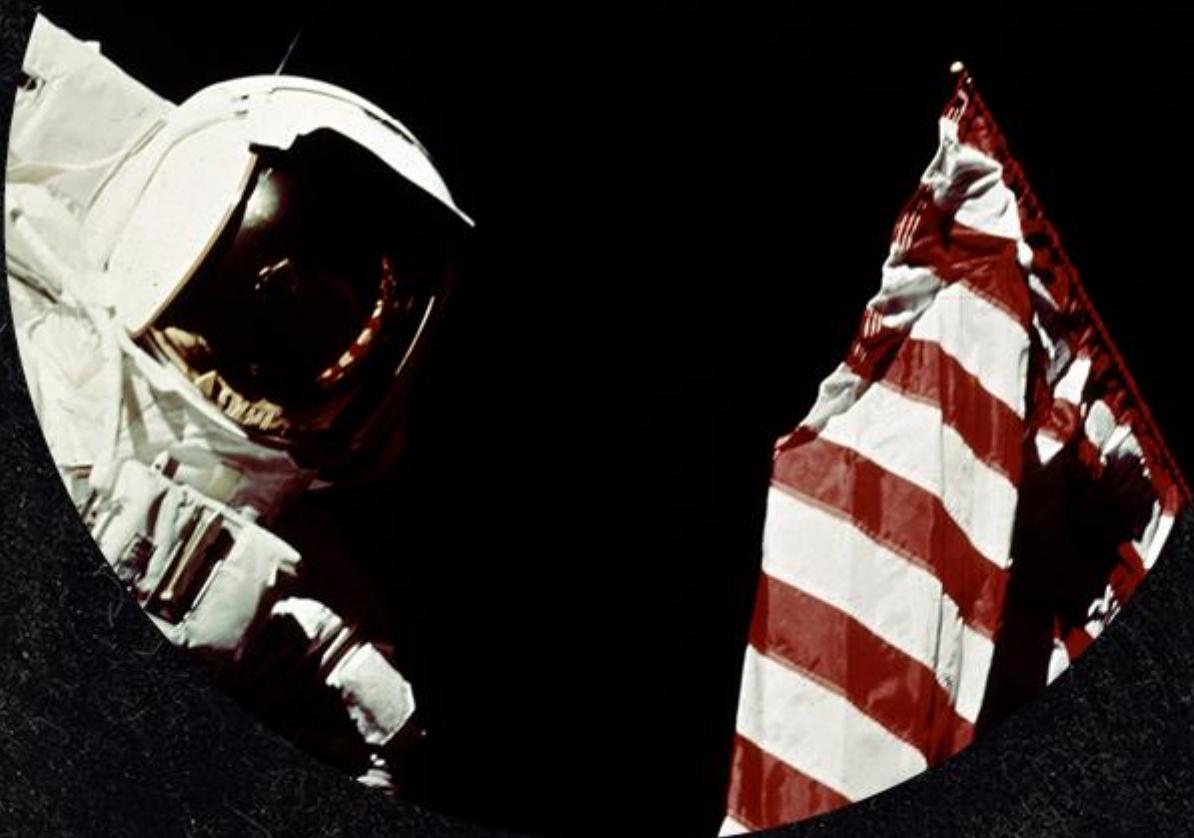
const readStream = fs.createReadStream('./encrypted.txt');
const decryptStream = crypto.createDecipheriv(algorithm, key, iv);
const writeStream = fs.createWriteStream('./decrypted.txt');

readStream.pipe(decryptStream).pipe(writeStream);
```

עלינו להזכיר `algorithm`, `key` ו-`iv` הם בדיק אוטם נתוניים שבהם השתמשנו כדי לבצע את ההצפנה. יוצרים את סטרים הפענוח בדיק באותו אופן כמו את סטרים ההצפנה. קוראים מ-`encrypted` וכותבים ל-`decrypted`.

פרק 10

אריזת הבוד שלנו נמושול



אריזת הקוד שלנו כמודול

עד כה השתמשנו במודולים – בין אם מדובר במודולים חיצוניים או במודולים של Node.js, הקוד שלנו הוכנס לתוך `js`.app והפעלנו אותו ישירות. אבל תמיד מתקבל לארוז את הקוד שלנו בתוך קלאס נאה שאפשר לעשות לו `require` והוא לא קסם אפל. הוא מבצע קריאה של קוד צדדי ומbia את מה שהוא קובץ מייצא.

בואו נראה זאת בעזרת דוגמה פשוטה. ניצור פרויקט שלנו תייקיה שנקראת `src` ובתוכה `module.js`. בתיקייה הראשית שלנו ניצור קובץ שנקרא `app.js`. בקובץ `app.js` ניצור את המודול הראשון שלנו, מודול שכל מה שהוא עושה הוא להכיל קבוע שווה 42.

```
const myNumber = 42;
```

אני רוצה להגיע למצב שם אני עושה ל-`js`-`module` פקודה `require` בקובץ אחר, כמו `js`.`app`, אני מקבל 42. ממשו בסגנון זהה:

```
const myModule = require('./src/module.js');

console.log(myModule); // 42
```

אני מיצא דברים באמצעות אובייקט מיוחד ל-`js`-`module` שנקרא `module.exports`. הוא נמצא ב-`js`.`Node` כאובייקט גלובלי ומה שנכנס אליו יוצא ב-`require` – אם אני אזכיר את המשתנה שלי – זה שנמצא בקובץ המודול שלי `module.js` – ל-`module.exports`, אני מקבל אותו בכל פעם שאני אעשה `require`. כך זה נראה בקובץ של המודול. למשל `module.js`:

```
const myNumber = 42;

module.exports = myNumber;
```

ואם תשמרו את הקוד שלעיל ב-`js`.`module`, תיכנסו לקובץ אחר ותכתבו:

```
const myModule = require('./src/module.js');
```

תראו שב-`myModule` או בכל משתנה אחר שמקבל את ה-`require` מקבלו 42.

ואפשר כמוון להשתמש בו כמה פעמים. כך למשל:

```
const myModule = require('./src/module.js');

console.log(myModule); // 42
console.log(myModule); // 42
console.log(myModule); // 42
```

כאמור, כל מה שנכנס ל-module.exports י יצא מהצד השני. כך למשל, אני יכול להכניס אובייקט שלם:

```
const oneWhoKnows = {
  1: 'God',
  2: 'Lochet Habrit',
  3: 'Fathers',
  4: 'Mothers',
}

module.exports = oneWhoKnows;
```

ואם אני אבצע לו require, אני אוכל לראות את האובייקט הזה.
אני יכול לעשות export לקלאס שלם, כמו למשל הקלאס שיצר שרת אינטרנט, שהראיתי את הקוד שלו בפרק הקודם. אם אני אשווה אותו ב-myModule.js, אני אוכל ליזא אותו בקלות כך:

```
const httpServer = require('http').Server;

class MyServer extends httpServer {
  constructor() {
    super();
    this.listen(3000);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {

    response.end('Hello World')
  }
}

module.exports = MyServer;
```

ואיך אצורך אותו? בדיק כמה משתנה, ככה:

```
const myModule = require('./src/module.js');  
myServer = new myModule();
```

אני יכול לבצע את היזירה של השירות באמצעות `new` במודול עצמו. למשל כך:

```
const httpServer = require('http').Server;

class MyServer extends httpServer {
  constructor() {
    super();
    this.listen(3000);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {
    response.end('Hello World')
  }
}

const myServer = new MyServer();

module.exports.myServer = myServer;
```

ואז כשאני עושה

```
require('./src/module.js');
```

מ-`js.app` וمفועל אותו עם `node app`, הוא יופעל בקלות.
יש דרך נוספת לצרוך מודולים ב-`js.Node`, על ידי `import` או `export`, אך אני לא מלמד אותה בספר זה.

תרגיל:

קחו את הפתרון לתרגיל בפרק על בקשת http והפכו אותו למודול Shiobash בקובץ נפרד. נסו לעשות זאת עם קלאס:

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

request('http://www.google.com', function (error, response, body) {
  console.log('error:', error); // Print the error if one occurred
  console.log('statusCode:', response && response.statusCode); // 
Print the response status code if a response was received
  console.log('body:', body); // Print the HTML for the Google
homepage.
});
```

פתרון:

```
const request = require('request');
class GoogleCaller {
  callGoogle() {
    request('http://www.google.com', function (error, response,
body) {
      console.log('error:', error); // Print the error if one
occurred
      console.log('statusCode:', response && response.statusCode); //
Print the response status code if a response was received
      console.log('body:', body); // Print the HTML for the Google
homepage.
    });
  }
}

module.exports = new GoogleCaller();
```

וההפעלה מתבצעת עמו:

```
const googleCaller = require('./src/module.js');
googleCaller.callGoogle();
```

על הקוד שמבצע את הקריאה ל-google.com כבר למדנו ואין טעם לחזור אליו. בחרתי להכניס אותו אל תוך קלאס. את הפונקציה הבונה של הקלאס זהה ייצאתי החוצה באמצעות module.exports ומInSection, כל מי שיעשה require לקובץ זהה, יקבל בחזרה את:

```
new GoogleCaller();
```

כאילו כתבתי את הכל באותן קובץ.

פרק 11

קביעות גרסאות



קביעת גרסאות

בפרק על NPM ומודולים חיצוניים הסבירתי שמדובר באחת החוזקות הגדולות של package.json. היכולת שלנו ליצור פרויקט עם "הוראות התקנה" למודולים חיצוניים היא הנדרת. כשאנו מعتبرים או משתמשים בפרויקט, אנו לא צריכים להעביר גם את המודולים החיצוניים, אלו שמאוחסנים ב-package, אלא רק את הקובץ node_modules המכיל את רשימת המודולים החיצוניים וגם מידע על הקוד שלנו.

אם אני מתakin את המודולים החיצוניים chalk, lodash ו request באמצעות:

```
npm i chalk
npm i lodash
npm i request
```

הם יותקנו בספריית node_modules ואני אראה ב-package.json את הגרסאות שלהם:

```
"dependencies": {
  "chalk": "^2.4.2",
  "lodash": "^4.17.11",
  "request": "^2.88.0"
}
```

כשאני מעביר את הקוד שלי למשהו אחר – בין שמדוברblkoch, לחבר נוסף בצוות או בשרת – אני לא צריך להעתיק את כל קובצי המודולים, אני רק צריך לוודא שלפרויקט שלי יש package.json בתיקיית האב שלו.blkoch יוכל לעשות npm install ותוכנת NPM תסתכם על package.json ותדע לקרוא ממנו את גרסאות המודולים והשמות שלהם ולבצע התקנה נוספת. על מנת לבדוק זאת, העתיקו את הקובץ package.json זהה אל תיקייה נטוה במחשב שלכם.

```
{
  "name": "node_projects",
  "version": "1.0.0",
  "description": "My first project",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC",
  "dependencies": {
    "chalk": "^2.4.2",
    "lodash": "^4.17.11",
    "request": "^2.88.0"
  }
}
```

היכנסו עם הטרמינל אל התיקייה וכתבו שם או וקח. תראו איך תוכנת NPM מתקינה את כל המודולים האלה בתיקיית `node_modules`.
אנו יכולים לקבוע את הגרסאות של המודולים השונים וכמוהן גם את הגרסה של המודול שלנו.

גרסאות סמנטיות

לכל תוכנה שהיא יש גרסאות – בין שמדובר במודולים של Node.js או במודול שאנו כותבים בעצמנו, או בכלל בתוכנות שלא קשורות לג'אווהסקרייפט. באקויסיטם של Node.js אנו עובדים לפי גרסאות סמנטיות. מדובר בסטנדרט של ממש שיש לו גם תרגום לעברית באתר הרשמי. אתם מוזמנים להיכנס ולקרוא באתר הרשמי: <https://semver.org/lang/he/> אבל אני אפרט גם כאן בקצרה.

גרסת תוכנה היא מספר שיש לו שלושה חלקים. למשל:

4.17.11

החלק הראשון, במקרה זה 4, הוא הינו **מייג'ור (Major)** – גרסה ראשית. כמשמעותם גרסה צו – שוברים API. ככל יותר משתנים/מוחקים מתודות או התנהוגיות. כמשמעותם מודול בגרסה

ראשית, נראה מהו יישבר בתוכנה שלנו. הדגש הוא על "כנראה". לא תמיד שוברים API בקידום של גרסה ראשית; יש כאלה המתכוונים את ה-API שלהם בצורה נכונה וgmtisha ויכולים לעשות שינויים גדולים גם בלי לשבור או לשנות התנהגות. אבל בקידום גרסה ראשית המפתח של המודול מאותת למי שבונה על הגרסאות האלו שצרכיך להיערך לכך.

החלק השני, במקרה זה 17, הוא הכנוי למינור (Minor) – גרסה משנה. כشمקדמים גרסה זו מבצעים שינוי מהותי בתוכנה – אבל עם תאימות לאחר. קלומר מהו לא יישבר בתוכנה שלנו אם נקדם את הגרסה הזאת, אבל יש סיכון שנראה אחרת.

החלק השלישי, במקרה זה 11, הוא הכנוי לפאטץ' (Patch) – תיקון באגים. כشمקדמים גרסה זו מתקנים באגים/בעיות אבטחה או משפכים פונקציונליות – קלומר קידום של הגרסה זו לא יעשה לנו בעיות. בואו נדגים. נניח שיש לי מודול זהה:

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

class GoogleCaller {

  callGoogle() {
    request('http://www.google.com', function (error, response,
body) {
      console.log('error:', error); // Print the error if one
occurred
      console.log('statusCode:', response && response.statusCode);
// Print the response status code if a response was received
      console.log('body:', body); // Print the HTML for the Google
homepage
    });
  }
}

module.exports = new GoogleCaller();
```

זה המודול שהוא בתרגיל בפרק על ייצור מודול משלנו. הגרסה שלנו היא:

1.0.0

אם החלטתי לתקן הערת כלשהו בקוד, למשל במקרה:

```
// Print the HTML for the Google homepage.
```

לכתוב:

```
// Print the HTML for http://google.com
```

از כשאני אוציה את הגרסה שלי ל-NPM או לגיטהאב או לכל מקום אחר, אני אתקן את הגרסה
לגרסה 1.0.1, ככלומר אקדמי את הגרסה בפאתץ', כי לא השתנה ממשו מהותי.

אם החלטתי לשנות את המתודה המרכזית שלי ל-`callGoogle` שמקבלת פרמטר `google`
(אולי כדי לתמוך באתרים נוספים בעתיד) אבל אני כן שומר את `callGoogle` כדי שתיהה תמיינה
למשתמשים אחרים, ככלומר ממשו זהה:

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

class GoogleCaller {

  call(site) {
    switch (site) {
      case 'Google':
        request('http://www.google.com', function (error, response,
body) {
          console.log('error:', error); // Print the error if one
occurred
          console.log('statusCode:', response &&
response.statusCode); // Print the response status code if a
response was received
          console.log('body:', body); // Print the HTML for the
Google homepage.
        });
    }
  }

  callGoogle() {
```

```

    console.warn('callGoogle is deprecated!, use call(\'Google\')');
    this.call('Google');
}

module.exports = new GoogleCaller();

```

מי שנסמך על `callGoogle` והשתמש בו במודול שלו באופן זהה:

```

const googleCaller = require('./src/module.js');
googleCaller.callGoogle();

```

הקוד שלו יעבד, אבל הוא יקבל התראה שהmethodה עומדת להשתנות. אני שיניתי בעצם קוד מהותי באפליקציה אבל עם תאיות לאחר. זהו שינוי מיינור. אני אשנה את הגרסה שלי ל-1.1.0 במקומם 1.0.0.

כשאני אחלייט להוריד את `callGoogle` לחלוטין, מי שנסמך על methodה זו במודול שלי יצטרך לשנות אותו. במידה שלא, הקוד לא יעבד. ככלומר ביצעת שינוי מהותי בלי תאיות לאחר. פה אני אהיה חייב לקדם את גרסת המיג'ור שלי ל-2.0.0.

אם אתם מיצרים מודול LNPM, כנה אתם-Amורים לעבוד וכן המודולים האחרים עובדים. ככלומר – אם אתם משתמשים על `request` – אין לכם בעיה לעדכן פאצ'ים, אין לכם בעיה לעדכן מיינוריים – אבל כן כדאי שתבדקו היטב בדוקומנטציה של המודול ובווד אם אתם רוצחים לעדכן מיג'ור.

קביעת גרסאות סמנטיות ב-`package.json`

וחזרה ל-`package.json`. אם ב-`package.json` יש גרסה מדוקית של המודול, ככלומר מספר לא תוספת של כובע או טilda או כל סימן אחר – כאשר אני או כל משתמש אחר עושה זוםק לקוד שלכם, אז LNPM תתקין את הגרסאות המדוקינות ביותר.

אם יש כובע (הסימן ^ או caret באנגלית) זה אומר LNPM תתקין את כל הגרסאות שתואמות לגרסה המצוינת. ככלומר אם יהיה שינוי בפאצ' או מיינור, היא תתקין אותו.

מה זאת אומרת? זאת אומרת שאם יש לי פירוט גרסה כזו:

`"lodash": "^4.17.11",`

אם אני כותב או npm ו-NPM תראה שהגרסה האחרונה של lodash היא 4.17.13, היא תתקין את הגרסה האחרונה. גם גרסה X.4.17.13 (X אומר כל מספר), כיוון שהוא לא שובר פונקציונליות. אבל אם הגרסה האחרונה היא X.5.0 אז npm לא תתקין אותה אלא את הגרסה האחרונה של X.4. אם יש טילדה (הסימן ~ או tilde באנגלית) זה אומר שה-NPM תתקין את הגרסאות של הפתאצ'ים בלבד, ולא של המינוריים. כאמור, אם יש לי פירוט גרסה כזו:

`"lodash": "~4.17.11",`

אם אני מבצע התקנה מאפס של המוצר עם או npm ו-NPM תראה שהגרסה של lodash היא גרסה 4.17.12 ומעלה, היא תתקין את הגרסה היותר חדשה, אבל היא לא תתקין את X.4.18. ובוודאי לא את X.X.最新: אפשר ל לנכט על הקצה ולצין בפניהם npm להתקין את הגרסה היותר חדשה באמצעות הטקסט

`"lodash": "latest"`

זה אומר שה-NPM תבצע התקנה של הגרסה החדשה ביותר ועוד אתם מסתכנים שהכל יישבר, ומאוד לא מקובל בתעשייה לעשות שיטות כאלה.

יש עוד אפשרות לקביעת גרסאות, אך אלו הנפוצות ביותר. רוב האנשים משתמשים בברירת המחדל של npm – ה"קובע" – עדכון של המינוריים והפתאצ'ים בלבד.

תרגיל:

קבעו את הגרסה של lodash בפרויקט שלכם ל-3.3.0. התקינו אותה עם npm ובדקו את הגרסה באמצעות כניסה ל-`node_modules`, משם לתיקיית `lodash` והצחה במספר הגרסה ב-`lodash/package.json` של `package.json`.

פתרונות:

```
{
  "name": "node_projects",
  "version": "1.0.0",
  "description": "My first project",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC",
  "dependencies": {
    "lodash": "3.3.0"
  }
}
```

אני מקבע את מספר הגרסה ב-`package.json` של הפרויקט שלי. אחרי שאני שומר את הקובץ אני מקליד בטרמינל, בעודי מכפיד להיות במקום של הפרויקט שלי, את הפקודה `npm`, ואז אנטר. NPM תבודוק את מספר הגרסה והתקין את 3.3.0. אם תבדקו את `lodash` שנמצאת ב-`node_modules` תוכלו לראות שהיא עודכנה ל-3.3.0.

פרק 12

התקנה גלובלית ו-נו



התקנה גלובלית CLI

עד כה עבדנו בעיקר מול הקונסולה ויצרנו גם שרת http. חלק מהעובדת המרכזית עם Node.js היא עבודה בכלים בטרמינל או **CLI** – ראשי תיבות של **Command Line Interface**. נשמע מעט מפחד אבל CLI הוא בעצם כתיבת פקודות בטרמינל. כתבתם dir (בחלונות) או al- ls (בלינוקס)? ברכוטי! – השתמשתם ב-CLI, סוג של, לפחות. בלינוקס יותר מקובל להשתמש ב-CLI לביצוע מטלות, ודאי וודאי בשרתים מבוססי לינוקס.

חלק גדול מכל ה-CLI יכולים להכתב על ידי node. אחד הכלים המפורטים ביותר הוא eslint,CLI שבודק תקינות קוד ג'אווהסקריפט. בוואו נתקן אותו בפרויקט שלנו. נכתוב בפרויקט:

```
npm i eslint
```

מיד תתווסף ל-package.json שלנו ול-dependencies, הגרסה الأخيرة של eslint עם "covع".
כלומר אם אני אעשה שוב בעtid ? npm – הגרסה الأخيرة של eslint שתואמת לגרסה המצוינת תותקן.

כדי להפעיל את eslint, אני חייב להפעיל את הטרמינל. הרוי eslint היא CLI והוא עובדת מהטרמינל בלבד. אנו נפעיל אותה בהקלדת הטקסט זהה בשורת הפקודה כאשר אנו בפרויקט שלנו:

```
node .\node_modules\eslint\bin\eslint.js . --no-eslintrc
```

אנו מתחילה ב-node כדי להריץ הכל בסביבה.js.Node ואחריו כן אנו מקlidים את הנתיב של eslint שהותקנה ב-node_modules. הנתיב הזה הוא CLI של eslint. אחר כך יופיעו שם הקובץ שאנו רוצים לבדוק (במקרה זה app.js. שאני מניח שיש לכם) והאפשרות --eslintrc שאמורת eslint לא להסתמך על קובץ קונפיגורציה. אם תכתבו קצר גיבריש בשורה הראשונה בקובץ, הקלדת הפקודה זו תיתן לכם שגיאה.

פתחי ג'אווהסקריפט משתמשים ב-`eslint` במהלך כתיבת קוד על מנת לוודא את תקינות הקוד שלהם – אבל זה מתייש לכתוב בכל פעם:

```
node .\node_modules\eslint\bin\eslint.js
```

בדוק בשבייל זה יש לנו התקינה גלובלית. בהתקינה הגלובלית אנו בעצם הופכים מודול js.js שעובד ב-CLI לגלובי. ככלומר אני יכול להפעיל אותו מכל מקום בטרמינל. מכל מקום! איך עושים את זה? כותבים (בכל ספרייה שהיא) בטרמינל:

```
npm i eslint -g
```

הפלאג (flag) הוא כינוי לפקודה בטרמינל שבאה לאחר התו "-". -g אומר ל-NPM להתקין את `eslint` באופן גלובי. NPM מתקינה את המודול לא בתיקייה של הפרויקט, אלא בתיקייה גלובלית במחשב ומקשרת (עם PATH בחלונות או סימリンク בליינוקס) את התקינה זו למערכת הפעלה כך שאפשר יהיה להפעיל את המודול מכל מקום שהוא.

התקין את `eslint` באופן גלובי וגושו לתיקייה שבה NPM מתקינה את המודולים הגלובליים. מוצאים אותה באמצעות הקלדה של

```
npm list -g
```

הפקודה זו מדפיסה היכן המודולים הגלובליים נמצאים וגם אילו מודולים הותקנו (עם התלוויות שלהם). אתם יכולים לגשת אל התקינה זו ולראות ש-`eslint` הותקנה בה. מה זאת אומרת? עכשו תוכלו להקליד `eslint` מכל מקום. הקlidו -v `eslint` בכל תיקיה שהוא ותוכלו לראות ש-`eslint` עובדת מכל תיקיה באופן גלובי. עכשו אתם יכולים להשתמש בכל זהה בכל מקום במחשב שלכם.

יש המון כלי CLI באkosיסטם של Node.js, כולל כלים שימושיים סביבות מלאות למגרוי כמו `create-react-app` או `angular cli`. קל להתקין אותם, קל לעבוד איתם והם חלק מהכח של Node.js. זה נחר.

מצד שני זה עלול להיות בדיתי – למלא את המחשב שלנו במידע שאין לנו לצריכים. נוסף על כך המודולים האלה לא נכנסים ל-`node_modules`. לפיכך התקנה גלובלית אינה מומלצת. מה הבחירה? `npx eslint`.

דרך נוספת לעבוד עם מודולים גלובליים היא באמצעות פקודה של NPM שנקראת `npm`. הפקודה זו מופעלת באמצעות `npm` ושם המודול, כמו למשל:

```
npx eslint
```

ובהרצה שלה, NPM בודקת אם יש מודול מקומי ב-`node_modules` של הפרויקט. אם כן, היא מפעילה אותו. אם לא, היא תבודוק אם יש מודול גלובלי. אם יש, היא תפעיל אותו ואם אין, היא תפעיל אותו. זה מעולה אם יש לכם פרויקטים שימושיים בגרסאות שונות של `eslint` או `CLI` אחרים. כפי שהזכירתי קודם, מומלץ מאוד להשתמש ב-`npm` ולא בהתקנה גלובלית.

תרגיל:

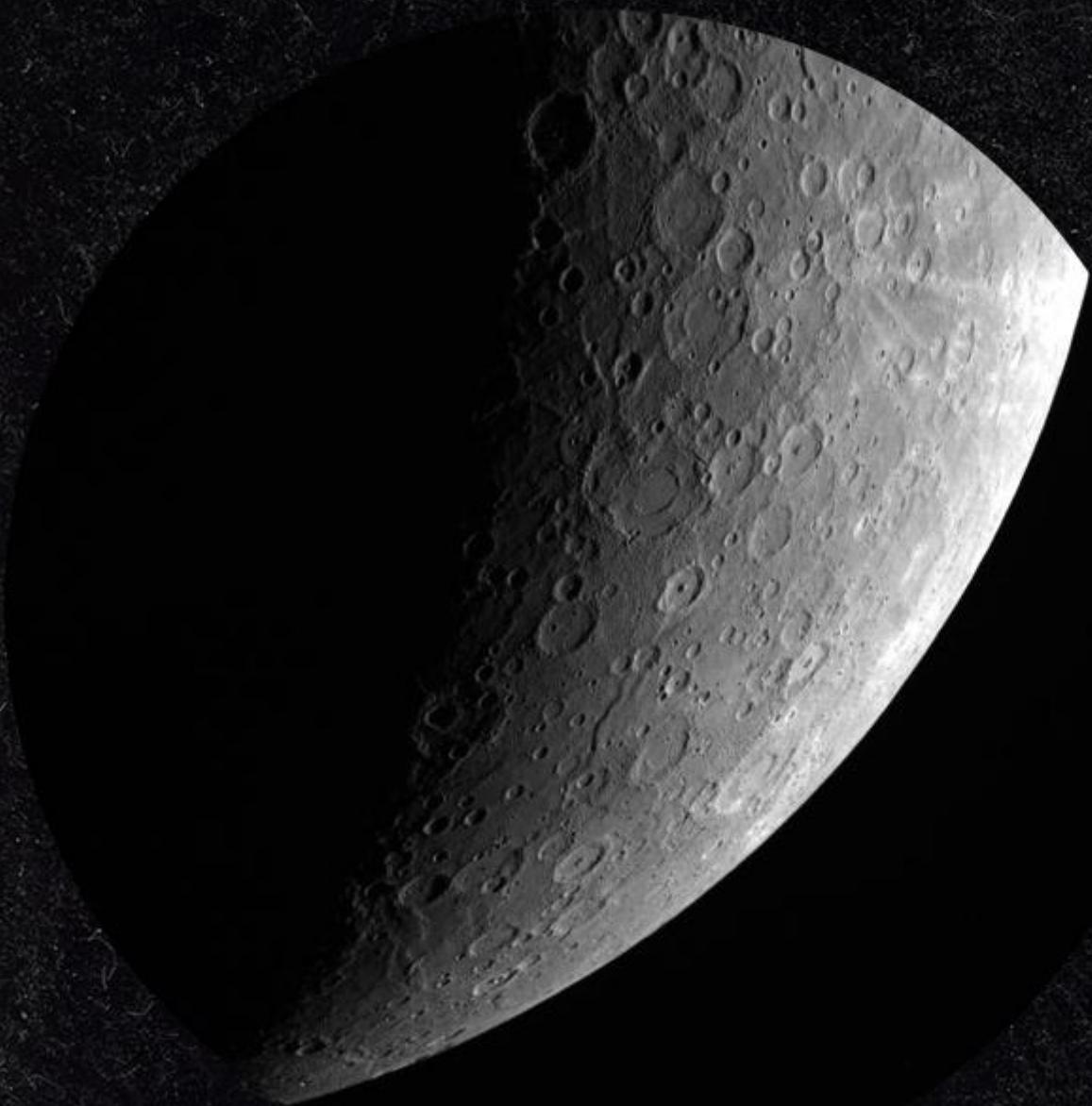
התקינו את מודול `xo` באופן גלובלי. מדובר במודול שדומה ל-`eslint`. בדקו שהואעובד על ידי הריצה של `xo` מכל תיקייה שהיא. הדוקומנטציה של המודול נמצאת פה: <https://www.npmjs.com/package/xo>

פתרון:

התקנת המודול נעשית באמצעות `npm install xo`. מהרגע שהקלידתם את הפקודה זו והקשרם על אונט, תוכלו להשתמש ב-`xo` באמצעות הקלדה של `xo` בלבד בכל מקום שהוא. אם אתם משתמשים בחלונות, יש סיכוי שתצטרכו לסגור ולפתח את הטרמינל כדי שהשינויים ב-`PATH` יעבדו.

פרק 13

כתייבת און והתמונשות עם ה-CLI



כתיבת bin והتمמשקות עם ה-CLI

קל מאד לנתח בCLI ב-node.js. יש לא מעט מודולים שימושיים בתחום ומאפשרים ליצור דיאלוגים, אнимציות וצבעים בטרמינל. אחד מהם כבר הכרתם בפרק הקודמים: המודול chalk שמאפשר לצבוע את הטרמינל בצבעים עלייזים.

כאשר אנו כותבים CLI, אנו צריכים להתmeshק עם שורת הפקודה. בדרך כלל אנו כותבים את הפקודה ועוד ארגומנטים. למשל הפקודה:

```
cd ..
```

שהיא בטרמינל ומאפשרת לנו להגיע אל תיקיית האב. היא מורכבת משניים – הפקודה עצמה (cd) והารגומנט .. שהוא בעצם "עליה לתיקיית האב". כשההשתמשנו ב-NPM, השתמשנו בכמה ארגומנטים, למשל: g- npm i eslint,i,eslint,g-. הם ארגומנטים. בואו נכתוב את ה-CLI הראשון שלנו. ה-CLI שלנו הוא פשוט למדי. הפקודה שלנו תהיה encrypt והารגומנט יהיה שם הקובץ. התוצאה תהיה קובץ מוצפן בשם זה רק עם הסיימת .decrypt

על האלגוריתם של ההצפנה כבר עברנו בפרק על הסטרימרים. הקוד של ההצפנה נראה כך:

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

const algorithm = 'aes-256-ctr';
const password = 'Password used to generate key';
const key = crypto.scryptSync(password, 'SomeSalt', 32);
const iv = 'myIVstringisnice1'.toString('hex').slice(0, 16);

const readStream = fs.createReadStream('./test.txt');
const writeStream = fs.createWriteStream('./encrypted.txt');
const encryptStream = crypto.createCipheriv(algorithm, key, iv);

readStream.pipe(encryptStream).pipe(writeStream);
```

אבל אנו נאזרז את זה באופן שונה לchlוטין, בתוך קלאס שעומד כמודול עצמאי למחרי. למדנו את זה בפרקם הקודמים והקוד הבא אמור להיות מובן לחchlוטין. בקובץ `src/module.js` ניצור את הקלאס שמבצע את ההצפנה. זהו אותו קוד בדיק, אבל ארוז בצורה אחרת:

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

class EncryptionCLI {

  constructor() {
    this.algorithm = 'aes-256-ctr';
    this.password = 'Password used to generate key';
    this.key = crypto.scryptSync(this.password, 'SomeSalt', 32);
    this.iv = 'myIVstringisnice1'.toString('hex').slice(0, 16);
  }

  encrypt(sourceFileName) {
    const destinationFileName = sourceFileName + '.encrypted';
    const readStream = fs.createReadStream(sourceFileName);
    const writeStream = fs.createWriteStream(destinationFileName);
    const encryptStream = crypto.createCipheriv(this.algorithm,
this.key, this.iv);
    readStream.pipe(encryptStream).pipe(writeStream);
    writeStream.on('finish', this.onEnd);
  }

  onEnd() {
    console.log('Finished!')
  }
}

module.exports = new EncryptionCLI();
```

יש לנו כאן קלאס. ב-`constructor` אנו מגדירים את המשתנים שנctrar בהמשך: `key`, `iv` ו-`algorithm`. במתודה `encrypt` יש ארגומנט אחד – שם הקובץ. אנו לוקחים אותו ומשתמשים בו בקריאה ובכתיבת – בסטרים. בסוף הכתיבה נזין לאירוע "סוף" ונdfs `Finished`.
נבנה את הקלאס באמצעות `new` ונעביר אותו לכל מי שעושה שימוש באתרים `require` באתרים `module.exports`

על מנת לבדוק שככל מה שכתבנו עובד, ניגש ל-`js.app`, נוצר את המודול זהה באמצעות `require` וונשתחם בו. באוטה תקיה של `js.app` ניצור קובץ `test.txt` ונכנס בו כמה מילים. ב-`js.app` נדרש את המודול ונשתחם בו על קובץ `test.txt`:

```
const EncryptionCLI = require('./src/module');

EncryptionCLI.encrypt('./test.txt');
```

אם הכל יעבוד, כאשר אני אכתוב `node ./app`. `node` ייווצר לי קובץ בשם `test.encrypted` ואני אראה `finished` בטרמינל. הקובץ `test.encrypted` יוכל מיד מוצפן.

אבל אני לא רוצה להפעיל את המודול בקוד, ב-`js.app`. אני רוצה להפעיל אותו ב-CLI! קלומר שאני אוכל להקליד `encrypt` ואת שם הקובץ בטרמינל (איזה שם קובץ שארצها!) והוא יבצע לי את ההצפנה. איך אני עושה את זה?

ראשית אני אצור תיקיה בשם `node`. נהוג, במקרים רבים של `Node.js`, לשים את כל הקבצים הקשורים ל-CLI בתיקייה זו. אני אצור שם קובץ. הוא יוכל להיות בכל שם שהוא אבל אני אבחר `encrypt.js`.

כדי לסמן ל-`Node.js` שמדובר בקובץ שניית להרצת מה-CLI אני חייב להקליד בתחילת הקובץ את:

```
#!/usr/bin/env node
```

זה נקרא **shebang** וזה קשור למערכת הפעלה. אני לא מסביר את השורה זו בספר זה, אך אני מזמין אתכם לחפש אותה וללמוד עליה. אם אתם משתמשים במערכת הפעלה שאינה חלונית, אתם יכולים לנסות כבר עכשיו להקליד את שם הקובץ ללא `node` ותראו ש-`Node.js` תריץ אותו. כמובן כתבתם `./bin/encrypt.js`.

עכשו ליעודה. אנו יכולים לצרוך את המודול שלנו כמו ב-`js.app`, בדוק כך:

```
const EncryptionCLI = require('../src/module');
```

כדי לשים לב שהשתמשתי במיקום ב.. – כדי לעלות לתיקית האב ומשם לגשת לתיקית `src`.

למה? כיוון שמבנה הפרויקט שלי נראה כך:

```
└──bin  
└──src  
└──app.js
```

ואם אני נמצא בקובץ בתיקייה `bin`, ואני רוצה לבצע `require` לקובץ בתיקייה `src` שבו נמצא המודול שלי – אני נדרש לעלות לתיקית האב וזרת לתיקית `src`. אם זה מסובך לכם, כדאי לכם להשתמש ב-VSCode שמאוד מקל את הניווט כי יש לו `auto complete`.

אחרי שצרכתי את המודול שלי, אני צריך להשתמש בו. בשביל זה אני צריך להשתמש בשם הקובץ. כאמור, בתחילת הדוגמה הסברתי שאני רוצה לקבל את שם הקובץ מהארגון בתשורת הפקודה. איך? באמצעות המשתנה `global`

```
process.argv
```

ראשית, `process` הוא אובייקט גלובלי שימושי של `Node.js` שמחזיק בתוכו מידע רב על התהיליך שמריץ את `Node.js` וגם אפשר לבצע פעולות רבות באמצעותו. בפרק הזה אנו מתמקדים באחת התכונות שלו: `argv`.

`argv` הוא מערך שמכיל באיבר הראשון שלו את המיקום של `node`, ובאיבר השני שלו את המיקום של קובץ ה-'אואחסקריפט' שבו אנו פועלים. בשאר האיברים הוא מכיל את הארגומנטים בתשורת הפקודה. אנו רוצים רק ארגומנט אחד, וזהו האיבר השלישי במערך (כלומר זה שנמצא במקום [2] – אנו זוכרים שמערך מתחילה מ-[0] ב-'אואחסקריפט'). אותו נשלח למتدת `encrypt` בבדיקה כי שעשינו ב-`app.js`:

```
#!/usr/bin/env node
const EncryptionCLI = require('../src/module');

const sourceFileName = process.argv[2];

console.log(`Source is: ${sourceFileName}`);

EncryptionCLI.encrypt(sourceFileName);
```

נבדוק את זה על ידי הפעלה של encrypt.js באמצעות node בבדיקה כמו js.app, אבל הפעם עם ארגומנט – נלומר עם שם הקובץ שאנו רוצים להצפן:

```
node .\bin\encrypt.js .\test.txt
```

אם הכל עובד כנדרה, נראה שנוצר לנו קובץ מוצפן בשם test.txt.encrypted אבל זה עדין לא מספיק טוב, אני רוצה פקודה גלובלית. ממש כמו eslint. אני רוצה להקליד encrypt. איך אני עושה את זה? באמצעות json.package. עד כה התייחסנו אליו רק בתור מקום שיש בו מעט מידע על המודול שלנו ועל המודולים שהוא משתמש בהם וברסאות. אבל הוא גם אמרור להכיל מידע על ה-CLI שלנו, אם יש לנו. אנו נוסיף ל-JSON שיש ב-json.package את הטקסט:

```
"bin": {
  "encrypt": "./bin/encrypt.js"
}
```

bin הוא החלק ב-json.package שטיפל בהרצה מתוך שורת הפקודה. באובייקט הזה יהיה כל הפקודות שהמודול שלנו מכיל. כרגע זו רק פקודה אחת בשם encrypt. הערך שלה יהיה מחורזת טקסט שמכילה את הנתיב אל הקובץ של ה-obj. כיוון ש-json.package נמצא בתיקייה הראשית של הפרויקט, הנתיב הוא נקודה (כלומר המיקום הנוכחי), תקנית bin ואז שם הקובץ.

אבל כדי שנוכל לנכתב encrypt מכל מקום, אנו צריכים לגרום למודול להיות גלובלי. כיוון זהה לא מודול ב-NPM (עדין), אני לא יכול להתקין אותו באופן גלובלי עם -i וקח או עם akch. אבל יש דרך לגרום למודולים להפוך לגלובליים גם אם הם נמצאים רק במחשב שלי. איך? אני אקשר אותם

באמצעות קישור סימבולי אל ספריית התקיימות הגלובלית שיש במחשב שלי, בדיקת CPF ש `wkm` ג-ו עשוה, וזאת באמצעות הפקודה:

```
npm link
```

אני אקליד `link wkm` בתיקייה הראשית שלuproject וזו אלחץ על אנטר. מהנקודה הזו אני יכול להקליד `encrypt` ושם קובץ מכל תיקייה שיש בטרמינל שלי והקובץ יוצפן בבדיקה CPF שרציתי! כך יוצרים CLI עם `node`. כך המודולים הגלובליים `ox`-`int` וכל מודול אחר שרצה עם CLI עובדים. אם תיקחו את המודול שלכם ותפרסמו אותו ב-NPM, הוא יעבד עם `g-wkm` – אבל עד אז גם `link wkm` יספק.

תרגיל:

צרו מתודת decrypt והפכו אותה ל-CLI, כך שאוכל לכתוב `decrypt FILENAME` עם כל שם קובץ שuber הצפנה על ידי `encrypt` והוא יפתח את הקובץ.

פתרונות:

ראשית, מתודת `decrypt`-ה-CLI. אני אוסיף אותה למודול שלי ועתיק את הפונקציונליות שלו מהפרק על הסטרימים, שם היא מפורטת. הקלאס שלי ייראה如下:

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

class EncryptionCLI {

  constructor() {
    this.algorithm = 'aes-256-ctr';
    this.password = 'Password used to generate key';
    this.key = crypto.scryptSync(this.password, 'SomeSalt', 32);
    this.iv = 'myIVstringisnice1'.toString('hex').slice(0, 16);
  }

  encrypt(sourceFileName) {
    const destinationFileName = sourceFileName + '.encrypted';
    const readStream = fs.createReadStream(sourceFileName);
    const writeStream = fs.createWriteStream(destinationFileName);
    const encryptStream = crypto.createCipheriv(this.algorithm,
this.key, this.iv);
    readStream.pipe(encryptStream).pipe(writeStream);
    writeStream.on('finish', this.onEnd);
  }

  decrypt(sourceFileName) {
    const destinationFileName = sourceFileName + '.decrypted';
    const readStream = fs.createReadStream(sourceFileName);
    const decryptStream = crypto.createDecipheriv(this.algorithm,
this.key, this.iv);
    const writeStream = fs.createWriteStream(destinationFileName);
    readStream.pipe(decryptStream).pipe(writeStream);
    writeStream.on('finish', this.onEnd);
  }
}
```

```

onEnd() {
  console.log('Finished!')
}

module.exports = new EncryptionCLI();

```

הדבר היחידי שהשתנה פה הוא הוספת מתודת `decrypt`. אם עברתם על הסטרימים, זה מהهو שאמור להיות ברור: יצירת סטרים של קריאה שם קובץ שmagiu כארגומנט, יצירת סטרים של פענוח עם הנ托נים מה-`constructor`, יצירת סטרים של כתיבה שם קובץ שmagiu כארגומנט, הוספה `decrypt` והאזהה לאירוע "סיום".

עכשו ניצור בתיקייה `bin` קובץ בשם `decrypt.js`. הוא כמעט זהה לקובץ `encrypt.js` שיצרנו בפרק:

```

#!/usr/bin/env node
const EncryptionCLI = require('../src/module');

const sourceFileName = process.argv[2];

console.log(`Source is: ${sourceFileName}`);

EncryptionCLI.decrypt(sourceFileName);

```

השורה הראשונה היא הכרחית, היא ה-Shebang. בשורה השנייה אני צריך את המודול. שימוש לב לשתי הנקודות שיש בכתיבה – כיון שאנו בתיקיות `bin` ואני צריך להגיע לתיקיות האחות `src`, אני עולה לתיקיית האב עם `..` ואז יורד לתיקיית `src` ושם לקובץ `module.js` שהוא קובץ הג'אוועסקראופט שבו נמצא המודול שלנו.

השלב החשוב ביותר הוא לחת את הארגומנט באמצעות האיבר השלישי של `process.argv`.
הערך שיש ב-`process.argv[2]` הוא מערך של כל הארגומנטים. האיבר הראשון הוא `node`, האיבר השני הוא שם הקובץ הרץ והאיבר השלישי הוא הארגומנט הראשון.
במהשך – הדפסה מהירה של מה שקיבלתי בקונסולה לבקשתו ואז קריאה למתחודת `decrypt` של המודול שלי עם מה שקיבلتוי מהארגומנט.

החלק הנוסף הוא הוספה bin-7 decrypt.json שיש ב-package:

```
{
  "name": "EncryptDecrypt",
  "version": "1.0.0",
  "description": "Decryption \\\ Encryption CLI",
  "main": "app.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC",
  "bin": {
    "encrypt": "./bin/encrypt.js",
    "decrypt": "./bin/decrypt.js"
  }
}
```

החלק הסופי והאחרון הוא הקלה

`npm link`

בתיקיה הראשית של הפרויקט. הקלה זו בעצם מאפשרת לモודולים גלובליים וקשריהם למערכת הפעלה. זהה. אם אני אכתוב `decrypt` ובארגון השני אני אכתוב שם של קובץ שהצפנתי עם אותו כל שכתבתי, אני אוכל לראות את הקובץ המקורי:

`decrypt .\test.txt.encrypted`

תחת השם `test.txt.encrypted.decrypted`

חשוב לציין שהשתמשנו במודול `crypto` לשם הדוגמה בלבד. בחיים האמיתיים מומלץ מאוד להשתמש במודולים קיימים ל-`crypto`. בנוסף על כן, אני מפנה את תשומת ליבכם אל `scryptSync` ואל העבודה שמדובר בfonkcizia סינכרונית שהיא בעייתית בכל מקום אחר שהוא לא CLI.

פרק 14

סוקטים - SOCKETS



סוקטים – Sockets

עד כה למדנו תקשורת עם פרוטוקול HTTP, הכולר שירות אינטרנט, זהה כמעט לכל ניסיון תקשורת ומחייב תשובה. כל מי שגולש באינטרנט מכיר את הפורמט הזה. אני נכנס לאתר, האתר שולח לי את המידע זהה. הפרוטוקול של HTTP הוא Stateless, זאת אומרת שהקשר ביןלי לבין השירות מסתיים ברגע שהользоватל נשלח חזרה אל מבקש הבקשה.

אבל בראשת המודרנית לפעם אנו כן צריכים לשמר על קשר עם השירות. יש לנו כל מיני דרכים לעשות את זה. הראשונה היא "עוגייה". קובץ טקסט שנשמר בדף (צד לקוח) ונסלח בכל בקשה אל השירות וכן מהשירות. כך השירות יוכל לקשר בין המשתמש לפועלות קודמות שלו באתר. השנייה היא AJAX, בקשות שנשלחות באמצעות ג'אווהסקרייפט מהדף אל השירות (עם עוגייה או בלי) ובambilות לנו מידע. אם יש לנו אתר חדשה למשל, שרוצה להתעדכן כשהוא פתוח, סкриיפט ג'אווהסקרייפט יdag לשולח כל פרק זמן מסוים בקשה לשירות ברגע חדשנותו. השירות יחזיר לו את החדשנות והוא יחליף את התצוגה של החדשנות באתר. דרך נוספת יותר היא long polling, שבה בעצם השירות היה מעכבר אצל השלם בבקשת HTTP.

אבל בסופו של דבר, ב-AJAX, שזו השיטה הפופולרית ביותר, השירות סטטי – אנו תמיד מאזינים לתקשרות. לעולם לא שולחים. רק הדף או שירות אחר מסוגל לשלווח בקשות לתקשרות, באמצעות טעינה של הדף (הקלדה של שם המתחם או פשוט לחיצה על F5) או באמצעות שילחת בקשה AJAX. יש להז יתرون, כי קל מאד לנצל כך כמות גדולה של משתמשים. אך יש מערכות שפטרון ה-AJAX פשוט לא יעבד עבורי. חשבו למשל על מערכות רפואיות או כאלה שחשוב שתהייה בהן תקשורת בזמן אמיתי. במערכות כאלה גם שיגור בקשה AJAX מדי שנייה עלול לא להיות מספיק.

אבל יש דרכים לבצע תקשורת בראשת שאינה פרוטוקול HTTP. פרוטוקול TCP יושב מעל פרוטוקול אחר, שנקרא TCP – ראשי תיבות של Transmission Control Protocol ואנחנו יכולים להשתמש בו לתקשרות דו-כיוונית. הכולר ברגע שהתקשרות נפתחת, נוצר קשר בין השירות לקוחות והקשר הוא סימטרי – הדף יכול לשולח מידע לקוחות (וכמוון להפוך). הדרך זו נקראת סוקט (באנגלית Socket).

למה אנחנו צריכים כזו תקשורת? חשבו למשל על צ'אט, כמו פייסבוק מסנגר. אני יכול למשך צ'אט זהה עם בקשות AJAX. ככלומר הלקוח פותח את הצ'אט ואז הדפדפן משגר בקשות לבדוק אם יש הודעות חדשות מדי כמה שניות. זה אפשרי, אבל זה מאוד לא יעיל. אם יש לנו לפחות משתמשים, הם יישגרו הררי בקשות לשרת, שרובן לא נחוצות. יש אפליקציות ניטור ובקרה שצרכו תקשורת מיידית. לא בטוח ש-AJAX יסייע, כי יש דילוי, למשל, בתוכנות מסחר בבורסה ושאר אפליקציות לצרכים תקשורת דו-כיוונית.

לא תמיד סוקט עדיף. יש כמה יתרונות ל-AJAX – קל להשתמש בו, אין מוגבלה תיאורטית לכמות המשתמשים ובעולם השירותי הענן מאוד קל לפורש עוד שרת מאשר להשתמש בסוקט, שזו התקשורת יקרה יחסית. סוקט הוא כלי שחייב להכיר וללמוד, אבל הוא לא תמיד הפתרון האפשרי הטוב ביותר.

את סוקט אנו ממשים באמצעות מודול `net`, שהוא אנו מכירים. זה מודול שממנו המודול `http` יורש. במודול `http`, להזכירם, השתמשנו לייצר שרת אינטרנט בפרק על יצירת שרת אינטרנט. המתודות של `http`, כמו `http.createServer`, והאירועים מגיעים מモודול `net`. הם כבר אמרו לנו להיות מוכרים להם. הכל מאוד דומה לשרת `http`, למעט הבדל אחדמשמעותי: שרת `http` יוצר אובייקט שנקרא סוקט. האובייקט הזה הוא סטרים (על סטרים למדנו בפרק קודם) שאפשר לכתוב אליו ולקראם ממנו – נזכר סטרים של כתיבה ושל קריאה, וזה שונה משמעותית משרת `http` שעבד עם סטרים שאליו כותבים בלבד. פה יש לי עroz תקשורת דו-כיווני שאנו יכול לכתוב אליו או לקרוא ממנו.

תקשרות TCP לא עובדת עם דף-דף אלא עם תוכנות אחרות. יש מגוון עצום של תוכנות שאפשר לעבוד מולם. אני אדגים באמצעות תוכנה שנקראת `telnet`, שמתחברת לsockטים. אם אתם משתמשים בلينוקס, התוכנה זו כבר קיימת אצלכם כברירת מחדל. אם אתם משתמשים במק, אפשר בקלות להתקין אותה עם `brew` באמצעות הקלדת הפקודה `brew install telnet`.

אם אתם עובדים עם חלונות, פתחו את הטרמינל שלכם (אפשר דרך פקודה ה-cmd או באמצעות הטרמינל ב-Visual Studio code) – על שתי הדרכים למדנו בפרק הראשון בספר) והקלידו:

```
pkgmgr /iu:"TelnetClient"
```

לחברת מיקרוסופט, שעומדת מאחורי חלונות, יש הסברים מפורטים באתר שלהם על התקנת תוכנת Telnet. אם השורה הזו אינהעובדת עבורכם, תצטרכו לחפש בגוגל כיצד מתקנים Telnet על מערכת הפעלה שלכם. מדובר בתוכנה מאוד נפוצה שלא מעט מתכנתים צריכים על המחשב שלהם – וזה הסיבה שקל להתקין אותה. אל תוותו על השלב הזה והתקינו את התוכנה זו.

כדי לדעת אם התוכנה הותקנה בהצלחה, הקלידו בטרמינל שלכם:

```
telnet -help
```

עתה הקישו על אנטר. אם ההתקנה הייתה תקינה, תראו הסבר על פועלות התוכנה ותוכלו להמשיך. התוכנה מאפשרת לנו לפתח חיבורים שונים לשירותים שעובדים עם סוקטים. הבה ניצור שירות זהה.

כאמור, אנו משתמשים במודול של `net` בדיק כפי שהשתמשנו במודול זהה כדי ליצור שרת HTTP. נכניס את השירות שלנו למודול שלנו. יצירת אפליקציית `Node.js` עם `package.json` עם ניצור לה תיקיות `src` ובתוכו תיקיות `src` ניצור תיקייה `snkraat` `modules`. בתיקייה זו ניצור קובץ שנקרא `SocketServer.js` ונכניס לתוכו את הקוד הבא:

```
const netServer = require('net').Server;

class SocketServer extends netServer {
  constructor() {
    super();
    this.listen('6969');
    this.on('connection', this.connectionHandler);
  }
  connectionHandler(socket) {
    console.log(`Someone connected! ${socket.remoteAddress}`);
    this.socket = socket;
    this.socket.on('data', this.dataHandler);
    this.socket.write(`Welcome to my server`);
  }
  dataHandler(typedData) {
    console.log(`Typed This ${typedData}`);
  }
}

module.exports = new SocketServer();
```

הקוד הזה כבר אמרו להיות מוכן לכם. ראשית אני יוצר קלאס שנקרא `SocketServer` שיירוש מ-`net.Server`. ב-`constructor` שלו אני לא שוכח להכניס `super(this)` כפי שנדרש ממני תמיד כשהאני יורש. אני מאזין לפורט 6969 (בדיק כמו במודול `http`) ומאזין לאירוע שנקרא `connection`. האירוע הזה מופעל כאשר מתחבר לשרת שלנו, בדיק כמו האירוע המקביל ב-`http`.

האירוע מנוהל עם `connectionHandler`. פה יש הבדלמשמעותי מ-`http`. בעוד המודול של `http` עובד עם `response` (התגובה של השירות) ו-`request` (הבקשה שהגיעה לשרת), כאן יש לי רק `socket`. הsocket הזה הוא סטרים דו-כיווני של כתיבה וקריאה. אני יכול לקרוא ממנו ויכול לכתוב אליו ולעבוד עם אירועים כמו כל סטרים דו-כיווני שלMANDNO עליון בפרטים הקודמים.

מה שאני עושה זה לנכון לسطריםזה "ברוך הבא לשרת שלי" ולהזין לאירוע `data` שמופעל בכל פעם שmagiushו איזשהו מידע המשמש. וזהו הדבר הייחודי שיש לנו בסוקטים – היכולת לעשות דברים כשהמשתמש מקליד. כשההמשתמש מעביר משוה, השרת מפעיל את מתודת `dataHandler` – במקרה הזה היא רק מדפסה לקונסולה.

אני מבצע now לקלט זהה ומחזיר אותו ב-`module.exports`. זהו, יצרתי את המודול הזה. בתיקיה הראשית שלuproject שלו, אני אצור קובץ של `js.app` ובו אפעיל את המודול שיצרתי. איך? באמצעות `require`:

```
const socketServer = require('./src/modules/SocketServer');
```

מהרגע שאני אפעיל את `js.app` באמצעות `node ./app` שאקליד בטרמינל, השרת פועל ואני יכול להתחבר אליו דרך הטלנט. אפתח חלון אחר של טרמינל ואקליד שם:

```
telnet 127.0.0.1 6969
```

הפקודה זו מורה לטלנט להתחבר ל-IP המקומי 127.0.0.1 ולפורט 6969. אם עשייתי הכל כבירה, אני אראה את הכתובת `Welcome to my server`. כל מה שאקליד בתוכנה – אני אראה מיד בקונסולה של השרת, בטרמינל השני.

הכתובת המקומית שלנו היא תמיד 127.0.0.1 – זהה כתובות שמורה המיועדת למחשב המקומי. המחשב המקומי נקרא גם `localhost` ואני יכולם להקליד גם:

```
telnet localhost 6969
```

וזה יעבוד.

ב-Visual Studio Code, שעליו כאמור למדנו בפרק הראשון, אפשר לעבוד עם טרמינל מפוצל ואז קל מאד לראות גם את צד השרת שיצרנו והפעלנו וגם את צד הלוקה.

```
PS C:\Users\barzik\node_projects> node .\app.js
Someone connected! ::ffff:127.0.0.1
Typed This

Typed This d
Typed This d
Typed This d
Typed This f
Typed This g
Typed This h
[]

Welcome to my server
dddfgh
```

מקובל ליצור שרת של סוקט כדי ללמד ולהבין איך סוקט עובד. בסופו של דבר, שרת סוקט נשמע מואוד מפוץ אבל אם אנו מכירים שרת `http` לעומק, הוא לא שונה מהותית, כי הכל מבוסס על `.net.Server`. הדבר החשוב הנוסף הוא שסוקט הוא בסופו של דבר סטרים נוע.

אבל בעולם האמיתי לא תצרכו (בדרך כלל) ליצור שרת שמתחברים אליו עם טלנט אלא לעשות דברים אחרים – כמו חלון צ'אט למשל. הכוח המרכזי של `Node.js` הוא המודולים רבים שמتبסיסים על המודולים הטבעיים שלו, ומודול זהה הוא `socket.io` שמאפשר לנו בקלות רבה ליצור סוקטים שיכולים לעבוד בצורה דפדן.

Socket.io הוא מודול מרכזי שנמצא ב-NPM. יש לו גם אתר נאה שאפשר ללמוד ממנו וככל דוגמאות בכתובות – <https://socket.io/> – לרוב המודולים הרציניים ב-NPM יש אתר משליהם עם הסברים ודוגמאות ולא רק ייצוג-ב-NPM. זה מה שטוב-ב-`Node.js` – לא צריך לקרוא מספר כדוגמת הספר הזה כדי להבין איך להשתמש במודולים אלו – אפשר לצלול אל הדוגמאות והסבירם שמספקים בדרך כלל ביד נדיבת ורחבת. יש כל כך הרבה מודולים מרכזים ב-`Node.js` בספר אחד או אפילו סדרת ספרים לא יכולם לכטוט אותם. במקרה, למשל, בזמן כתיבת שורות אלו, יש הסבר מקיים איך ליצור אפליקציה צ'אטים מקיפה. אני ממליץ לכם, לאחר קראית הספר, לנשות לבנות אותה.

יש מתכנתים שיכולים להעביר קריירה שלמה ב-node.js לאלגוריתם של סוקט, אבל זה חלק חשוב מ-node.js וגם סוקטים עומדים מאחוריו כמו אטרים ותוכנות חשובות וכך לתרגל ולהבין שהה לא קסם – אלה סוקטים.

תרגיל:

צריך לשדר באמצעות סוקט את התאריך של השרת בכל שנייה. חישוב התאריך נעשה באמצעות:

```
const current_time = new Date().getTime().toString();
```

הקיים שרת שתומך בסוקטים, יצרו setInterval שמשדרת בכל שנייה את ה-current_time.

פתרון:

```
const netServer = require('net').Server;

class SocketServer extends netServer {
  constructor() {
    super();
    this.listen('6969');
    this.on('connection', this.connectionHandler);
  }
  connectionHandler(socket) {
    console.log(`Someone connected! ${socket.remoteAddress}`);
    this.socket = socket;
    this.socket.write(`Welcome to my server`);
    this.repeater();
  }
  repeater() {
    setInterval(() => { //running it every second
      const current_time = new Date().getTime().toString();
      //calculating the time
      this.socket.write(current_time);
    }, 1000);
  }
}

module.exports = new SocketServer();
```

אנו יוצרים קלאס שירוש `m-Server.net`. הקלאס זהה הוא שרת לכל דבר, בדומה לשרת `http`. ב-`constructor` שלו אנו מażינים לאירוע מסווג חיבור. האירוע הזה מופעל בכל פעם שימושו `this.connectionHandler` מתחבר לשרת (למשל עם טלנט). בכל פעם שימושו מתחבר, `this.connectionHandler` מופעלת.

בתוך `this.connectionHandler` אנו מבצעים קריאה לפונקציה שנקראת `repeater`, ככלומר מפה זה ג'אווהסקריפט כמעט טהור. אנו יוצרים `setInterval` שמופעלת מדי שנייה. דנו בפונקציה זו בפרק על טיימרים והוא נלמדת גם בספר "לימוד ג'אווהסקריפט בעברית". `setInterval` מקבלת שני ארגומנטים, אחד מהם הוא פונקציה שמבצעת כתיבה ל-`socket` ומדפיסה בכל שנייה את הזמן לפי הפונקציה שניתנה בתחילת התרגיל. שמו לב שאנו משתמשים בו `toString` כדי להמיר את המספר למחזור טקסט שרק היא יכולה לעבור בסוקט.

אנו נוצרות את המודול הזה באמצעות `require` ב-`app.js`:

```
const socketServer = require('./src/SocketServer')
```

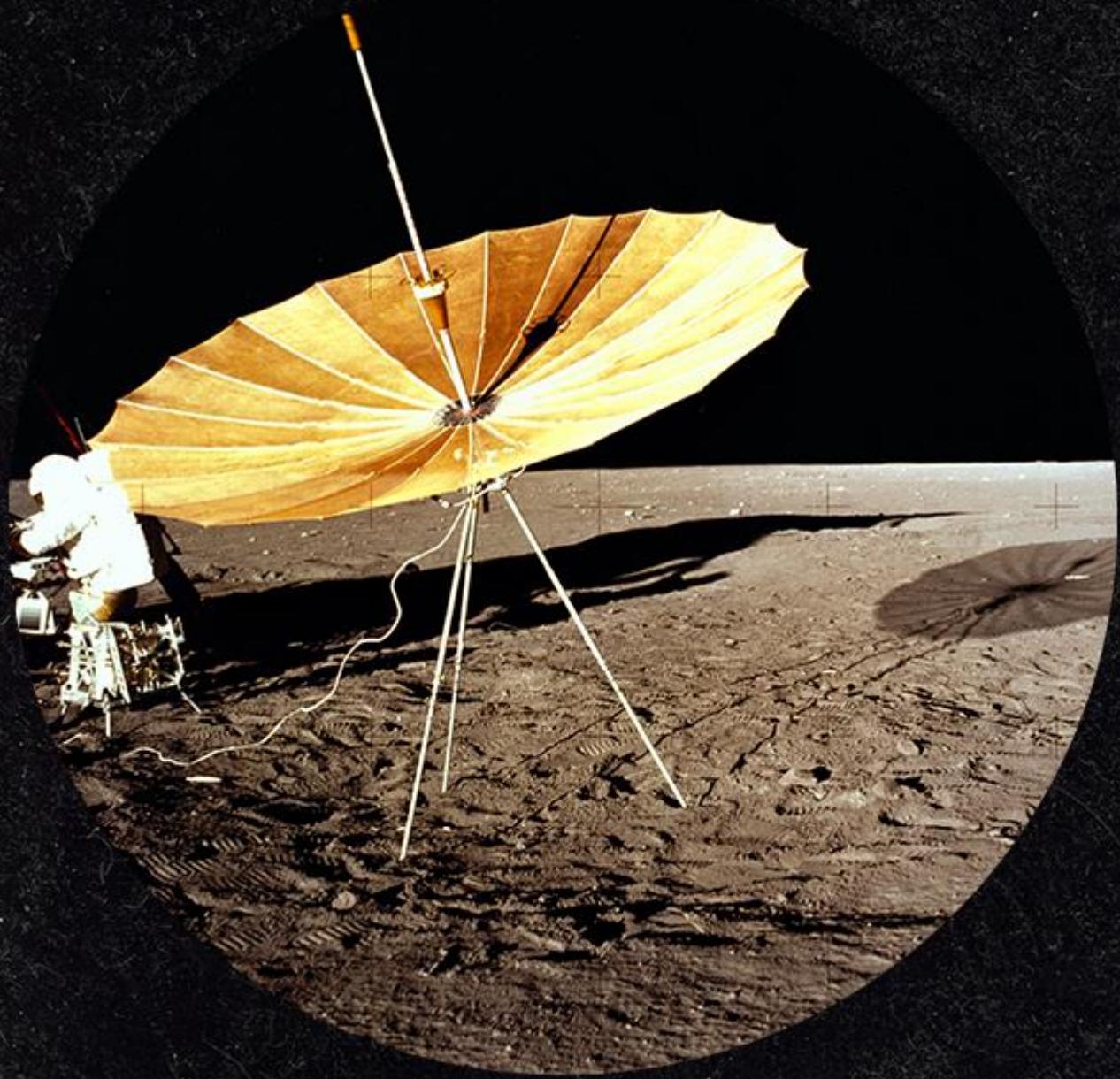
ונפעיל את השרת באמצעות `node ./app.js`.
כדי לבדוק את השרת, נפתח טרמינל נוסף ונכתב בו:

```
telnet 127.0.0.1 6969
```

נראה שמיד לאחר הודעת החיבור, השרת מזרים דרך הסוקט בכל שנייה את הזמן כפי שהוגדר לו.

פרק 15

קוריאת מושבים PATH באמצענות נזודול



קריאה משאים באמצעות מודול path

עד כה, כשקראנו לקבצים באמצעות סטרימים או קריאות אחרות, יצאנו מנקודת הנחה שהקבצים נמצאים באותה תיקייה. אבל זה לא תמיד נכון. פעמים רבות אנו צריכים לקרוא לקבצים שנמצאים בתיקיות שונות לחלוטין, למשל במצב הנפוץ מאד שבו יש לנו שרת שמציגקובצי `html`. בואו ניצור שרת פשוט שקורא לקובץ `html`.

ניצור תיקייה בשם `simple-client` ובתוכה ניצור תיקיית `src`. בתיקייה זו ניצור תיקייה שנקרויה `modules`, שם יהיו כל המודולים שלנו. תיקייה נוספת תחת תיקיית `src` תיקרא תיקיית `static` ושם ניצור את דף ה-`HTML` של האפליקציה שלנו. נתחל את הפרויקט שלנו בהקלה `init npm` ושם נעה על השאלות. יש לנו פרויקט! עכשו נאכלס אותו בקוד.

בתוך תיקיית `static` ניצור קובץ בשם `:index.html`:

```
<!doctype html>
<html>

<head>
  <title>My Clock</title>
  <meta name="description" content="WebSocket clock example">
  <meta name="viewport" content="width=device-width, initial-
scale=1">
</head>

<body>
  <h1>Hello world!</h1>
</body>

</html>
```

אני רוצה להגיע במצב שבו אני מקליד `localhost` ואני רואה את הקובץ הזה. איך עושים את זה? מרים שרת פשוט, בתוך מודול. כבר למדנו איך עובדים עם שרתים ואיך עובדים עם מודולים. הנה ניצור את המודול שלנו – מודול שיוצר שרת `http` רגיל. נלך לתיקיית `modules` וניצור שם קובץ שמו הוא כשם הקלאס ומתחילה באות גדולה: **WebServer**.

```

const httpServer = require('http').Server;
const fs = require('fs');

class WebServer extends httpServer {
  constructor() {
    super();
    this.listen(3000);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {

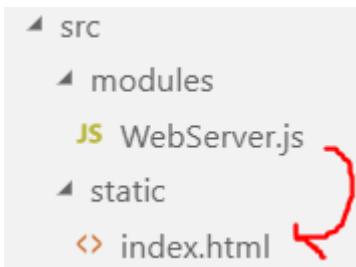
    const src = fs.createReadStream('../static/index.html'); // 
    ERROR!!!
    src.pipe(response);

  }
}

exports.module = new WebServer();

```

זהו קלאס שדומה לchlוטין לקלאסים של מודולים שלמדו לנו עליהם בעבר. שווה להתעכ卜 על ה-`fs.createReadStream`. אני צריך להקליד את הנתיב שבו הוא נמצא. המודול נמצא בתיקיית `modules`. הקובץ `index.html` נמצא בתיקיית `static`.

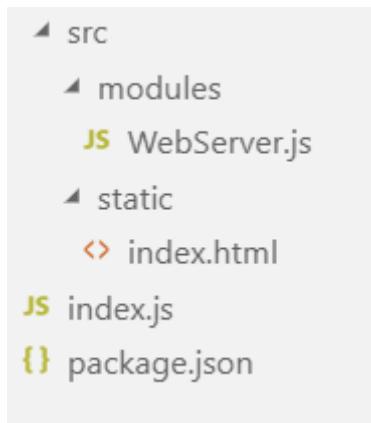


איך אני מגיע אליו? פשוט שבסוטרים! מהמקום הנוכחי שלי /. אני עולה באמצעות .. תיקיית `src` ומשם יורד אל `static/index.html`. הנתיב שאני צריך להכניס הוא: `./../static/index.html`

השלב הבא והאחרון הוא ליצור `index.js` בתיקייה הראשית של הפרויקט ושם לקרוא למודול שלי:

```
const server = require('./src/modules/WebServer.js');
```

מבנה האפליקציה אמור להיראות כך:



נ裏ץ את האפליקציה כולה באמצעות `node index`. ניכנס ל-`localhost:3000` ונראה שהטרמינל מתפוצץ מהודעת שגיאה. הودעת השגיאה טעונה שהשורה הזו:

```
const src = fs.createReadStream('../static/index.html');
```

לא עובדת כי המשאב לא נתען. איך זה יכול להיות?

התשובה היא `sh.js` מחשבת את הנתיבים מנקודת ההרצה. כלומר, אם אני מ裏ץ את הפרויקט מהתיקייה הראשית שלו, `Node.js` תחשב את הנתיב בצורה שונה לחולוטין.

איך פותרים את זה? באמצעות מודול `path`, שהוא מודול טبעי לגמרי `Node.js`, והקבוע הגלובלי `__dirname`.

הקבוע `__dirname` הוא קבוע שתמיד מחזיר את המיקום הנוכחי של הקובץ. אם אני מ裏ץ אותו בכל קובץ שהוא, הוא תמיד יביא לי את הנתיב המלא של הקובץ. אם הקובץ נמצא בconscious C בתיקיית `barzik`, הוא יחזיר את המיקום הזה. זהו דבר שימושי מאוד כאשר אני רוצה להגיע למשאים בתת-תיקיות. במקרה זה אני רוצה להגיע למשאים בתיקיות אחרות וудין רוצה להשתמש בנתיב יחסית כמו `..` למשל.

בדיקת CAN יש `path.join`, שמקבלת שני ארגומנטים ומרכיבת משנייהם נתיב אחד, בעוד הארגומנט השני יכול להיות נתיב ייחודי. כלומר אם אני רוצה לקרוא לקבצים מתיקיות אחרות, אני צריך להשתמש ב:

```
path.join(__dirname, '../static/index.html')
```

וכמובן לא לשכוח לעשות `require('path')` לモודול path:

```
const httpServer = require('http').Server;
const fs = require('fs');
const path = require('path');

class WebServer extends httpServer {
  constructor() {
    super();
    this.listen(3000);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {
    const src = fs.createReadStream(path.join(__dirname,
      '../static/index.html'));
    src.pipe(response);
  }
}

exports.module = new WebServer();
```

לモודול path יש עוד כמה מוגדרות חשובות שכדי להציג בדוקומנטציה עברון – אבל השימוש שלו עם `__dirname`, הקבוע הגלובלי שיש ב-node.js, הוא שילוב שתראו המון בקטעי קוד וגם כתבו בעצמכם. נוסף על כן, יתרון גדול נוסף לモודול path הוא סיוע משמעותית בהבדלים שבין מערכות הפעלה שונות. מערכת הפעלה חלונות ומערכת הפעלה לינוקס ומק מנהלות קבצים ותיקיות באופן שונה, וכך למשל יכולה להיות בעיה אם אתם מפתחים על חלונות ומעלים את סקורייפט ה-node.js שלכם לשרת מבוסס לינוקס. שימוש ב-path יכול לעזור את הבעיה.

פרק 16

PACKAGE.JSON SCRIPTS



package.json scripts

עד כה, כשרצינו להפעיל את התוכנה שלנו, נאלצנו להקליד באופן ידני `node ./app.js` או `node index.js`. כמו בדוגמה שבפרק הקודם. אבל בעולם האמיתי אנו לא עושים את זה בדרך כלל אלא משתמשים ב-NPM על מנת להריץ את האפליקציה שלנו, את השרת שלנו או את הקוד שלנו. אנו עושים את זה כדי לבצע סטנדרטיזציה. מתכנת אחר שישתמש בקוד שלנו לא אמרו להזכיר אותו. במקרה לשבור את הראש אם להקליד `node ./app.js` או כל entry point אחר, הוא יוכל להקליד פקודה אחרת ונגדייר בדוק מה היא עשויה.

הפקודות האיחודות נמצאות מתחת ל-`scripts` בקובץ `package.json`. בדוגמה הקודמת, הפעלונו את השרת שלנו באמצעות הקלדה של `node ./index.js`. אם נכנס את הקוד שלנו תחת `start` ב køפּן זהה:

```
{
  "name": "http-example-server",
  "version": "1.0.0",
  "description": "Example of HTTP Server",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node ./index.js"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC"
}
```

ונקליד בטרמינל `npm start` – הפקודה תריץ את כל מה ששםנו ב-`start`. במקרה זהה:

```
node ./index.js
```

למה זה שימושי? כי יש עוד פקודות והוקים (Hooks) שאפשר לעשות איתם המונן דברים שימושיים ב-`npm scripts`. למשל הhook `prestart` שרצה תמיד לפני שהוא מרים `npm start`.
כאן, למשל, אני מבקש ממנו להציג הودעה:

```
{  
  "name": "http-example-server",  
  "version": "1.0.0",  
  "description": "Example of HTTP Server",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1",  
    "prestart": "echo \"READY TO START\"",  
    "start": "node ./index.js"  
  },  
  "author": "Ran Bar-Zik",  
  "license": "ISC"  
}
```

כשאני אקליד `start` וקח, לפני שהתוכנה תרוץ, היא תכתוב הודעה של `READY TO START` לטרמינל. מובן שלא משתמשים ב-`restart` כדי להציג הודעות. אם יש לי שרת, אני יכול לבצע בדיקות תקינות על הסביבה, כיווץ מסמכים ומשאבים וניקוי כללי, ובעצם להריץ כל סקריפט שבא לי להריץ עם פקודה מסוימת.

סקריפטים עם שמות

אנו לא חייבים להציגם לשמות של NPM אלא יכולים להשתמש בכל שם שהוא. כך למשל, אני יכול להחליט שיש לי סקריפט שנקרא ahla וכשאקרה לו הוא ידפיס לי BAHLA:

```
{
  "name": "http-example-server",
  "version": "1.0.0",
  "description": "Example of HTTP Server",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "prestart": "echo \"READY TO START\"",
    "start": "node ./index.js",
    "ahla": "echo BAHLA"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC"
}
```

איך אקרא לו? הפעם אני אצטרך להקליד חנוך לפני הפקודה. במקרה של ahla:

```
npm run ahla
```

שימוש לב שחייבים להשתמש ב-חנוך כאשר מרצים כל פקודה שהיא לא חלק מסט הפקודות המובנות ב-NPM. יש מעט פקודות מובנות שלא דורשות חנוך לפני הקלהת הפקודה, והבולטות שבהן הן:

הסבר על הפקודה	פקודה מובנית ב-NPM
מתקינה את כל dependencies ב-package.json	install \ i
מריצה את פקודה test המפורטת ב-package.json	test \ t
מריצה את פקודה start המפורטת ב-package.json	start

משתני סביבה

אחד הדברים שימושי כל לעשנות עם חנו וקח הוא לעבוד עם משתני סביבה ולהשתמש בהם בקוד. משתני סביבה הם ניפוי שהם נשיינים – משתנים שמשמעותם מסביבת ההרצה ולא מהקוד. אנו משתמשים במשתני סביבה בתנאים רגילים שאנו לא רוצחים להכניס לקוד או בתנאים שעלולים להשנות מסביבה לשונית, למשל פורט. בסביבת הפיתוח, פעמים רבות אני רוצה להרים את השרת שלי בפורט 80 שהוא הפורט של ברירת המחדל. אך בסביבת הprdוקשן, הסביבה האמיתית, אני רוצה להרים את השרת שלי בפורט ששרת הדיפלומנט מגדר לו. איך עושים את זה?

זהו הקלאס של שרת http שעבדנו עליו בפרק הקודם על path. בכל הדוגמאות של שרתי http כתבנו את הפורט ממש כמספר. הפעם אני אקח אותו כמשתנה סביבה:

```
const httpServer = require('http').Server;
const fs = require('fs');
const path = require('path');

class WebServer extends httpServer {
  constructor() {
    super();
    const port = process.env.PORT;
    this.listen(port);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {
    const src = fs.createReadStream(path.join(__dirname,
      './static/index.html'));
    src.pipe(response);
  }
}

exports.module = new WebServer();
```

זו השורה החשובה – כאן אני לוקח את הפורט בעצם ממשתנה הסביבה PORT:

```
const port = process.env.PORT;
```

איך אני קובע את משתנה הסביבה?

קביעת משתנה סביבה דרך הסкриיפט

אפשר לקבוע משתנה סביבה דרך הסкриיפט של package.json באמצעות פקודה set באופן פשוט במיוחד – הצבת

```
set PORT=3000
```

לפני הפקודה ושרשוור עם &&. הינה הדוגמה:

```
{
  "name": "http-example-server",
  "version": "1.0.0",
  "description": "Example of HTTP Server",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "set PORT=3000 && node ./index.js",
    "dev": "set PORT=80 && node ./index.js"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC"
}
```

במקרה שלנו – יש לנו שני סкриופטים: start ו-dev. כל אחד מהם רץ עם פורט אחר. אם אקליד npm run start ו-npm run dev, יוכל להיכנס לשרת ה-HTTP שלי רק מlocalhost:3000 ואם אקליד npm run dev אז יוכל להיכנס לשרת ה-HTTP שלי רק מlocalhost:80.

קביעת משתנה סביבה דרך הגדרות מערכת הפעלה

הדרך הקודמת تعمل אך בהרבה מקרים היא בעייתית – במיוחד במקרים מסוימים רגשיים. לא מעט פעמים אנו צריכים טוקנים – סוג של סיסמות המשמשות לחבר למסדי נתונים ולשירותים שונים. אנו לא יכולים לאחסן אותם בקובץ מילוי סיבוב – אבטחת מידע וגם עניינים אחרים – לעיתים הטוקנים משתנים. בדיק בשביל זה משתמשים פעמים רבות על מערכת הפעלה.

משתני סביבה הם לא ייחודיים ל-node.js ויש לא מעט תוכנות שמסתמכו עליהם. לכל מערכת הפעלה יש כמה וכמה דרכי להגדיר משתני סביבה.

חלונות – CMD

set PORT "3000"	קבעת המשתנה PORT כ-3000
echo %PORT%	בדיקה המשתנה PORT
SET	הצגת כל המשתנים

חלונות – הטרמינל של Visual Studio Code (ידעו גם כ-Powershell)

\$Env:PORT = "3000"	קבעת המשתנה PORT כ-3000
echo \$Env:PORT	בדיקה המשתנה PORT
gci env:*	הצגת כל המשתנים

לינוקס/מак

export PORT=3000	קבעת המשתנה PORT כ-3000
echo \$PORT	בדיקה המשתנה PORT
printenv	הצגת כל המשתנים

מודול cross-env, שלא אלמד עליו פה, מסייע למתכנתים ב-node.js לנהל משתני סביבה במגוון פלטפורמות. כדאי להכיר אותו ואפשר למצוא עליו מידע ב-ויקי:

<https://www.npmjs.com/package/cross-env>

קביעת משתנה סביבה דרך קובץ

דרך נוספת לניהול משתני סביבה היא באמצעות קובץ `.env`. (נקודה בתחילת שם הקובץ ולא סימנת קובץ). מדובר בקובץ שיותב בתיקייה הראשית של הפרויקט שלכם ומכיל משתני סביבה. הוא יכול להיראות כך למשל:

```
PORt=666
NODE_ENV=development
```

כדי לקרוא את הקובץ, יש להתקין את מודול `dotenv` ולקראות לו באופן הבא בסקריפט שלכם:

```
const dotenv = require('dotenv');
dotenv.config();

console.log(process.env.PORT); // 666
```

אך חשוב שחשוב מאוד להקפיד שלא לשמר את `.env`. נחלק מקובצי הפרויקט שלכם! בכל האפליקציות והאתרים משנים את ההתנהגות באמצעות משתני סביבה. משתנה הסביבה החשוב ביותר, זהה שימושים בו כמעט הרבה, הוא `NODE_ENV` שיכול להיות `development` או `production`. תלוי בסביבה – בסביבת הפיתוח, המחשב שלנו, הוא יהיה `development`. בסביבה שבה האפליקציה שלנו עובדת הוא יהיה `production`.

זה חשוב מאוד כיון שיש התנהוגיות שאנו ממש לא רוצים בסביבה חיה. יש לנו כל מיני סיבות – למשל אנו לא רוצים `console.log` בסביבה חיה, אנחנו לא רוצים שבסביבה חיה יהיו לנו קבצים לא דחוסים וכו'. אם אתם רואים `process.env.NODE_ENV` תדעו שמדובר במשתנה סביבה ותדעו גם שחשוב לקבוע אותו במחשב שלכם, אבל גם לוודא שבשרת שבו התוכנה יושבת הוא `production`.
יוגדר כ-`C-production`.

dev dependencies

בתחילת הפרק הסבירתי מה הם scripts ומוק. משתמשים בדרך כלל בסקריפטים האלה לביצוע פעילות כמו בדיקת תקינות עם eslint או עם כלים אחרים. אנו נבצע הדוגמה באמצעות eslint שעליו דיברנו בפרק על מודולים חיצוניים ו-אקס. המודול הזה משמש לבדיקת תקינות קוד. אנו נתקין אותו באמצעות npm.

אבל יש שהוא חשוב במיוחד לגבי eslint: הוא משמש לבדיקת תקינות קוד והוא רלוונטי רק למפתחים. אני רוצה אותו בסביבת הפיתוח ורוצה שהוא יותקן למפתחים כאשר הם יכתבו npm – ככלומר אני רוצה שהמודול הזה יהיה ב-package.json, אבל אני ממש לא רוצה שהמודול הזה יהיה בסביבה production, בסביבה חיה. איך אני מונע ממנו להיות מותקן בסביבה חיה?

בדוק בשביל זה קיים **dev-dependencies**. מדובר במודולים שאנו משתמש בהם אך ורק בסביבת פיתוח. כשהאני מתקין מודול שנדרש אך ורק לסביבת פיתוח, אני אתקין אותו באמצעות הקלדה של הפלאג –save-dev. ההקלדה הזו מכניסה את המודול שהותקן ל-package.json. הנה נציגים. באפליקציה השרת שלנו, שעליה עבדנו בתחילת הפרק, נפתח טרמינל ונקליד:

```
npm i eslint --save-dev
```

מדובר בהתקנה של eslint רק בסביבות פיתוח. אם נסתכל ב-package.json, נראה שיש לנו חלק חדש:`:devDependencies`

```
"devDependencies": {  
    "eslint": "^5.16.0"  
}
```

אם ה-NODE_ENV שלי הוא production, כשאבצע npm install, אז NPM לא תתקין את מה שיש ב-`devDependencies`. זה נכון גם זמן של התקנות אבל גם בעיות אבטחה.

תרגיל:

התקינו את eslint בפרויקט שהוסבר עליו בפרק הקודם. הקפידו שהוא מותקן בסביבה חיה.

צרו קובץ הגדרות של בדיקת תקינות באמצעות eslint-init. צרו משימה שתבצע בדיקה סטטistica של הקוד באמצעות lint run npm.

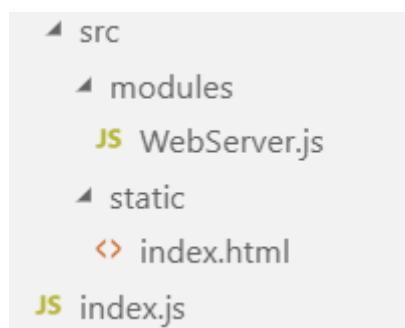
רמז: אפשר לבצע בדיקת קוד לאחר יצירה קובץ הגדרות באמצעות:

```
npx eslint **/*.js
```

אם אתם זקוקים לتذكرת על akch, יש לקרוא שוב את הפרק על התקנות מודולים ועל מודולים גלובליים.

פתרון:

מבנה התקינות של הפרויקט שלי נראה כך:



בתיקית המודולים נמצא המודול של השרת, שלו דיברנו בפרק הקודמים:

```

const httpServer = require('http').Server;
const fs = require('fs');
const path = require('path');

class WebServer extends httpServer {
  constructor() {
    super();
    const port = process.env.PORT;
    this.listen(port);
    this.on('request', this.requestHandler);
  }
  requestHandler(request, response) {
  }
}
  
```

```

    const src = fs.createReadStream(path.join(__dirname,
'./static/index.html'));
    src.pipe(response);
}
}

exports.module = new WebServer();

```

ב-`index.html` נמצא קובץ ה-`index.html` שנטען:

```

<!doctype html>
<html>

<head>
  <title>My Clock</title>
  <meta name="description" content="WebSocket clock example">
  <meta name="viewport" content="width=device-width, initial-
scale=1">
</head>

<body>
  <h1>Hello world!</h1>
</body>

</html>

```

ואילו ב-`index.js` נמצאים הקריאה לモול וטעינתו:

```
const server = require('./src/modules/WebServer');
```

אתקין את eslint בפרויקט שיש לו `package.json` – ואם קראתם את הפרק הקודם וביצעתם את התרגילים, אמור להיות לכם `package.json` בפרויקט. במידה שלא, `npm init` ייצור לכם `package.json` ואז תוכלו להתקין את `eslint`. אנו מתקינים את `eslint` באמצעות:

```
npm install eslint -save-dev
```

מודול `eslint` יותקן ב-`devDependencies`. כך בסביבה חיה הוא לא יותקן. סביבה חיה מוגדרת כסביבה שבה `NODE_ENV` הוא `production`.

אחרי ההתקנה, ניצור קובץ הגדרות באמצעות הקלדה של:

```
npx eslint -init
```

אחרי שנעננה על כל השאלות, נראה שבתיקייה הראשית של הפרויקט נוצר קובץ הגדרות של eslint. כדי לבדוק את העניין, נקליד בטרמינל:

```
npx eslint **/*.js
```

אם הכל עובד, נראה את כל השגיאות הסגנוניות שיש בקוד שלנו. כדי להכניס את הפקודה לסקריפטים, ניצור ב-file.json תחת scripts את המשימה lint עם השורה שמבצעת את eslint:

```
"lint": "npx eslint **/*.js"
```

קובץ file.json יראה כך:

```
{
  "name": "http-example-server",
  "version": "1.0.0",
  "description": "Example of HTTP Server",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node ./index.js",
    "lint": "npx eslint **/*.js"
  },
  "author": "Ran Bar-Zik",
  "license": "ISC",
  "devDependencies": {
    "eslint": "^5.16.0",
    "eslint-config-standard": "^12.0.0",
    "eslint-plugin-import": "^2.17.3",
    "eslint-plugin-node": "^9.1.0",
    "eslint-plugin-promise": "^4.1.1",
    "eslint-plugin-standard": "^4.0.0"
  }
}
```

עכשו, בכל פעם שנקליד npm run lint, בדיקת התקינות תרוץ.

פרק 17

אקספלור



אקספרס

אחד המודולים החשובים ביותר ב-Node.js הוא מודול אקספרס. מדובר במודול של שרת אינטרנט. זוכרים את מודול `http` הטבעי של Node.js? בפרקם הקודמים למדנו איך להשתמש בו באופן בסיסי מאד, ואפשר ללא ספק לבנות שרת אינטרנט על בסיס המודול זהה. אבל האמת היא שזה קשה ודורש המון קוד, ולאורך כל הספר חזרתי והסבירתי שהכח של Node.js הוא לא במודולים הטבעיים שלו שבאים כבירית מחדל אלא בספריה העשירה של המודולים שיש ב-`NPM`. אקספרס הוא בדיק מודול זהה שמאפשר לנו להרים שרת אינטרנט בקלות ויתר מכ-להוסיף לו פונקציונליות רובה ובקלות. בשורה אחת אני יכול להוסיף לאקספרס נתיבים, התנהוגיות, מודולי אבטחה, ניתוב, לוגים ועוד המון **פיצ'רים** שהיינו צריכים לכתחזק המון שורות כדי למש אוטם בעצמנו.

בפרק זהה אני אסביר מהו אקספרס. המודול עצמו מורכב מאד והוא יכול לאנלס ספר שלם, אז לא אוכל לסקור את כל הfonקציונליות, אבל כן אנסה את הדברים הבסיסיים מתוך הבנה שכשתרצו להרחב – תוכלו לעשות זאת בעזרת הדוקומנטציה העשירה של אקספרס. הפרק הזה הוא גם דוגמה לדרך שבה עובדים עם Node.js בעולם האמיתי.

ראשית, ניצור פרויקט חדש באמצעות יצרת תיקיה חדשה בשם `my-express-server` והפעלת `npm init` כדי ליצור את `package.json`. עכשו אנו יכולים להתחיל. ניצור את `app.js` ונתקין את אקספרס. איך? `i` `npm` – נוכל לראות שאקספרס נוצר ב-`node dependencies`. עכשו אפשר להשתמש בו ב-`app.js`. הדיבקו את הקוד הזה והריצו את `:app.js`

```
const express = require('express');
const app = express();

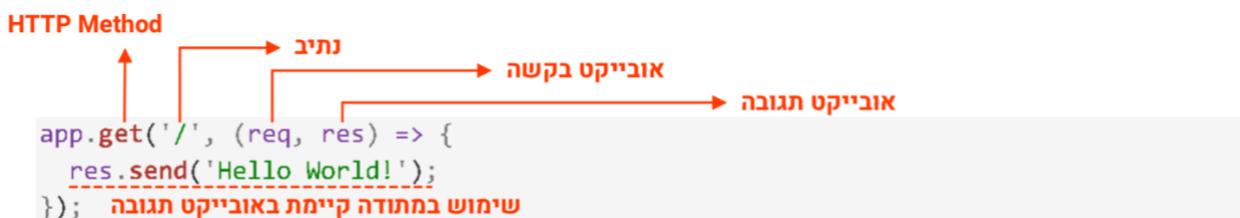
app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

app.get('/', (req, res) => {
  res.send('Hello World!');
});
```

```
app.get('/moshe', (req, res) => {
  res.send('Hello Moshe!');
});
```

אם תנווטו אל 000 localhost:3000 תוכלו לראות Hello World! ואם תנווטו אל localhost:3000/moshe תוכלו לראות Hello Moshe!.

אפשר לראות שמדובר בקוד שמאוד דומה לשרת http טבעי שבו כבר מכירים. ראשית אנו קוראים למודול ואז אנו יוצרים את השירות באמצעות קרייה לפונקציה הבונה במודול express ויצרים הפניה לשרת קבוע app. נודיע לשרת להאזין לפורט 3000. מהרגע זהה אנו יכולים לבצע בעצם האזנה לתנועה שמגיעה מהדף במתודת GET לניבים שונים.



ה-req וה-res הם אובייקטים. ה-req שהוא קיצור ל-request – בקשה, הוא הקריאה המגיעה מהלקוח (כלומר הדף) – הכוללת פרטיהם השונים (כמו למשל headers או body). ה-res שהוא קיצור ל-response – תגובה, הוא התגובה המגיעה מהשרת ללקוח, הלא הדף, וגם פה יש פרמטרים שונים.

בדוגמה זו אני משתמש במתודת send כדי לשלוח מחרוזת טקסט כתגובה ללקוח. איך ידעתו לבדוק מהי המתודה הזו? יש בדוקומנטציה של אקספרס הסבר מפורט עליה ועל שאר המתודות באובייקט response.

<https://expressjs.com/en/4x/api.html#res.send>

באוטו מקום נמצאת גם הדוקומנטציה על אובייקט request. פעמים רבות מתכוונים נמנעים מלהיכנס לדוקומנטציה, וחבל – בדרך כלל הדוקומנטציה של המודולים מספק טובה כדי להבין איך להשתמש בהם באופן אופטימלי.

טיפול במתודות של בקשות HTTP

עד עכשיו דיברנו על בקשות GET. בפרוטוקול HTTP יש לנו כמה וכמה סוגים של בקשות. אנו מכירים את בקשת GET שהדף משלח כאשר אתם גולשים ל-URL מסוים. כשאני נכנס לאתר, כמו localhost:3000 או google.com או localhost:3000 מהפרוטוקול של HTTP. גם בדוגמה שלעיל אני מראה איך אני מאמין לתנועה של GET עם מתודת .get

אם תשתמשו בכלים המפתחים בדפדפן, תוכלם לראות את בקשת ה-GET בклות בתוך טאב ה-**:Network**

The screenshot shows the Network tab in a browser's developer tools. A single request is listed under the 'posts' entry. The 'General' section shows the following details:

- Request URL:** `http://jsonplaceholder.typicode.com/posts`
- Request Method:** GET
- Status Code:** 304 Not Modified
- Remote Address:** 23.23.157.114:80

The 'Response Headers' section shows the following headers:

- Access-Control-Allow-Credentials:** true
- Cache-Control:** no-cache
- Connection:** keep-alive
- Content-Length:** 0
- Date:** Thu, 18 Feb 2016 16:21:52 GMT
- Etag:** W/"6b80-uPwhAkDat3Fl5TugzmyYpQ"
- Expires:** -1
- Pragma:** no-cache
- Server:** Cowboy
- Vary:** Origin
- Via:** 1.1 vegur
- X-Content-Type-Options:** nosniff
- X-Powered-By:** Express

יש גם מתודות אחרות שהדף יכול לשלוח באמצעות ג'אוوهסקריפט שנמצאות לצד הלקות.

סוג בקשה	פירוט
POST	בקשה ליצירה – יכולה להכיל payload – כולל מחרוזת טקסט, מספר או אובייקט שנשלח לשרת. השרת יכול לחת את המידע הזה ולהשתמש בו ליצירת משאב כלשהו.
PUT	בקשה לעדכון – הבקשה יכולה להכיל payload בדיק נמו POST. השרת-Amor לחת את המידע הזה ולהשתמש בו לעדכון משאב כלשהו. במקרה או ב-URL שאליו הבקשה נשלחת יהיה ID של המשאב שאותו מעדכנים.
DELETE	בקשה למחיקה – בדומה ל-GET אין כאן payload. ב-URL שאליו הבקשה נשלחת יהיה ID של המשאב שאותו מוחקים.
PATCH	בקשה לעדכון חלקו – הבקשה דומה ל-PUT ומשתמשים בה לעדכון חלקו של המשאב.

כאשר אני נמצא לצד הלקות, אני יכול לשלוח בקשות במתודות שונות אל השרת. השרת, שעליו אנו אחראים,-Amor לדעת לטפל בבקשת האלו.

אנו יוצרים בקשות כאלה באמצעות פקודה `fetch` בג'אוوهסקרייפט הצד הלקוח, קלומר בדף.

היכנסו אל `localhost:3000` והקלידו בקונסולה את הפקודה הבאה:

```
fetch('http://localhost:3000/', {
  method: 'POST',
  body: JSON.stringify({ data: 'sampledata' })
})
.then(response => response.text())
.then(data => console.log(data));
```

זהי פקודה שכתכני ג'אוوهסקרייפט אתם אמורים להכיר – היא מבצעת שליחת בקשה POST אל `localhost:3000` עם המידע:

```
{ data: 'sampledata' }
```

אם תדבוקו אותה בקונסולה של כלוי המפתחים, תקבלו שגיאה. מדוע? כי השרת שיצרנו יודעת להתמודד רק עם בקשות GET ולא עם POST. אם נשתמש בMETHOD post הוא ידע לטפל בבקשתות כאלה.

היכנסו את הקוד הבא ל-`app.js`:

```
const express = require('express');
const app = express();

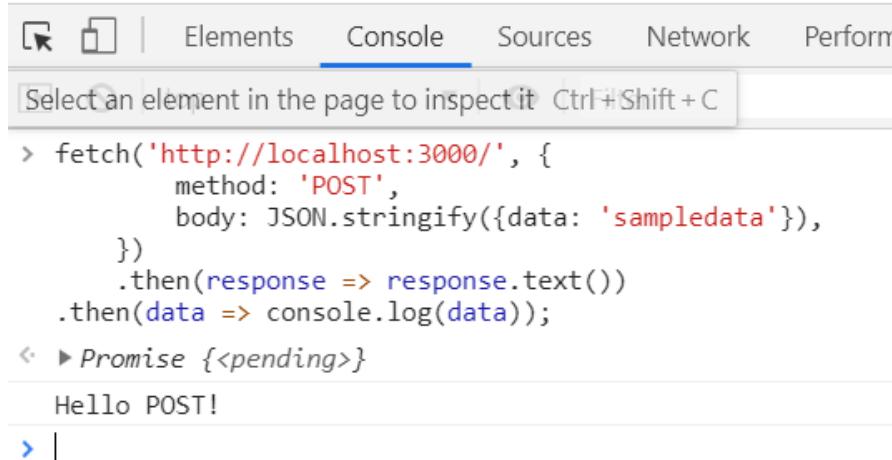
app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.post('/', (req, res) => {
  res.send('Hello POST!');
});
```

אפשר לראות שבשורות התחתיות אני מורה לשרת, כאשר הוא מקבל בקשה `post`, להגיב עם Hello POST!. אני אפעיל את השרת שוב (זהו צעד חשוב! אחרי כל שינוי קוד בשרת אני חייב

להרוג את התהיליך שרצה בטרמינל באמצעות קונטROL + C ולפתחו אותו מחדש באמצעות node.js.app אחרת שינוי הקוד לא יעודכנו ולא ירוצו). אז אפתח את localhost:3000 בדף. אדיבק את הקוד ששולח בקשה POST ואוכל לראותشبוקונסולה של הדף אני מקבל! Hello POST!

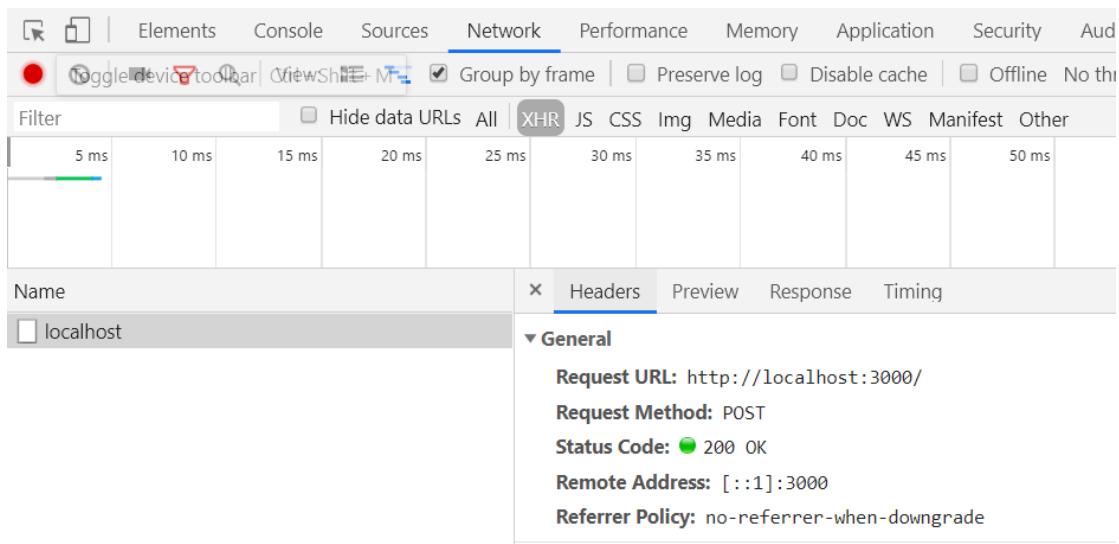


```

Elements | Console | Sources | Network | Perform
Select an element in the page to inspect it Ctrl+Shift+C
> fetch('http://localhost:3000/', {
    method: 'POST',
    body: JSON.stringify({data: 'sampledata'}),
})
  .then(response => response.text())
  .then(data => console.log(data));
< ▶ Promise {<pending>}
Hello POST!
>

```

אם אפתח את לשונית ה-Network בכל המפתחים, אני אוכל לראות שהבקשה אכן נשלחה במתודת POST והתקבלה תגובה מהשרת.



איך ניתן את הבקשות האלה? כמו שיש לנו get כדי לטפל בבקשת GET, יש לנו post כדי לטפל בבקשת POST ובהתאם put, delete ו-patch גם all, כדי לטפל בכל הבקשות. שימושו לב שיכ说得 במתודות של HTTP אלו כותבים באותיות גדולות: למשל, GET, POST, DELETE.

אנו צריכים לזכור של מרבית שמודול express עוטף את הכל – עדין מדובר במודול שזהה מבחינה תפקודית ו מבחינה שימושית לשרת ה-HTTP הבסיסי שיצרנו. כך למשל, אם אנו רוצים להחזיר דף HTML בסיסי (ולא סתם טקסט של Hello World) – אני עושה זאת בדיק כמו בשרת רגיל. אם למשל ארצה לשחרר את הדוגמה שהראיתי בפרק על טיענות משבאים באמצעות `path`, ולשגר אל המשמש דף HTML סטטי, שנמצא בתיקייה `static`. אני אעשה זאת באמצעות קרייה של קובץ והזרמתו למי שנכנס לאתר. קובץ `index.js` ייראה כך:

```
const express = require('express');
const app = express();
const fs = require('fs');
const path = require('path');

app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

app.get('/', (req, res) => {
  fs.createReadStream(path.join(__dirname,
  './src/static/index.html')).pipe(res);
});
```

אפשר לראות שאני קורא את התוצאה אל סטרום באמצעות מודול `fs` ומשרג אותה באמצעות `pipe` אל ה-`res` בבדיקה כמו שרת HTTP בסיסי וטבעי של Node.js. בהנחה שהקובץ `index.html` באמת קיים בנטייב `src/static` – אני אראה את הקובץ אם אפעיל את השרת באמצעות `node index.js`. ואכן עם הדפסן לכתובת `localhost:3000`. זה כמובן בסיסי מאוד ולא משהו שמקובל לעשות. באקספרס יש לנו מנوعי רנדורי שעלייהם נרחיב בהמשך הפרק.

ראוטינג

בתה-הפרק הקודם טיפלנו במתודות, ועכשו נטפל בנתיבים. כישיש לי אתר קtan עם כמה נתיבים, למשל עמוד הבית ודף help, לא צריך לשבור את הראש. ה-.js.app שלו יוכל להכיל כמה נתיבים:

```
const express = require('express');
const app = express();
const fs = require('fs');
const path = require('path');

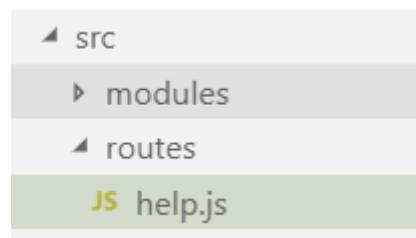
app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

app.get('/', (req, res) => {
  fs.createReadStream(path.join(__dirname,
    './src/static/index.html')).pipe(res);
});

app.get('/help', (req, res) => {
  res.send('Help page!');
});
```

אבל מה קורה כשהיש לי כמה וכמה נתיבים סטטיים? 10? 100? 1,000? להכניס את הכל לתוך רשימה ארוכה ב-.js.app איננו רעיון טוב. מה עושים? בדיק בשביל זה יש לנו ראוטינג, או ניתוב בעברית, זו פשוט דרך לנתח את הבקשות השונות לקבצים שונים. זה עובד בדיקות כמו מודולים. אנו יוצרים מודול של ראוטינג ומיצאים אותו. מי שצורך אותו הוא האפליקציה שלנו ב-.app.js

כדי להדגים, ניצור ראוטר לדפי עזרה. מקובל לשים את כל המודולים של הראוטרים בתיקייה :routes



כأن יצרתי את `js.help`. זהו מודול שבו יהיו נתיבים שונים שהיו מתחת `help`. למשל:

```
localhost:3000/help/about-me
localhost:3000/help/how-to-use
```

בקובץ זהה יהיה מודול הראوتر, שמעט הבדל קטן אחד הוא מתנהג בדיק במו הפניות שראינו ב-`:app.js`:

```
const express = require('express');
const router = express.Router();

router.get('/about-me', (req, res) => {
  res.send('About me page');
});

router.get('/how-to-use', (req, res) => {
  res.send('How to use');
});

module.exports = router;
```

מה ההבדל הקטן אך המשמעותי? במקרים של ראוטינג אני משתמש ב-

```
const router = express.Router();
```

ואת כל הראוטינג אני עושה באמצעות `router.get` ולא `app.get`. בהמשך אני מיציא את המודול הזה כמו מודול רגיל.

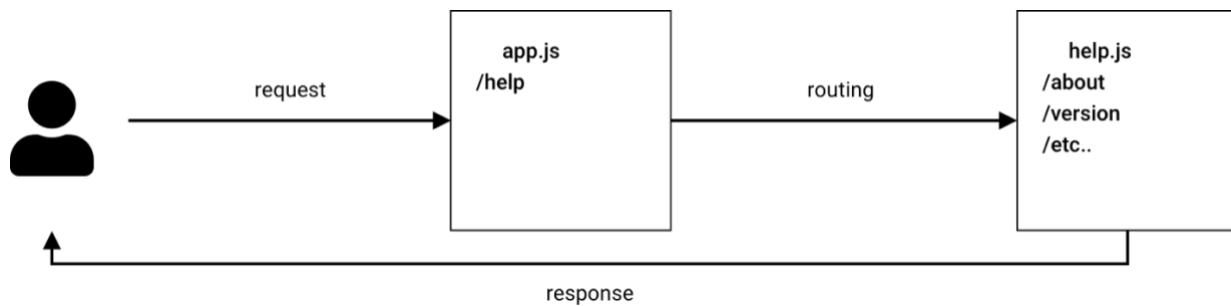
איך אני צריך אותו? באמצעות `app.use` – זהו מתודה שנותרת באופן טבעי-קדם של אקספרס ואפשר להשתמש בה ממש בקלות. היא עצמה מקשרת בין הקוד שרצץ ב-`app` לבין הראوتر הרלוונטי, במקרה שלנו `help`. ב-`use` אני מקשר בין כתובות הנתיב, שהיא טקסט פשוט, כך, שהוא משתנה שמניל רפרנס לבין הראوتر:

```
const express = require('express');
const app = express();
const help = require('./src/routes/help');

app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

app.use('/help', help);
```

בעם ב-`app` הגדרנו את הנתיב `/help` וצינו שמי שמתפל בו הוא הראوتر. הראوتر של `help` מטפל בשני נתיבים: `about-me` ו-`how-to-use`, והם יהיו זמינים מיד אחרי הנתיב שהראوتر מטפל בו – במקרה זה `help`, ככלומר הכתובות `/about-me` ו-`/how-to-use`.

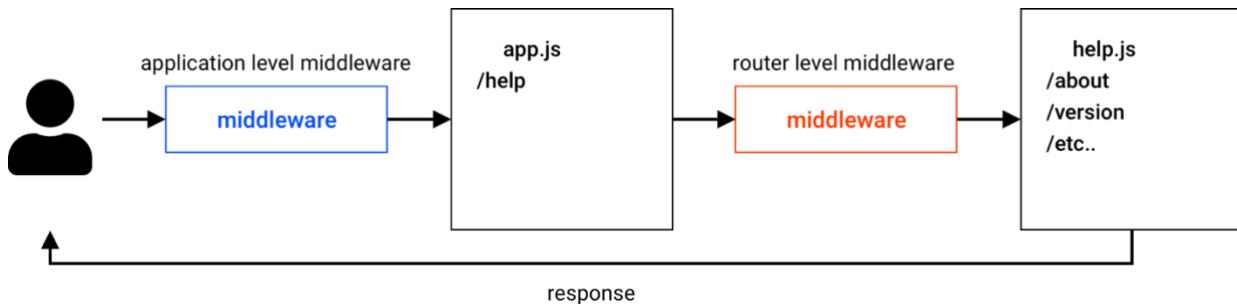


ה-`use`, שבו או משתמשים כדי לחבר את מודול הראوتر שלנו אל הנתיב, הוא מתודה שבה אנו משתמשים כדי לחבר רואוטינג אבל גם **Middleware**.

MiddleWare

אותה הkopנספציות החשובות ביותר באקספרס היא עבודה עם Middlewares. MiddleWare זהה בעצם ייחידת תוכנה שהבקשות והtagיות עוברות דרכה ואנחנו יכולים לבצע פעולות שונות. למשל, אם יש לנו מוכחת לוגים, אני משתמש ב-`Middleware` על מנת לבצע רישום של בקשות במערכת. אם למשל אני מקבל מהמשתמש תМОנות ואני רוצה ליצור אותו לפני שאני כותב אותו לדיסק הקשיח – אני אעשה את זה ב-`Middleware`. העיקרון העומד מאחורי `Middleware` הוא בעצם לקבל תנועה, לטפל בה וואז להעביר אותה להמשך טיפול. מדובר בעצם בפונקציה שמקבלת שלושה ארגומנטים: `(req, res, next)`. הארגומנט הראשון הוא הבקשה, `request`; השני הוא התגובה – `response`. כבר הסבירנו וdone באובייקטים האלו מוקדם יותר בפרק.

`next` הוא פונקציה שאנו קוראים לה בסיום העבודה שלנו כדי להעביר את התנועה להלאה. אנו יכולים להשתמש ב-`Middleware` בכל שלב באקספרס.



הבה נדגים באמצעות `Middleware` שבמצע לוגינג. בפרויקט שלנו ניצור תיקיה בשם `logger` ששם הוא `middlewares`, בה ניצור קובץ שנקרא `logger.js` ובו נכניס את הקוד הבא:

```
const myLogger = (req, res, next) => {
  console.log('Someone entered to the site', Date.now());
  next();
};
```

```
module.exports = myLogger;
```

שימוש לב שיש לנו כאן שלושה פרמטרים – `req`, `res` ו-`next`. אני יכול בשלב זה לקרוא את הבקשה מהлокוח, ולכתוב לתגובה שהוא קיבל בעתיד. במקרה זה אני רק כותב ל-`console.log` ומיד אחר כך קורא ל-`next`. מה ש-`next` עשו הוא להעביר את התנועה להלאה. אם אני לא אקרא

ל-next, התנועה לא תעבור והבקשה לא תושלם, אז זה חשוב. אם יש בקשות, קריאות, כתיבה לדיסק המקומי וכו' – ה-next ייקרא מיד אחרי שהפרומים יושלמו.

איך אנו משתמשים ב-Middleware? פשוט יותר, מבצעים require לモודול שלנו, במקרה הזה ל-app logger.js middleware use:app.logger

```
const logger = require('./src/middleware/logger');

app.use(logger);
```

אני יכול לעשות use.app בכל מקום. אם אני רוצה להשתמש בו בראוטינג, גם אין בעיה – אני פשוט משתמש ב-router. אם אני אזכיר את ה-Middleware הזה בראוטר של help שלמדנו עליון בתחילת הפרק, הוא ייראה כך:

```
const logger = require('../middleware/logger');

router.use(logger);
```

ב-Middleware אנחנו יכולים לעשות דברים בהתאם ל-request ולשנות את ה-response. מובן שהכוח הגדול של אקספרס, בדומה ל-NPM, הוא שעת רובה Middleware אנו לא צריכים לכתוב בעצמנו אלא אנו יכולים להשתמש ב-Middleware שאנשים כתבו ופורסםם ב-NPM. בדיק נסוב כל מודול אחר. פשוט משתמשים במודול הזה אך ורק באקספרס.

למשל – לוגר. בדוגמה שלילית כתבנו Middleware מאפס. אבל במצבות אם נרצה לוגר, סביר להניח שנחפש מודול ב-NPM שעושה את זה, למשל מודול כמו morgan.

<https://www.npmjs.com/package/morgan>

ה-Middleware הזה הוא מודול של NPM והוא מותקן בדיק נסוב כל מודול:

```
npm i morgan
```

איך משתמשים בו? בדיקן כמו Middleware שכתבנו בעצמנו, כאשר במקרה זהה (ובכמעט בכל המקרים) יש הוראות מדויקות להפעלה בדף המודול. במקרה של `morgan`, מפעילים אותו כך:

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

app.use(morgan('common'));
```

אם תפעילו אותו במקום המודול של `logger` באפליקציית האקספרס, תפעילו את האפליקציה ותיכנסו אל `localhost:3000`, תוכלו לראות בקונסולה את הכניסות עם מעט יותר מידע. אפשר ליצור בקלות לוגים מוחכמים, שאוגרים מידע נוסף או שכותבים לקובץ. איך? הכל בדוקומנטציה של `morgan`. הדוקומנטציה לא טוביה? ה-`Middleware` לא טוב מספיק? נחפש אחר או שנתרום לו קוד שישפר את המצב.

הכוח המרכזי של אקספרס הוא בספרייה העצומה של המודולים שבאה אליו. לפני שאתם מתחילהים לכתוב קוד בעצמכם, אפשר ורצוי לבדוק אם יש מודול שמתפל בעזה. אקספרס הוא פופולרי ביותר, וסביר להניח שרוב הדברים שחשבתם עליהם כבר פותחו ונמצאים ב-NPM.

URL דינמי

מוקדם יותר בפרק הסבבונו על רואוטינג וראינו איך אנו מגדירים כתובות שונות לאתר באמצעות אקספרס. כתובות כאלה נקראות **כתובות סטטיות**. למה סטטיות? כי כתבנו, ממש בקוד, את הנתיבים ברואוטינג. אבל רוב האתרים מכילים כתובות דינמיות. אם יש לנו אתר שאליו מתחברים ולכל אדם יש פרופיל משתמש, לא נגדיר ידנית עבור כל משתמש, רואוטינג משלו. אם יש לנו אתר שיש בו מאמרים רבים, לא נגדיר עבור כל מאמר ומאמר כתובות ברואוטינג. בבדיקה בשליל זה יש לנו כתובות דינמית, כתובות שבה יש פרמטר מסוים משתנה כל הזמן ואקספרס יכול לקרוא אותו ולהציג מיד תוכאה שונה לכל משתמש ומשתמש.

הבה נדגים באמצעות רואוטינג של `user/NAME`. ה-`NAME` הוא בעצם משתנה דינמי, כלומר אם אני נכנס לאתר בכתובת: <http://localhost:3000/user/moshe> אני אזכה לראות התichשות `moshe`. לפרטט `moshe`.

הדבר הראשון שנចטרך לעשות הוא להגדיר ראותינג שיטף בבקשת שומות לנתיב `user`.
בתיקיית `routes` בפרויקט האקספרס שלנו, ניצור קובץ שנקרא `user.js` ונותר אותו ריק. נקבע
לו בקובץ ה-`app.js` שמוביל את האפליקציה שלנו באופן זהה:

```
const express = require('express');
const app = express();
const fs = require('fs');
const user = require('./src/routes/user');

app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

app.use('/user', user);

app.get('/', (req, res) => {
  fs.createReadStream('./src/static/index.html').pipe(res)
});

});
```

כל הקוד שלעיל אומר לנו מוכר מאוד כיון שעברנו עליו מוקדם יותר בפרק. בעצם ניגש
למודול הראותינג של `user` ונגדיר את ה-URL הדינמי. זה הרבה יותר קל ממנו שוחובים. כאשר
אנו רוצים נתיב דינמי, אנחנו צריכים פשוט להוסיף נקודותים בנתיב, לפני שם הפעמטר. למשל,
אם החלטתי שאני רוצה פונטר בשם `name` יהיה `:name`, כך אני מקבל אותו:

```
const express = require('express');
const router = express.Router();

router.get('/:name', (req, res) => {
  res.send(`User ${req.params.name} entered the system`);
});

module.exports = router;
```

הקוד הזה יהיה בראوتر של ה-`user`. קלומר ב-`./src/routes/user`. אני ניגש אל הפעמטר בכל
מקום באמצעות `req.params.name`. כאמור ה-`req` הוא אובייקט הבקשה שבו מקבלים ב-`Middleware`. אחת התכונות שלו היא
`params`, אובייקט נוסף אליו ממויינים כל הפעמטרים כשייש בבקשת. קלומר אם יש פעמטרים

של POST או GET – הם יהיו שם. את זה לא נראה במודול http קלאסי אלא זו פונקציונליות שיש באקספרס, אחת מרבות שעזרו למודול זהה להפוך לאחד הפופולריים ביותר.

אני יכול לקרוא לפרמטר שלי בכל שם, למשל השם המופרך ahla גם הולך – השם עצמו לא משנה כלל, הוא רק קובע איך אני אקבל אותו ב-req.params:

```
const express = require('express');
const router = express.Router();

router.get('/:ahla', (req, res) => {
  res.send(`User ${req.params.ahla} entered the system`);
});

module.exports = router;
```

אם אני אכנס אל:

<http://localhost:3000/user/moshe>

אני אוכל לראות את הכתוב:

User moshe entered the system

אני יכול להשתמש בכמה פרמטרים בנתיב ללא הגבלה, למשל:

```
router.get('/:name/:id', (req, res) => {
  res.send(`User ${req.params.name} entered the system. The ID is
${req.params.id}`);
});
```

כאן יש לי שימוש בשני פרמטרים: id ו-name. אם אני אכנס אל:

<http://localhost:3000/user/moshe/1>

אני רואה את הכתוב המתאים.

שימוש לב שם אני מגדיר נתיב דינמי – אני חייב להכניס את כל המספרים. אני לא חייב להשתמש רק ב-router.get אלא יכול להשתמש גם ב-post או בכלל מתודה אחרת.

tabniot

עד עכשו החזרנו שני סוגי תגיות – דף HTML סטטי, שלו קראנו באמצעות `fs.read` או תגובה פשוטה באמצעות `res.send`. אבל בחים האמיתיים פעמים רבים אנו צריכים להשתמש בתבניות – ככלומר שילוב של דף סטטי עם נתונים דינמיים. בדוק בשביב זה אנו צריכים להשתמש בתבניות. יש כמה סוגים התבניות שאקספרס תומך בהם ואנו נלמד כאן על `ejs`, הסוג הפופולרי ביותר לאתרים סטטיים פשוטים. אנו נלמד עליו בעת כדי להציג איך מנוע רנדום עובד, למרות שבעולם האמיתיסביר להניח שתעבדו עם ריאקט, אנגולר או שע.

`ejs` לא בא כברירת מחדל עם אקספרס ויש להתקינו באמצעות:

```
npm install ejs
```

אחרי שהתקינו את המנוע הזה, יש צורך להוראות לאפליקציית האקספרס שלנו להריץ אותו. עושים את זה באמצעות הוראה מפורשת בקוד שמצויבים ב-`-ejs`, הקובץ המרכזי של האפליקציה שלנו, באופן זהה:

```
app.set('view engine', 'ejs');
```

מהנקודה הזו אנו יכולים להשתמש במנוע התבניות. ניצור תיקיה מהנתיב הראשי שנΚראת `views`. השם הוא ברירת מחדל והוא מכיל את כל התבניות. בהה ניצור תבנית בשם `index.ejs` ומכניס בה HTML רגיל וסטנדרטי:

```
<html>
    <h1>Hello world from template!</h1>
</html>
```

עכשו אנו יכולים להשתמש בתבנית זו בדוק כמו בקובץ סטטי. הדרך להשתמש בה היא פשוט להשתמש בMETHOD `render` שנמצאת ב-`result` ולהכניס את שם התבנית, במקרה שלנו `index`. אם למשל אני רוצה לקרוא לתבנית זו המנתיב המרכזי והראשי, ב-`-ejs`, אני אכתוב (עוד לפני הרואTING):

```
app.get('/', (req, res) => {
    res.render('index');
});
```

אם אני אכנס אל:

<http://localhost:3000/>

אני אוכל לראות את הדף. בדיק כמו דף סטטי זהה נחמד, אבל היתרון בתבניות הוא שאני יכול לדוחף פנימה משתנים, אלו משתנים שבא לי – בתור הארגומנט השני של מתודת `render`. למשל, בואו נכניס `subtitle`. אני אצור אובייקט עם `subtitle` ופושט עביר אותו כארגומנט השני, ממש כמו:

```
app.get('/', (req, res) => {
  res.render('index', {subtitle: 'This is subtitle'});
});
```

מעכשיו בתבנית שלי יש משתנה ששמו הוא `subtitle`. הוא יכול להיות כל דבר. במקרה הזה הוא יהיה מחוזת טקסט, אבל הוא יכול להיות הכל. על מנת להציג את המשתנה הזה, אני צריך להשתמש בתחביר מיוחד – חיצים עם אחוז. זה בעצם השימוש בתבנית:

```
<html>
  <h1>Hello world from template!</h1>
  <h2><%=subtitle%></h2>
</html>
```

אם אני אכנס אל:

<http://localhost:3000/>

אוכל לראות את הטקסט המלא כפי שהכנסתי אותו. כאמור, זה יכול להיות משהו נחמד יותר. אנו יכולים להכניס את הפרמטרים האלה מכל מקום, למשל מתוך URL דינמי:

```
router.get('/:name/', (req, res) => {
  res.render('index', {name: req.params.name});
});
```

וההדפסה תהיה מראה כזה:

```
<html>
  <h1>Hello <%=name%></h1>
</html>
```

אני גם יכול להכניס מערכים – מה שיש ב-*aze* זה ג'אוوهסקריפט לכל דבר ועניין. למשל במקום הקוד שלעיל, אני יכול לכתוב קוד כזה:

```
router.get('/:name/', (req, res) => {
  res.render('index', {params: req.params});
});
```

וחתנית תיראה כך:

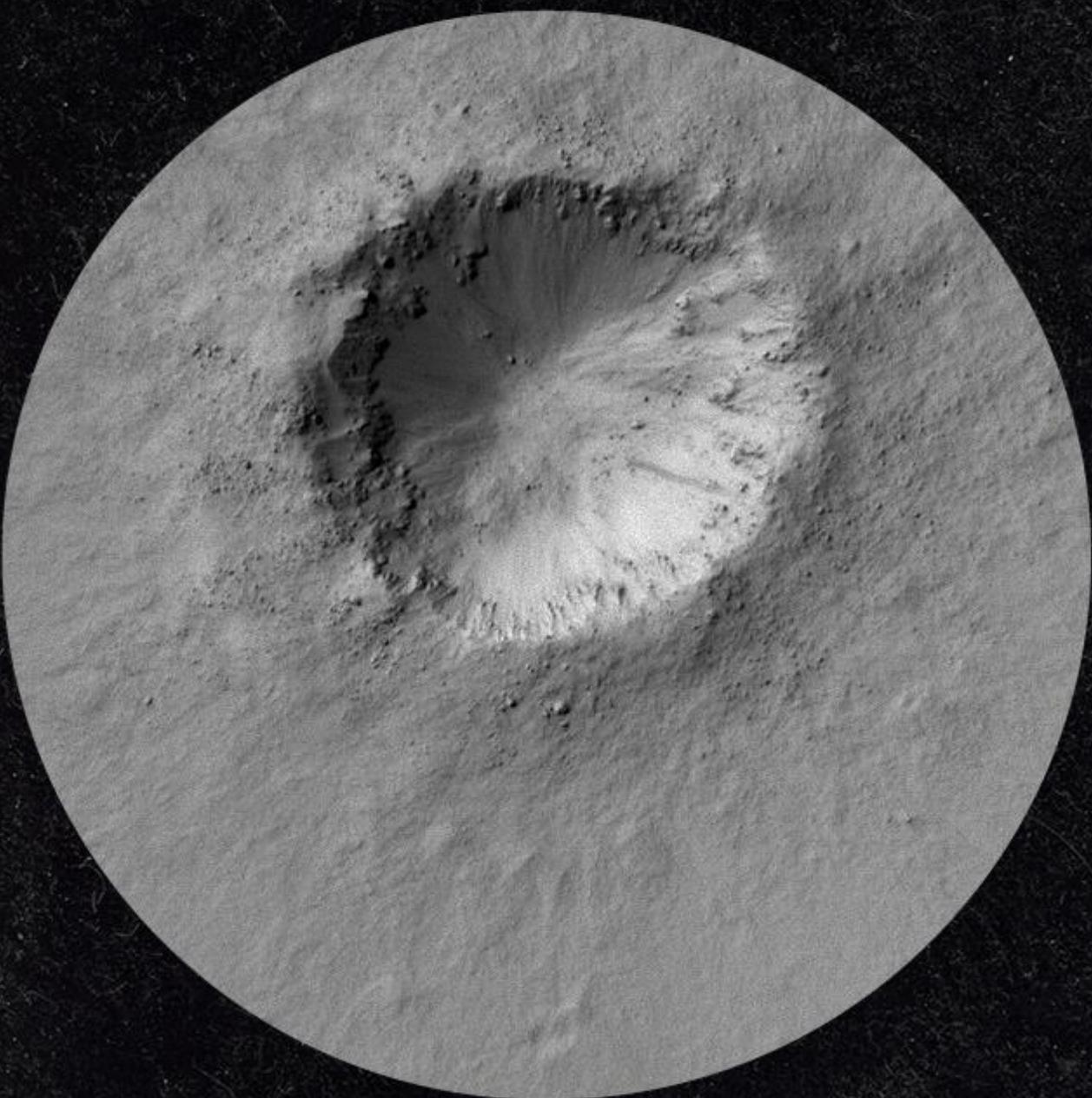
```
<html>
  <h1>Hello <%=params.name%></h1>
</html>
```

בתבניות *aze* יש המון אפשרויות נוספות, כמו תת-tabיות, לולאות ותוספות נוספות, אבל הבסיס מספיק ברוב האפליקציות. ברוב המקרים אנו משתמשים בספרייה צד שלישי כמו ריאקט על מנת לבנות את התבנית, בעוד בצד השרת אנו משתמשים באקספרס כדי לבנות את ה-API – ככלומר את הממשק she-AJAX פונה אליו ואת התבנית הבסיסית שטוענת את ספריית הריאקט.

אפליקציה אמיתית באקספרס תכיל רואטרים שיגדרו את הנטיבים, או endpoint בrama מקצועית שקיימות AJAX יכולו לפנות אליה. באמצעות *Middlewares* יהיה בקרה אבטחתית, זיהוי ואוטנטיקציה. במודולים עצם יהיה המידע שיועבר לראוטינג. מי שיקבל את המידע זה צד הלוק, שמציג אותו באמצעות ריאקט, אנגולר או שע, וביהם ספר זה אינו עוסק.

פרק 18

חיבור ל-MYSQL



חיבור ל-MySQL

תוכנה אינה רק קוד אלא גם מידע. מידע מאוחסן במסדי נתונים. יש מגוון גדול מאוד של מסדי נתונים מסוגים שונים שלכל אחד יש יתרונות (וחסרונות). מסד נתונים מסוג MySQL הוא אחד ממשדי הנתונים הנפוצים בעולם. ספר זה אינו מלמד ממשי נתונים והפרק הזה מניח הבנה בשאלות בסיסיות ב-MySQL. במידה שאין לכם ידע בנושא, אפשר לדלג על הפרק הזה ולהמשיך לפרק הבא.

כמו בכל דבר, יש לא מעט מודולים שמתחררים ל-MySQL. אנו נבחר במודול הפופולרי ביותר mysql ונתקין אותו באמצעות npm. למודול זה יש דוקומנטציה נרחבת מאוד וברורה למדי. אנו עוברים על הבסיס פה, רק כדי לראות עד כמה זה קל. אתם מוזמנים לקרוא את הדוקומנטציה בעצמכם פה: <https://www.npmjs.com/package/mysql>

אנו נציג בשרת MySQL שעבוד. אני שוב יוצא מנקודת הנחה שאתם מכירים מספיק MySQL כדי להרים שרת זה בעצמכם. אפשר להרים MySQL על כל מערכת הפעלה: חלונות, מק או לינוקס. ההתקנה היא קלה וההרצה עוד יותר. אם יש לכם מסד נתונים שרצ במחשב, סביר להניח שהוא נמצאlocalhost בפורט 3306, אבל מסד נתונים יכול להיות בכל מקום, כמובן.

פרק זה אני משתמש בדוגמה במסד נתונים שנמצא על המחשב שלי, localhost בשם זה הוא test שמתחררים אליו עם שם משתמש root וסיסמה שהיא 123456. הטבלה היא clients ויש לה שלושה שדות: id (המפתח הראשי), name ו-city.

מבנה שבאתר אמת לעולם אל תשימושו בשם משתמש root ובסיסמה פשוטה. ונוסף על כן, שם המשמש והסיסמה צריכים להיות במשתני סביבה בפרק הקודם.

חיבור ראשוני

אנו מתחברים אל MySQL באמצעות מתודה אסינכרונית – כולם היא תבלום את כל הפעולות שיבאו אחריה. זו אחת הסיבות שחשיבות לביצוע החיבור פעם אחת ולשמור את הרפרנס הזה במקומם שקל להגיע אליו. החיבור הוא פשוט ונו אנו צריכים לספק את השירות שעליו מסד הנתונים, שם מסד הנתונים והסיסמה ושם המשתמש לחיבור. כך זה נראה:

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

const connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'root',
  password : '123456',
  database : 'test'
});

connection.connect();
```

מהרגע הזה יש לנו חיבור. אם נריץ את הקוד הזה (למשל נבדיק אותו ב-node app ונריץ את הקוד באמצעות node ./app) נראה שהתהליך נותר פתוח. הרצת הקוד לא מסתיימת כיוןSCP שכל עוד החיבור פתוח, הקוד רץ.

חשוב מאד לציין שהקוד המובא כאן הוא רק לצורך הדוגמה. לעולם לעולם אל לנו להכניס סיסמות בקוד המקור של האפליקציה שלנו. זוכרים את משתני הסביבה שעלייהם למדנו בפרק מוקדם יותר? זה מקום מצויין להשתמש בהם. למשל, ליצור משתני סביבה בנוסח זה:

```
HOST=localhost
USER=root
PASSWORD=123456
DATABASE=test
```

ולקרוא להם בקלות באמצעות הבא:

```
const dotenv = require('dotenv');
dotenv.config();

const mysql = require('mysql');

const connection = mysql.createConnection({
  host: process.env.HOST,
  user: process.env.USER,
  password: process.env.PASSWORD,
  database: process.env.DATABASE
});

connection.connect();
```

אפשר לסגור את החיבור באמצעות:

```
connection.destroy();
```

ואז החיבור ייסגר והקוד יפסיק לרוֹץ.

בדרכ נכל לנו לא יוזמים חיבור ישיר אלא מבצעים pooling, כלומר יוצרים כמה חיבורים, שומרם אוטם בצד ומשתמשים בהם לפי הצורך. מודול MySQL מאפשר לנו לעשות את זה בקלות ובדרך דומה מאוד לחבר ישיר באמצעות מתודת `createPool`. מוגדר זו זהה אחת לאותה המתודה `createConnection` אלא שבניגוד אליה, אנו צריכים לומר כמה חיבורים צריך ליצור בצד. כך זה נראה:

```
const pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '123456',
  database: 'test'
});
```

אנו לא צריכים לבצע `connection`.`pool` מהנקודה הזאת, אנו יכולים להתחיל לעבוד. אנו כן צריכים לשמור את הקבוע `pool` בצד כי דרךנו אנו עושים את כל החיבורים. מקובל לשים את יצירת החיבורים (באמצעות `pool` או `connect`) במודול נפרד ולבקש את ה-`pool` או את ה-`connection` באמצעות מתודת `get` ייועודית.

הערה חשובה: אם אתם מקבלים את השגיאה זו:

Client does not support authentication protocol requested by server;
consider upgrading MySQL client

הרייצו את השאלה הבאה בשורת ה-MySQL:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password  
BY 'password';
```

כשה-word password היא הסיסמה שלכם.

שאילתת בסיסית

אחרי שיש לנו חיבור, אנו יכולים להתחיל לעבוד. נתחיל עם שאלה בסיסית של SELECT. אפשר לבצע שאלות ישרות מ-pool או מ-connect query. השאלה הזו מקבלת שני ארגומנטים. הארגומנט הראשון הוא השאלה עצמה, הארגומנט השני הוא קולבך שיש בו שלושה ארגומנטים: שגיאה, תוצאה (שנוגע כאובייקט) ומידע על השדות. ככה זה נראה:

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

const pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '123456',
  database: 'test'
});

pool.query('SELECT * FROM `clients`', (error, results, fields) => {
  // error will be an Error if one occurred during the query
  if (error) throw error;
  // results will contain the results of the query
  console.log(results);
  // fields will contain information about the returned results
  fields (if any)
  console.log(fields);
});
```

אנו יכולים להכניס כל שאלתה שהיא אל מתודת `query` ותמיד נקבל תוצאה או שגיאה. זה עד כדי כך קל. גם `INSERT` עובד או כל שאלתה אחרת. כן, כולל `DROP`. מודול MySQL של Node.js הוא בסופו של يوم גשר – גשר בין הקוד שלנו למסד הנתונים. כל מה שצריך לעשות הוא להחליט איזו שאלתה אנו שואלים.

המרת הקוד לעובדה עם פרומיסים ולא עם קולבקים

אפשר לראותות כמה זה קל. אבל יש עם זה בעיה: הקוד עובד עם קולבקים ולא עם פרומיסים. פתרון הבעיה הזה הוא קל יחסית, כיוון שאפשר להמיר כל קולבק לפרומיס באופן הבא:

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

const pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '123456',
  database: 'test'
});

function CreateQuery(pool) {
  return new Promise((resolve, reject) => {
    pool.query('SELECT * FROM `clients`', (error, results, fields)
=> {
      // error will be an Error if one occurred during the query
      if (error) return reject(error);
      // results will contain the results of the query
      resolve(results);
    });
  });
}

CreateQuery(pool).then((results) => {
  console.log(results);
});
```

זו היא טכניקה שמתכונתי ג'אווה סקריפט אמורים לשנות בה. אנו יוצרים פונקציה ושם אנו מוחזירים פרומיס. בתוך הפרומיס אנו מבצעים את הקריאה האסינכרונית עם הקולבק ובתוך הקולבק מבצעים resolve או reject. עכשווי אפשר לקרוא לפונקציה שיצרנו ולקבל פרומיס כרגע, או להכניס אותה לתוך `:async/await`

```

יצירת החיבור
const mysql = require('mysql');

const pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '123456',
  database: 'test'
});

פונקציה שמחזירה פרומיס
function CreateQuery(pool) {
  return new Promise((resolve, reject) => {
    pool.query('SELECT * FROM `clients`', (error, results) => {
      if (error) reject(error);
      resolve(results);
    })
  });
}

קריאה רגילה לפונקציה שמחזירה פרומיס
CreateQuery(pool).then((results) => {
  console.log(results);
});

```

אנן חיבטים להעבירה את החיבור

از בוגע לקולבקים, אין בעיה. פשוט צריך לזכור שבחיהם האמיתיים ולא בספר או בדוגמאות. נעצוף תמיד את הקוד של הקולבקים בקוד שמחזיר פרומיס.

Prepared Statement

בעיה גדולה יותר עם כתיבת Queries ישירות באופן זהה היא שמדובר בפתח לצורות אבטחה גדולות. אנו תמיד חייבים להשתמש ב-prepared statement כדי להימנע מ-SQL injections. קוד ללא prepared statement הוא קוד סופר בעייתי. ברגע שモומחה אבטחה יראה קוד שיש בו queries שנכנסות ישירות למסד הנתונים הוא יפסול את הקוד הזה. זו הסיבה שתמיד חייבים להשתמש ב-prepared statement כדי למנוע מראש בעיות אבטחה.

ב-prepared statement אנו בעצם יוצרים תבנית של שאלתה ואז יוצאים אליה את הנתונים. זה נראה מסובך אבל המימוש של זה הוא פשוט. כותבים את ה-Query כרגיל אבל במקום הנתונים מציבים סימן שאלה [?] וublisherים את הארגומנטים שמחליפים את סימן השאלה לפי הסדר:

```
pool.query('SELECT * FROM `clients` WHERE id = ? ', [1], (error,
results) => {
  if (error) reject(error);
  console.log(results);
});
```

כאן יש לנו נתון אחד, ה-`p`. אנו מחליפים אותו בסימן שאלה וublisherים את הערך שלו במערך. במקרה הזה יש לנו רק נתון אחד. אם יש לנו כמה, אין בעיה – רק צריך להקפיד על הסדר במערך:

```
pool.query('SELECT * FROM `clients` WHERE name = ? AND city = ?',
['Moshe', 'Petah Tiqwa'], (error, results) => {
  if (error) reject(error);
  console.log(results);
});
```

סימני השאלה יכולים לבבל, אבל זה בהחלט פשוט – בשאליתה שלכם אסור שייהיו נתונים שמאגים מבחן. אנו מציבים סימן שאלה בכל נתון כזה וublisherים את הנתונים מבחן במערך לפי הסדר של סימני השאלה. זה נעשה כי ברוב המקרים הנתונים לשאלות מגיעים מקלט של המשתמש ואנחנו ממש לא רוצים לקלוט פה injection:

```
pool.query('SELECT * FROM `clients` WHERE name = ? AND city = ?',
  ['Moshe', 'Petah Tiqwa'], (error, results) => {
  if (error) reject(error);
  console.log(results);
});
```

אפשר לראות כמה קל להשתמש ב-MySQL יחד עם Node.js באמצעות מודול ייחודי, ומודולים כאלה קיימים לכל מסד נתונים שיש. זו הגדולה והכוה של Node.js.

פרק 19

עליה לפורודהשו



עליה לפודקשן

למදנו איך מרים שרת HTTP, ללא אקספרס ועם אקספרס, איך מרים שרת שתומך בסוקט, ונשאלת השאלה – איך מעלים הכל לרשות? אם יצרתי אתר או API או שירות כלשהו מבוסס Node.js, איך אני מעלה אותו לרשות? איך אני מאפשר לאנשים אחרים להשתמש בו? איך אני עולה לסייע אמיתית? זה נקרא עליה לפודקשן – סביבת ייצור אמיתית.

יש כמה דרכים להעלות קוד של Node.js. הראשונה היא לשכור שירות אמיתי, Private Server או מכונה וירטואלית על שירות – Virtual Private Server, ולהפעיל את הקוד ממש בדיקון כמו שהפעלנו אותו מהמחשב בזמן הלימוד. צריך לזכור שבסוףו של דבר שירות זה מחשב, ויש לו מערכת הפעלה (חלונות או לינוקס). לעיתים אין מערכת הפעלה גרפית ומה שיש לנו זה טרמינל בלבד. אנו מתחברים מרוחק לטרמינל ומפעילים את האתר, את האפליקציה או את השירות מבוסס ה-Node.js באופן דומה למחשב שלנו.

דרך שנייה היא באמצעות שירותי ענן. יש לא מעט שירותי ענן בעולם: AMAZON, AZ'OR והרוכקו הם המובילים. שירותי הענן דואג בעצמו לכל – הוא יפרוש עוד שירותי וירטואליים אם הוא חש בעומס על השירות שלנו, למשל. יהיה בו גיבוי והפרישה תהיה אוטומטית. רוב החברות עובדות עם שירותי ענן כאלה ולכל שירות ענן כזה יש מדריך מסוים מSAMPLECODE./node.js קוד Node.js.

בפרק הזה נראה את שתי השיטות.

עליה לפודקשן עם שירות

אם אתם מתכנתים PHP או מתוכנתים בשפות אחרות, העלייה לפודקשן היא פשוטה – להעלות קבצים אל השירות, וה-Apache או ה-Xhttpd יודעים לטפל בזה. אבל ב-Node.js אין לא יכולות לעשות את זה, אנחנו צריכים ממשם את השירות. אנחנו צריכים לדאוג שהוא כל הזמן יהיה באוויר. אם תרגלتم כמו שצריך, ראייתם שאם אתם לא מרכיבים את תחילה Node.js שארחrai להרים את השירות שלכם – השירות שלכם לא יעבד. אם יש תקלת – התהיליך יקרוס וצטרכו להרים אותו מחדש, או אם אתם עושים ריסט למחשב, צטרכו שוב להריץ אותו מחדש. זה בסדר כאשר לומדים ותרגלים, אבל פחות כאשר מדובר באתר שמשרת ללקוחות. אנחנו צריכים לדאוג לכך שהטהיליך יירץ כל הזמן. זהו האתגר האמיתי כאשר אנו פורשים קוד של Node.js בשרת שלנו.

גם כאשר השרת עבר רוסטרט וגם כאשר יש לנו תקלת מסויימת – התהיליך תמיד צריך להיות למעלה. תהשבו על גמד קטן שבכל פעם מרים `js/app.`. `node` כאשר האפליקציה נופלת. לגםד הקטן הזה יש שם: `2mk`. נחשו מה? זה מודול של `Node.js` שדווג שהקוד שלנו יהיה כל הזמן למעלה.

מתוקינים אותו על השרת באמצעות התקנה גלובלית של מודול.

`npm i pm2 -g`

אחרי התקינה, נקליד:

`pm2 start app.js`

האפליקציה תתרום לאויר. אם תעשו ריסטרט למונגה או תסגורו את הטרמינל, השרת עדיין יהיה באויר. אפשר להריץ כמה וכמה שירותים ולראות אותם באמצעות:

`pm2 list`

`2mk` גם שימושי מאד לסביבת פיתוח, ואפשר להתקין אותו על המחשב בכל מערכת הפעלה. אם תקליד:

`pm2 start app.js --watch`

ומשנו את הקוד שלכם, האפליקציה תיתען מחדש.
אם אתם רוצים לעזור סופית לאפליקציה פשוט תקלידו:

`pm2 stop app.js`

הDocumentation העשירה של `2mk` מסבירה היטב איך לעשות את כל הפעולות האלה. אם עברתם על כל פרקי הספר, לא צריכה להיות לכם בעיה להסתדר עם הדוקומנטציה זו. אפשר להגדיר שם כמה סביבות לכל אפליקציה, להגדיר משתני סביבה שונים ועוד דברים מעניינים. אם אתם רוצים להרים את אפליקציית `Node.js` שלכם על שרת משלכם, `2mk` זו הדרך.

עליה לפרויקטן בענן

לכל סביבת ענן יש הדרכים שלה להעלות אפליקציית Node.js – בהתאם לצרכים. בחברות גדולות ואפילו קטנות יש אדם שתפקידו הוא להעביר את הקוד לסביבת הענן. התפקיד של האדם זהה נקרא DevOps וזהו הלוחם של שתי מיללים: Operations ו-Developer. תפקידו לקשר בין הפיתוח לפרויקטן והוא גם מופקד על סביבת הענן והתחזוקה שלה. לא פשוט לתחזק סביבת ענן ונדרש לך ידע עמוק בכל פלטפורמה ופלטפורמה.

למרות זאת, יש סביבות ענן ידידותיות למפתחים בודדים. הפופולרית שבנה היא Heroku, שמספקת גם סביבה זולה, בעלות של חמישה דולרים בחודש, למפתחים בודדים. נלמד עליה בפרק זה.

על מנת לעבוד עם Heroku, יש ליצור שם חשבון ולהתקין על המחשב שלכם את Heroku toolbelt. מדובר ב-CLI פשוט שמאפשר לכם להקליד בטרמינל את הפקודה heroku ובאמצעותה לעשות פעולות שונות. מיד לאחר ההתקנה נקליד login heroku. נקליד את הסיסמה ונוצר מפתח התחברות. בעצם זה לחוץ על Yes, Yes, Yes ולספק פרטים בכל פעם שנדרש מאייתנו. אחרי שעשינו את זה. ניצור בתיקיית האב של הפרויקט שלנו קובץ שנקרא procfile – כן, ללא סויומת קובץ. הקובץ הזה מכיל את כל ההוראות לפירישה של הפרויקט שלנו. במקרה של כל הדוגמאות בספר זה יהיה משוחב בסגנון: `./app.js`. נפתח את קובץ ה-`procfile` ונכתבו פנימה:

```
web: node app.js
```

נשמר את הקובץ.

עכשו להעלאה. כדי להעלות את הפרויקט שלנו ל-Heroku, אנו נדרשים לעבוד עם מנהל גרסאות מסוג גיט. גיט וניהול גרסאות אינם נלמדים בספר זה אבל זה כלי חשוב מאוד למפתחים. נקליד Heroku create כשאנו בתיקייה הראשית של הפרויקט שלנו. ואחרי כן:

```
git push Heroku master
```

וזאת בהנחה שאתם רוצים לבצע deployment ל-`master`. הקפידו שככל הקבצים שלכם אכן יהיו בגרסה זו. הפעילו את האפליקציה באמצעות:

```
heroku ps:scale web=1
```

והויכנסו אליו באמצעות open heroku. שימו לב שכבר תועברו לנתיב ש-[heroku](#) יוצרה עבורכם. שימו לב שם יצרתם שרת ואתם רוצים לגרום לו לעבוד, הקפידו לא להכניס פורטים במספריים בשרת שלכם. במקומם:

```
app.listen(3000);
```

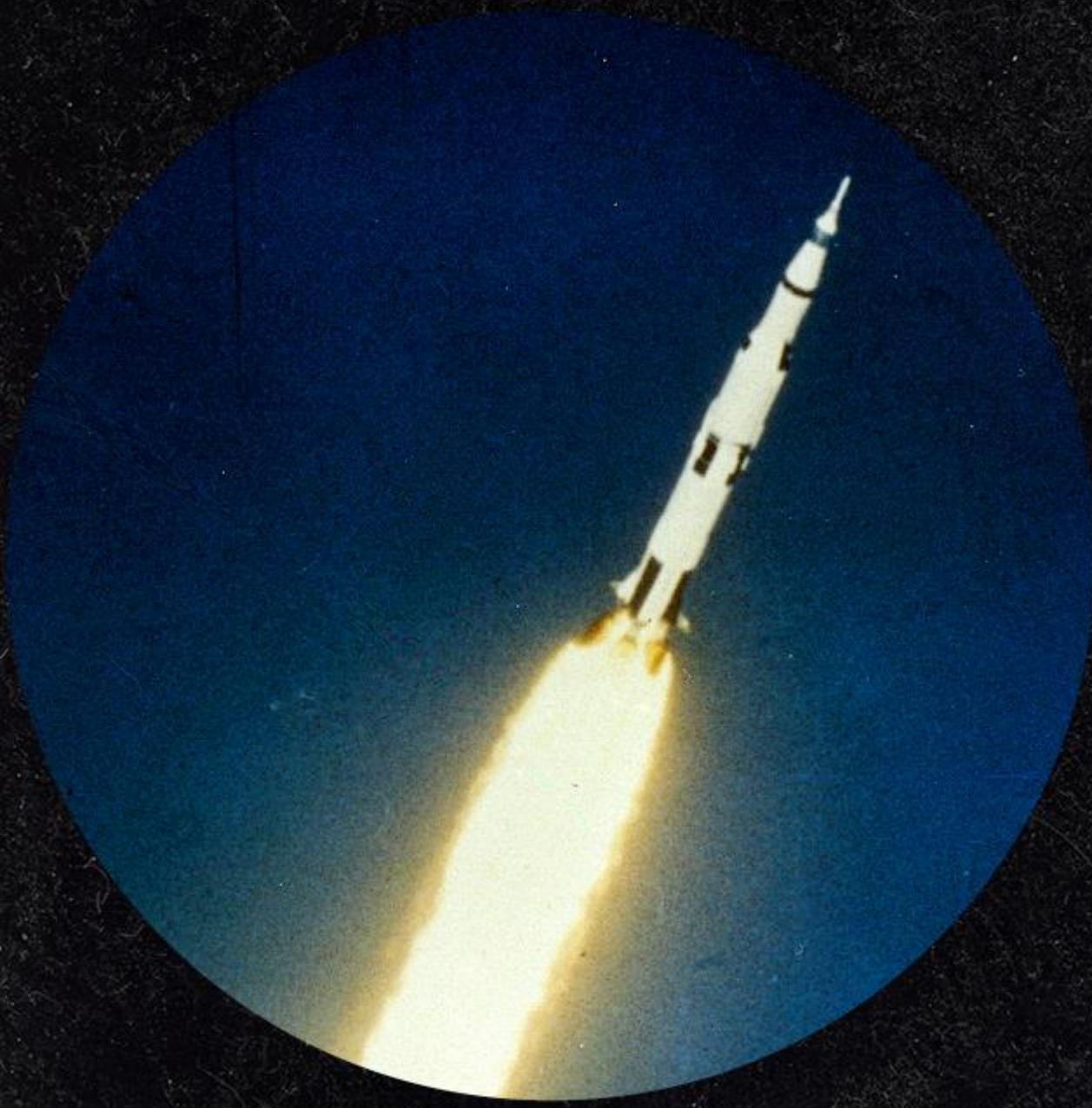
או מהו בסגנון, הקלידו:

```
const port = process.env.PORT || 3000;  
app.listen(port);
```

אחרת השירות שלכם לא יעבוד-[heroku](#) (או בכלל בסביבת ענן אחרת). כאמור, המדריך הזה הוא ברמת Hello World. לאפליקציה ממשית זה לא יספיק, כמובן, אבל זה יהיה מעולה על מנת להראות לכם איך האתר שלכם נראה באינטרנט האמיתי. בשביל אחר אמיתי – ובטח בשビル אתר מרובה משתמשים – תצטרכו לחקור יותר את הדוקומנטציה של Heroku ואולי גם להתייעץ עם איש DevOps. בספר המשך "לימוד פיתוח ווב מעשי", אני מלמד על עלייה לפורדקשן במגוון סביבות: שירותים פורטיים ושירותי ענן של אמזון.

פרק 20

סינום



סיכום

אם אתם יודעים ג'אווהסקריפט היטב, ועברתם על כל פרקי הספר זהה ותרגלתם את כל התרגילים – אתם כבר אמורים להבין היטב איך Node.js עובדת ואיך התוכניות שלך רצות. עכשו, כל מה שצרכי הוא ניסיון. לא ידע תיאורטי. שפת תכנות ופלטפורמות שונות לומדים תמיד דרך הידים – ניסיון מעשי. אפשר לצבור את הניסיון המעשית הזה בקלות גם בלי עבודה בחברה.

השאלה שתמיד שואלים היא – אין? אין צוברים ניסיון? למידה תיאורטיבזה טוב ויפה, אבל יש צורך גם בניסיון פרקטני. מהניסיון שלי, הקהילה בארץ מאד חמה ותומכת ואפשר לפגוש אותה בכם וכמה מקומות. האנשים שם ישמחו לסייע לכם להשתלב בתחום בהתאם ליכולות ולשאייפות שלכם.

כתבתי ספר בשם "פיתוח ווב מעשי" המסביר איך יוצרים פרויקטים, תורמים קוד ומפגינים את היכולות שלכם ואיך משלבים את היכולות האלה בקורות החיים. קריאה בספר יכולה מאד לסייע: כמו כן, ניתן לפגוש ולהעזר בגופים הבאים:

ミטאפים

הדרך הטובה ביותר לפגוש אנשים שעובדים עם Node.js ביוםום היא להגיע למיטהפים. יש בישראל לא מעט מיטהפים למתכנתים Node.js, שרובם מרכזים באתר [meetup.com](https://www.meetup.com). הגיעו למיטהפים, שמעו הרצאות מעניינות ודברו עם אנשים שם. הם יכולים לסייע לכם בנוגע לתקומות הלאה.

נוסף על מיטהפים יש האקתוונים. זו הזדמנות נדירה להצטרף לצוות של מתכנתים מנוסים יותר, ללמוד מהם ולבוד כיצד בפרויקטים אמיתיים.

קבוצות דין

בפייסבוק יש כמה וכמה קבוצות בעברית המוקדשות לג'אווהסקריפט ו-Node.js. הקהילה שם היא חמה ומקצועית. הצטרפו לקבוצות, קראו את הדיונים והשתתפו בהם. זה יועיל מאוד לדעת מה מקצועיים שלכם ויכונן אתכם לעבודות ולפרויקטים.

גייטהאב

בגייטהאב יש לא מעט פרויקטים של קוד פתוח מבוססי Node.js. חלק מהם כבר השתמשתם בספר הזה. נסו לראות אילו פרויקטים מבוססי Node.js יש שם ונסו לתרום להם או לבחון את הקוד שלהם.

התנדבות בעמותות ובمיזמים

לא מעט עמותות, מיזמים כמו נסנת פתוחה, למשל, וארגוני מחפשים מתכנתים בתחילת דרכם שיווכלו לתרום ולסייע. ההתנדבות במיזמים כאלה יכולה לסייע מאוד להתחיל לעסוק בקוד אמיתי לצד מתכנתים מנוסים יותר.

בין שבחרתם באחת הדרכים שלעיל או בדרך משלכם – חשוב מאוד לתרגל, לתרגל, לתרגל. שפה ופלטפורמה הן בדיקן כמו שפה בעולם האמיתי. אם לא תשתמשו ותתרגלו – תשחחו. ב-node.js אפשר לעשות המון דברים. אפשר ליצור תוכנות לשרתים, אבל יש גם פלטפורמות המאפשרות להשתמש/node.js על מנת ליצור אפליקציות לנידים, תוכנות למחשב ביתי ואפילו תוכנות ל-Desktop – האינטרנט של הדברים. בחרו אחד התחומיים שמושך אתכם ונסו ליצור פרויקט משלכם. יהיו קשיים, יהיו בעיות – לא יכול אפשר ללמוד מספר או מקורס, טובים ככל שיהיו. נתקלتم בבעיה? חפשו את השגיאה בגוגל ונסו לפתור אותה עצמאית, חפשו באתר StackOverflow או נסו להיעזר באחת מקבוצות המתכנתים בפייסבוק. מה שחשוב הוא לא להתייאש. Node.js באמת נמצאת בכל מקום, קל להפעיל אותה והיא יופי של פלטפורמה לכל מתכנת ג'אווהסקריפט. בהצלחה!

נספח: בדיקות אוטומטיות ב-Node.js

מאת דניאל ניב כהן – חברת Elementor

מה זה בדיקות אוטומטיות?

תתארו לעצמכם מגדל ג'נגה גבוה, ואתם יושבים עליו בזמן שבאופן אקראי נשלפים ממנו לבנים. מרגשימים את הפקד? אז זהה השגרה בארץ ה-Node.js. הקוד שלכם יושב מעל מגדל זהה הבניי ממודולים שימושניים כל הזמן, ובו בזמן אתם וחברי הצוות שלכם גם מכניםים קוד חדש שיכול להכיל באגמים חדשים או רgresיות בדברים שעבדו קודם.

כל אפליקציה Node.js תלויה לכל הפחות בגרסה של Node.js עליה היא רצתה, רוב האפליקציות משתמשות גם בספריות וקח רבות, וכך הן תלויות גם בגרסה של וקח עצמו. וכל אחת מהחבריות בהן האפליקציה משתמשת עלולה להיות תלואה בחבריות נוספת בסוף עצמה. רק כדי לסביר את האוזן, אפליקציית express המכילה מינימלית תלואה בכ-400 חברות תוכנה שונות.

از איך מייצבים את המגדל? איך בכל זאת בונים כלים ושירותים יציבים דוגמת Netflix ו-ebay, מהשירותים הייציבים והסיקילבליים בשוק.

כדי לענות על השאלה הזאת, בואו נסתכל על תוכנה לא קוד, אלא נאוסף של חזים אשר ממומשים באמצעות קוד. ככלمر, כשמיDEL חושף פונקציונליות כלשהי, הוא בעצם יוצר חזה שהפונקציונליות הזאת לא תשנה בעתיד.

לדוגמה, נניח שקיים 2 מודולים של מחשבון; `euclid` ו-`pythagoras`:

[pythagoras.js](#)

```
function subtract(a, b) {
  return a - b;
}
```

```
module.exports = { subtract }
```

[euclid.js](#)

```
function add({ x1, x2 }) {
  return x1 + x2;
```

```

}

function subtract({ x1, x2 }) {
  return x1 - x2;
}

module.exports = { add, subtract }

```

כרגע, ברור לנו לcolm שני המודולים אינם זהים, שכן בעוד שמודול pythagoras יכול רק לחסר מספרים, המודול euclid יכול הן לחסר והן לחבר. בעצם, נניח שמיימשתי ב-`pythagoras` גם יכולת חיבור, אז באמצעותה מימושי מחדש גם את יכול החישור כה:

pythagoras.js

```

function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return add(a, -b);
}

module.exports = { add, subtract }

```

אם הרגע הפכתי את `euclid` ל-`pythagoras`? ברור שלא. למורת שני המודולים כתעת בעלי יכולות זהות, `pythagoras` לא נהפך ל-`euclid` בגלל שיש הבדל בקלט בין מה ש-`euclid` מבין מה ש-`pythagoras` מצפים לקבל; בעוד ש-`euclid` מצפה לקבל 2 ארגומנטים (`a` ו-`b`), המודול `euclid` מצפה לקבל אובייקט עם 2 מאפיינים (`x1` ו-`x2`); יש להם חוזים שונים!

שאלה נוספת טובה היא: האם המודול `pythagoras` עדין נשאר `pythagoras`? או שעכשיו הוא כבר משהו שלישי – חדש?

אם נגידיר את תוכנה zusätzlich של חוזים ממומשים, נוכל להוכיח שהיות והגרסה החדשה של `pythagoras` עדין מમמשת את החוצה שהוגדר בגרסה המקורית שלו (`subtract(a, b)`) – זה עדין אותו המודול.

אבל רגע! האם אכן הגרסה החדשה של pythagoras עדין ממחשת את החוזה הקודם? כדי שנוכל לקבוע בוודאות שהגרסה החדשה מכבדת את החוזה הישן, אנחנו צריכים דרך להוכיח שעובד כל קלט, הגרסה החדשה תחזיר את אותו הפלט כמו הגרסה הישנה.

באו נבדוק כמה דוגמאות של סוגי קלט ופלט על הגרסה המקורית:

pythagoras.use.js

```
const { subtract } = require('./pythagoras'); // @1.0.0

// 1. it should return the result of subtraction of two numbers
console.log(subtract(3, 2)); // Outputs: 1

// 2. it should support string values
console.log(subtract('3', '2'));
```

אוקי, אז ראיינו שהגרסה המקורית מחסרת בהצלחה ערכים מסוג number או מסוג string ומחזירה תוצאה נכונה - זהו החוזה של המודול.

כעת, נzin את אותו הקלט לגרסה החדשה ונווידא שהחוזה עדין מתקיים:

pythagoras.use.js

```
const { add, subtract } = require('./pythagoras'); // @2.0.0

// 1. it should return the result of subtraction of two numbers
console.log(subtract(3, 2)); // Outputs: 1

// 2. it should support string values
console.log(subtract('3', '2'));
```



```
// 3. it should return the result of addition of two numbers
console.log(add(1, 2)); // Outputs: 3
```

רגע, מה? למה החישוב $2-3=3-2$? ציפינו לקבל 1, החוזה נשבר!

בגרסה הראשונה המתוודה subtract ת麥ה בארגומנטים מהסוגים number ו-string. ובגרסה השנייה זיהינו רגרסיה - חוזה שהוא מכובד בגרסה הקודמת, הופר בגרסה החדשה. מה שגרם לרגרסיה הוא שימושנו מחדש של subtract את add באמצעות ולבסוף, שינוינו את האופרטור בו השתמשנו, מאופרטור חיסור (-) שאוטומטית מנסה להמיר מחרוזות למספרים, לאופרטור חיבור (+) שבמקרה זה מבצע שרשור מחרוזות, וכך איבדנו את התמייה במחרוזות בפועלות החיסור.

למעשה, הבדיקות שהרגע הרצינו, היו בדיקות ייחודיה, ובעזרתן גילינו שההמיושח החדש לא עונה יותר על החוצה שהגדרנו, או במילויים אחרים - מצאנו בכך!

בדיקות אוטומטיות

כל מה שחרס לנו על מנת להבטיח את המשך קיום החוצה של המודול הוא לוודא שאנו חנו מרכיבים את סט הבדיקות זהה שוב אחרי כל עדכון של המודול. וכך ניש לנו עדיין שני אתגרים:

1. אמנם כתבנו בקוד את הבדיקות, אך הווידוא של תוצאות הבדיקה הוא עדיין דני.
2. אין לנו דרך נוחה להרייך את סט הבדיקות.

על מנת להתמודד על האתגר הראשון נגדיר את התוצאה הרצויה, ובמידה ומקבלת תוצאה שונה נזורך שגיאה:

pythagoras.err.js

```
const { add, subtract } = require('./pythagoras');

// ...

// 2. it should support string values
const actual = subtract('3', '2');
const expected = 1;

if (actual !== expected) {
  throw new Error(
    `contract 'it should support string values' failed: ${expected} but actual is ${actual}`
  );
}

// ...
```

וכדי שנוכל להרייך את הבדיקות בקלות, נוסיף סקורייפט test בקובץ package.json ככך:

package.json

```
{
  "name": "pythagoras",
```

```

"version": "2.0.0",
// ...
"scripts": {
  "test": "node pythagoras.err.js"
},
// ...
}

```

עכשו נוכל הרץ `test run npm` ולדעת מיד האם כל הבדיקות עברו או לא:

Error: contract 'it should support string values' failed:expected '1' but actual is '3-2'

והנה, יש לנו בדיקות אוטומטיות על הקוד שלנו!

אם כתעת אתם חושבים לעצמכם, רגע... שבשביל לבדוק 3 שורות קוד, הרגע כתבתי 8 שורות, אתם לגמרי צודקים. ישנוום כלים שמקזרים ומייעלים בהרבה את כתיבת והרצה הבדיקות האוטומטיות. הם מתחולקים בעיקר לשתי קבוצות:

1. **MRIICI BDIKOT - Test Runners** - Jest Mocha דוגמת Assertion libraries שמאפשרים לבטא בקלות את תנאי הבדיקה, ומיצרים תיאור שגיאה אינפורטטיבי במקרה של כשלון.
2. **PRIMORAK BDIKOT - Primary Assertion libraries** דוגמת assert-Chai Jasmine. אונחנו נתמך ב-Mocha, אבל אין הבדלים משמעותיים באופן כתיבת הבדיקות לספריות האחרות.

Mocha

על מנת שכטיבה והרצה של בדיקות תהיה נוחה ועקבית, עם הזמן נוצרו "MRIICI BDIKOT" שונים, המוכרים ביניהם הם Jasmine, Jest Mocha. אנחנו נתמקד ב-Mocha, אבל אין הבדלים משמעותיים באופן כתיבת הבדיקות לספריות האחרות.

התפקיד של MRIICI הבדיקות הוא לאתר קבצי בדיקות בפרויקט, להריץ אותם, להחליט אילו בדיקות עברו ואילו נכשלו, ולדוח את התוצאות במגוון דרכים - לדוגמה הצגה של התוצאות בקונסול או שמירתם לקובץ.

לצורך הוספה של Mocha לפרויקט, נריץ בשורת הפקודה:

`npm install --save-dev mocha`

ונשנה את הסקרופט `test` שב-`package.json` כך שישתמש ב-`Mocha` להרצת הבדיקות:

package.json

```
{
  "name": "pythagoras",
  "version": "2.0.0",
  // ...
  "scripts": {
    "test": "./node_modules/.bin/mocha *.test.js"
  },
  "devDependencies": {
    "mocha": "^6.2.0"
  }
  // ...
}
```

הערה:

לצד ההוספה לפרויקט, ניתן גם להתקין את `mocha` בצורה שתיהיה זמינה להריצה מכל מקום במחשב ע"י הפקטרט-ג:-

`npm install -g mocha`

חיסרון בדרך זו, הוא - להיות ומדובר בהתקנה גלובלית, כל הפרויקטים על המחשב המדובר יישתמשו באותה הגרסה של הספריה, ולפעמים זו לא התוצאה הרצויה.

כעת שהוספנו את `mocha` לפרויקט, בוואנו נחוץ ל-`pythagoras`, ונכתב את הטסטים מחדש :Mocha

pythagoras.test.js

```
const { add, subtract } = require('./pythagoras');
const assert = require('assert');

describe('pythagoras', () => {
  describe('subtract()', () => {
    it('should return the result of subtraction of two numbers', () => {
      const actual = subtract(3, 2);
      assert.strictEqual(actual, 1);
    });
  });

  it('should support string values', () => {
```

```

    const actual = subtract('3', '2');
    assert.strictEqual(actual, 1);
  });

describe('add()', () => {
  it('should return the result of addition of two numbers', () => {
    const actual = add(1, 2);
    assert.strictEqual(actual, 3);
  });
});
});

```

ואם נריץ את הטעות:

\$ npm run test

נקבל את הפלט הבא:

> pythagoras@2.0.0 test

...

> mocha *.test.js

```

pythagoras
  subtract()
    ✓ should return the result of subtraction of two numbers
  1) should support string values
  add()
    ✓ should return the result of addition of two numbers

```

2 passing (16ms)

1 failing

1) pythagoras

subtract()

should support string values:

AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:

```
'3-2' !== 1
```

...

למעשה, הבדיקה עם דומה מאוד לקוד הבדיקה הראשון שכתבנו, רק שעכשיו mocha מאפשרת לנו לארגן את הבדיקות באמצעות מethodות describe ו-it (עליה נרחיב בהמשך), ומדפסה אוטומטית את הפרש בין התוצאה הרצוייה לתוצאה בפועל. והחבילת assert מאפשרת לנו לבטא בקלות את התנאים להצלחת הבדיקה, בלי הצורך בכתיבה משפטית זו ארוכים.

עכשו שראינו איך זה נראה מלמעלה,בואו נתייחס לחלקים השונים המרכיבים את הבדיקה יותר לעומק.

describe

כפי שראינו בדוגמה מלמעלה, בлок describe מגדר נושא שאותו אנחנו יכולים לבדוק, אבל לא מכיל את קוד הבדיקות עצמו. ניתןukan מה בлокי describe אחד בתוך השני. לדוגמה, כשאנו רוצים לכתוב בדיקות על מетодה ()add של מודול pythagoras אנחנו יכולים לבטא את זה כבלוק בשם 'add' מוקון בתוך בלוק בשם:'pythagoras'

```
describe('pythagoras', () => {
  describe('add()', () => {
    ...
  });
});
```

הפרמטר הראשון title מקבל את התיאור המילולי של הבלוק, והפרמטר השני fn מכיל את תוכן הבלוק בצורה של מתודה אנונימית.

ניתן להשתמש בבלוק describe לא רק כדי לבדוק בדיקות על חלק טכני של הקוד (כמו מודול או מетодה), אלא גם לפי כל נושא העולה על הדעת. mocha תשמש בהיררכיה של בלוקי describe לארגן התוצאות, דבר שמקל מאוד על הבנת הדוחות.

it

כל קובץ בדיקות של mocha חייב להכיל בлок `it` אחד לפחות. בлок `it` מכיל את קוד הבדיקה עצמו והוא יימצא בדרך כלל מוקון בתוך בлок `describe`, אך מבחינה טכנית זה לא נדרש.

הפרמטר הראשון `title` מקבל את התיאור המילולי של הבדיקה, והפרמטר השני `fn` מכיל מתודה אונומית שהיא הבדיקה עצמה.

החוקיות לפיה נקבעת הצלחת הבדיקה פשוטה מאוד - אם הקוד זורק שגיאה, הטעט נחשב לכשל, אחרת - הצלחה. לדוגמה, הבדיקה הבאה תחשב להצלחה למורות שהיא ריקה מקוד בכלל שהמתודה אונומית שמסרנו לו לא זורקת שגיאה.

0-empty.test.js

```
describe('basic usage', () => {
  it('will succeed', () => {
    // nothing here...
  });
});
```

ואכן אם נריץ את הבדיקה, נקבל:
 basic usage
 ✓ will succeed

1 passing (13ms)

לעומת זאת אם נריץ:

1-throw-error.test.js

```
describe('basic usage', () => {
  it('will fail', () => {
    throw new Error('custom error');
  });
});
```

נקבל:

basic usage

1) will fail

1 failing

1) basic usage

will fail:

Error: custom error

למעשה, לא מומלץ לזרוק שגיאות ידנית. בדרך כלל נשתמש בפרימורק בדיקות דוגמת `assert` או `chai` לצורך כתיבת הבדיקה, ואלו מתחורי הקלעים יזרקו שגיאה מסווג `AssertionError` במקרה שבדיקה נכשלה. בין היתר נשתמש במתודות בסיסיות של `assert` לצורך הדוגמאות, ונפרט על זה יותר בהמשך.

כל:

2-basic-assert.test.js

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

describe('basic usage', () => {
  it('true assertion', () => {
    assert.strictEqual(1 + 1, 2);
  });

  it('false assertion', () => {
    assert.strictEqual(1 + 1, 42);
  });
})
```

הבדיקה 'true assertion' תעביר, כי ההוצאה $1+1=2$ נכונה, מה שקשה לומר על $1+1=42$ ולכן הבדיקה 'false assertion' תכשל.

מחזור חיים

לפעמים בדיקה תדרוש פעולות הכנה לפניה ופעולות ניקוי אחרת. לדוגמה נניח שאנו בודקים מודול שמבצע שאלות על מסד נתונים. ניתן שאנו נרצה להביא את מסד הנתונים במצב ידוע לפני הבדיקה, ולהחזירו במצבו ההתחלתי אחריו.

בשביל זה יש ב-mocha את המתודות `before`, `after`, `beforeEach`, `afterEach`. המתודות האלה נקראות `hooks` – ווים כי הם מושרים לאיירועים בזמן ריצת הבדיקות.

כל המethodות האלה מקבלות רק ארגומנט אחד - fn שזה מתודה אנונימית שמכילה את הפעולות שיש לבצע בכל שלב.

לדוגמה:

```
before(( ) => {
  // code..
});
```

בדרך כלל hook מופיע בתוך בлок describe והוא חל על כל הבדיקות שבו. אם קיימים כמה בלוקים describe אחד בתוך השני, כל hook חל על הבלוק שבו הוא מופיע ועל כל הבלוקים שבתוכו.

הmethodות before ו-after רצות לפניו תחילת ביצוע הבדיקה הראשונה בבלוק, ולאחר מכן סיום ביצוע הבדיקה האخונה בבלוק עליון חלות - בהתאם.

הmethodות afterEach ו-beforeEach רצות לפניו ואחרי ביצוע של כל בדיקה ובדיקה בבלוק עליון חלות - בהתאם.

הmethodה before רצתה לפני methodה beforeEach והmethodה after רצתה אחרי methodה afterEach.

במקרה שיש hook-ים בבלוקים מקוונים, methodות beforeEach ו-beforeEach רצות של בלוק חיצוני יותר רצות לפניו methodות הנ"ל של בלוק פנימי, ומmethodות after ו-afterEach של בלוק חיצוני יותר רצות אחרי methodות הנ"ל של בלוק פנימי.

הערה:

ניתן לנתח hook גם בגוף קובץ הבדיקות, מחוץ לכל ה-block describe, וזה הוא נקרא גלובלי וחול על כל הבדיקות בחבילה - לא רק בקובץ הבדיקות שבו הוא מופיע.

נדגים את היכולות הללו:

3-hooks.test

```
before(( ) => {
  console.log('global before');
```

```

});
```

```

beforeEach((() => {
  console.log('global beforeEach');
});
```

```

describe('describe block', () => {
  before((() => {
    console.log('block before');
  }));

```

```

  beforeEach((() => {
    console.log('block beforeEach');
  });

```

```

  it('first test', () => { });

```

```

  it('second test', () => { });

```

```

  afterEach((() => {
    console.log('block afterEach');
  });

```

```

  after((() => {
    console.log('block after');
  }));
});
```

```

afterEach((() => {
  console.log('global afterEach');
});
```

```

after((() => {
  console.log('global after');
});
```

הפלט שיתקבל הוא:

global before

 describe block

 block before

 global beforeEach

 block beforeEach

 ✓ the test

```

block afterEach
global afterEach
block after
global after
global beforeEach
block beforeEach
  ✓ first test
block afterEach
global afterEach
global beforeEach
block beforeEach
  ✓ second test
block afterEach
global afterEach
block after
global after

```

מבנה בדיקה

מקובל לחלק כל בדיקה ל-3 חלקים:

- החלק הראשון - **Arrange**. בחלק זהה אנחנו נכין את כל נדרש לצורך ביצוע הבדיקה ונביא את המודול שבבדיקה למצב הרצוי.
- החלק השני - **Act**. זה החל שבו בפועל קוראים לפונקציה שבבדיקה עם כל הארגומנטים שהכנו בשלב הראשון.
- החלק השלישי - **Assert**. פה נמצאת הבדיקה עצמה. בחלק זהה בודקים את הפלט של הפונקציה שבבדיקה, ואם רלוונטי, גם את מצב התוכנה אחרי הביצוע.

בדיקות במבנה זהה נקראות בדיקות **Triple A** או **AAA**.
במקרים רבים החלקים הללו מסומנים בתוך גוף הבדיקה בהערות.

4-triple-a.test.js

```
it('is a well structured test', () => {
  // Arrange
```

```

const text = 'Elementor';

// Act
const actual = text.length;

// Assert
assert.strictEqual(actual, 9);
});

```

פרימורק בדיקות

כדי שהטסטים יהיו יעילים הם צריכים לוודא שלאחר ביצוע הפעולה הנבדקת, התוצאה או המצב זהה למה שאנו מכפים לקבל. כמובן שטכנית אפשר לכתוב את תנאי הבדיקה במבנה כמו שעשינו בתחילת הפרק:

```
if (!<condition>) throw new Error ('<error message>')
```

ישנה דרך נוחה ואלגנטית יותר לוודא תוצאות הבדיקה באמצעות פרימורק בדיקות, במקרה שלנו המודול `assert` של `Node.js`

המודול `assert` מכיל רשימה של מתודות כשל אחת מהן מיועדת לבדיקת תסרים נפוץ מסויים, ובמקרה של כישלון היא יודעת להפיק הודעה שגיאה מפורטת שיעזרת להבין מה בדיקת השتبש. לדוגמה נשתמש במתודה `:strictEqual`

```

it('will fail', () => {
  const actual = 'foo';
  const expected = 'bar';

  assert.strictEqual(actual, expected);
});

```

הבדיקה תיכשל עם השגיאה:

```

AssertionError [ERR_ASSERTION]: 'foo' === 'bar'
+ expected - actual
-foo

```

```
+bar
```

השגיאה מפרטת מה היה הערך הרצוי, ומה הייתה התוצאה בפועל, ומה ההבדל ביניהם. כמו כן שזו דוגמה בסיסית ביותר, ו-`assert` מכיל מתודות נוספות לבדיקות מורכבות בהרבה. להלן כמה מהן:

`assert.ok(value)`

אנו שימושי כדי לבדוק אמינות (truthiness) של ערך כלשהו והטסט יעבור רק אם הערך הוא `true` ויכשל אם הוא `false`. דוגמה:

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

describe('assert.ok()', () => {
  it('will pass', () => {
    const actual = 'truthy';
    assert.ok(actual);
  });

  it('will fail', () => {
    const actual = null; // falsy
    assert.ok(actual);
  });
});
```

הטסט הראשון יעבור, והטסט השני יכשל עם ההודעה:

`AssertionError [ERR_ASSERTION]: The expression evaluated to a falsy value:`

```
assert.ok(actual)
```

`assert.notStrictEqual(actual, expected)` | `assert.strictEqual(actual, expected)`

כדי להשוות ערך רצוי לזה שהתקבל בפועל, אפשר להשתמש ב-`strictEqual`, שייכשל במקרה שהערכים לא זהים. מנגד, `notStrictEqual` יכשל דווקא במקרה שהערכים זהים. דוגמה:

```
assert.strictEqual('foo', 'foo'); // PASS
assert.strictEqual('foo', 'bar'); // FAIL
```

ומנגד

```
assert.notStrictEqual('foo', 'foo'); // FAIL
assert.notStrictEqual('foo', 'bar'); // PASS
```

חשוב לציין שבשני המקרים ההשוואה תבוצע באמצעות האופרטור `==` כלומר ללא המורות. וכך הבדיקה הבאה תיכשל:

```
assert.strictEqual(1, '1'); // FAIL
```

ולכן גם השוואת 2 אובייקטים זהים תיכשל, בגלל שאופרטור `==` משווה אובייקטים לפי המצביע. השגיאה גם תוכל הودעה על כך שההשוואה נכשלה בגלל השוני במבנה ולא בגלל שוני בין האובייקטים עצמם:

```
assert.strictEqual({ name: 'foo' }, { name: 'foo' }); // FAIL
// (Values have same structure but are not reference-equal)
```

assert.notDeepStrictEqual(actual, expected) assert.deepStrictEqual(actual, expected)

במקרים בהם נדרש לוודא את שוויון הערבים בשני אובייקטים יש את המתודה `deepStrictEqual` המבצעת השוואת של ערכים בין שני אובייקטים בצורה רקורסיבית. ההשוואה בין זוג של ערכים שאינם אובייקט או מערך עדין מתבצעת באמצעות האופרטור `==`. הפעם עם השוואת deepStrictEqual השוואת 2 אובייקטים זהים תעבור:

```
assert.deepStrictEqual({ name: 'foo' }, { name: 'foo' }); // PASS
```

במקרה של אובייקטים שונים השגיאה תכלול פרטיים על ההבדל בין האובייקטים:

```
assert.deepStrictEqual({ name: 'foo' }, { name: 'bar' }); // FAIL
```

+ actual - expected

```
{
+ name: 'foo'
- name: 'bar'
```

. notDeepStrictEqual עבד בצורה זהה רק שנכשל דוקא במקרה שהאובייקטים זהים.

assertthrows(fn)

מה אם אנחנו רוצים לוודא שמתודה שלנו זורקת שגיאה, לדוגמה במקרה בו קוראים לה עם ארגומנט חסר?

```
function greet(name) {
  if (typeof name !== 'string') {
    throw new Error('name should be a string');
  }

  return `Hello ${name}`;
}

it('will pass', () => {
  assertthrows(() => greet(null)); // PASS
});
```

זה בדוק מה שהמתודה throws עשויה. היא מקבלת כารוגמנט פונקציה שאמורה לזרוק שגיאה, והבديקה עוברת במקרה שאכן נזרקת שגיאה.

ספריות נוספת

כמו בכל נושא אחר ב-node.js, גם בתחום פרימוורק בדיקות יש מבחן של אפשרויות שונות. נראה הראויו ביותר לציין היא ספריית `chai`. ב-`chai` יש מבחן עשיר יותר של סוגים בדיקות כמו כן הספרייה תומכת בשלוש סגנונות של כתיבת קוד, ויש עבורה עולם של תוספים. לדוגמה בסגנון כתיבה `expect`, בדיקות נראים כך:

```
const { expect } = require('chai');

const foo = 'bar';
const beverages = { tea: ['chai', 'matcha', 'oolong'] };

expect(foo).to.be.a('string');
expect(foo).to.equal('bar');
expect(foo).to.have.lengthOf(3);
expect(beverages).to.have.property('tea').with.lengthOf(3);
```

ספריות mock

עד כה התייחסנו בעיקר לבדיקות של פונקציות טהורות - פונקציות שאין להן השפעות חיצונית (side effects) ולא תלויות חיצונית.

העולם האמיתי לעומת זאת מלא בפונקציות שתלויות אחת בשניה, וגם בפונקציות או שירותים חיצוניים שונים.

להמחשה, נסה לבדוק פונקציה שמקבלת נתיב לקובץ, ומחזירה את מספר השורות באותו הקובץ:

line-count.js

```
const { promises: fs } = require('fs');

async function getLineCount(path) {
  const str = await fs.readFile(path, { encoding: 'uft-8' });
  return str.split('\n').length;
}

module.exports = { getLineCount }
```

הבעיה היא שהפונקציה `getLineCount` קוראת ל-`fs.readFile`, פונקציה חיצונית, בנוסף צריכה גם קובץ פיזי כדי לעבוד. איך נכתוב בדיקה במקרה מקרה?

התשובה היא mocking, או חיקוי. ספריית `mock` מאפשר ליצור חיקויים למתודות אמיתיות לצורך בדיקות. באkosיסטם של Node.js הספרייה המוכרת ביותר בתחום היא `sinon`.

נתקין את `sinon` משורת הפקודה:

```
npm install --save-dev sinon
```

כעת, בעזרה `sinon` נוכל לבדוק את הפונקציה שלנו בצורה הבאה:

0-intro.test.js

```
const sinon = require('sinon');
const assert = require('assert');

const sandbox = sinon.createSandbox();

afterEach(() => {
  sandbox.restore();
});
```

```

describe('stub demo', () => {
  it('should return correct line count', async () => {
    // arrange
    const { promises: fs } = require('fs');
    const { getLineCount } = require('./line-count');

    sandbox.stub(fs, 'readFile').resolves('line1\nline2')

    // act
    const actual = await getLineCount(null);

    // assert
    assert.strictEqual(actual, 2);
  });
});

```

בבדיקה, אנחנו משתמשים במתודה stub של `sinon` כדי "לדחוס" את המתודה `readFile` של `Node.js` באמצעות מימוש דמה שלנו שתמיד מחזיר את הערך `'line1\nline2'`.

המתודה stub מקבלת 2 ארגומנטים, הראשון הוא האובייקט המכיל את המתודה אותה אנחנו רוצים להחליף בדמה (stub), והARGOMENT השני מכיל את שם המתודה. המתודה stub מחזירה אובייקט stub שעליו קיימת (בין היתר) המתודה `resolves` שבuzzurtha ניתן להגדיר מה stub יחזיר כשיופעל. שימו לב, שאנחנו קוראים ל-`resolves` כי המתודה `readFile` אמורה להחזיר `Promise`.

בהמשך אנחנו מרים את הפונקציה `getLineCount` עם `null` במקום נתיב הקובץ, ובודקים שהחטאה היא אכן 2 שורות, בדיק נפי שציפינו.

בראש הבדיקה יוצרים `sandbox` (עליו נרჩיב בהמשך) ו-`beforeEach` אנחנו מאפסים את `sandbox` כדי שהבדיקות מבדיקה אחת לא ישפיעו על הבדיקות הבאות אחריה. קריאה ל-`restore` על `sandbox` מבטלת את כל ה"דרישות" שעשינו באותו ה-`sandbox` ע"י `mock` או `stub`.

עכשו נניח שאנחנו רוצה לוודא שהמתודה `getLineCount` אכן משתמש בקידוד utf-8 בזמן קריאת הקובץ ל-`string`. כאן באה לעזרתינו הינולת השניה של `stub`, והוא ש-`stub` גם משמש כ-`spy`.

[source](#)

```

describe('spy demo', () => {
  it('should use utf-8 for reading the file', async () => {
    // arrange
    const { promises: fs } = require('fs');
    const { getLineCount } = require('./line-count');

    const stub = sandbox.stub(fs, 'readFile').resolves('line1\nline2')

    // act
    await getLineCount(null);

    // assert
    stub.calledOnceWithExactly(null, 'utf-8');
  });
});

```

הפעם אנחנו שומרים את ה-stub stub שיצרנו במשתנה stub על מנת שנוכל לתחקור אותו בסוף הבדיקה ע"י stub.calledOnceWithExactly(null, 'utf-8'); שמדובר באCTION שנקרא בדיק פעם אחת, עם בדיקת הפרמטרים null ו-'utf-8'.

הmethod spy זמינה עקב העובדה שב-sinon כל stub הוא גם spy. כתעת נפרט קצת יותר על היכולות השונות של spies, ועל השימוש בהם.

spy
האפשרות "לרגל" אחריו קריאות למethodות. בעזרת spy נוכל לדעת כמה פעמים נקראה method מסוימת, ואילו ערכיהם הועברו לה כארוגמנט.

spy מחזיר פונקציה שאפשר לקרוא לה כמו כל פונקציה אחרת, אך בנוסף לזה, יש לה מבחן methods שמאפשרת לתשאל את ה-spy על האופן שבו הוא נקרא. להלן כמה דוגמאות:

1-spies.js

```

const func = (x) => x * 2;
const spy = sinon.spy(func);

[1, 2, 3].map(x => spy(x));

console.log(spy.callCount); // Outputs: 3
console.log(spy.firstCall.args[0]); // Outputs: 1

```

```
console.log(spy.firstCall.returnValue); // Outputs: 2
console.log(spy.args[2][0]); // Outputs: 3
```

fake

בעזרת `fake` ניתן ליצור מתודה "דמה" בעלת התנהגות מוגדרת, ולקבוע מה תחזיר מתודה הדמה עם אחת מהפונקציות `rejects`, `resolves`, `throws`.

דוגמא:

2-fakes.js

```
const fake = sinon.fake.returns(42);

console.log(fake()); // Outputs: 42
console.log(fake.callCount); // Outputs: 1
console.log(fake.firstCall.returnValue); // Outputs: 42
```

stub

המתודה `stub` מאפשרת להציג ערכים שונים בהתאם לערכי הארגומנטים ולמספר הפעם שה-`stub` נקרא. בנוסף ל-`stub` יש את כל היכולות של `fake`.

3-stubs.js

```
const stub = sinon.stub();

stub.withArgs('foo').returns('bar');
stub.withArgs('heyyy').returns('hoooo');

console.log(stub()); // Outputs: undefined
console.log(stub('foo')); // Outputs: 'bar'
console.log(stub('heyyy'))); // Outputs: 'hoooo'
```

בנוסף ליכולות האלו, `stub` מאפשר לדرس מתודה בתוך אובייקט, דבר שמאוד שימושי בדיספה של תלויות ממודולים חיצוניים.

3-stubs.js

```
const obj = {
  greet: () => 'hello!'
};

const stub = sinon.stub(obj, 'greet');
```

```
stub.onFirstCall().returns('hi!');
stub.onSecondCall().returns('howdy!');

console.log(obj.greet()); // Outputs: 'hi!'
console.log(obj.greet()); // Outputs: 'howdy!'
```

mock(obj)

בשונה מכל היכולות שכבר פירטנו mock מיועד יותר לכתיבת וידוא, ולא רק לדרישות אובייקטים. כשלורדים או עוטפים אובייקט ב-`mock` ניתן להגדיר את הציפיות ממנה באמצעות `expects`, ולאחר מכן לוודא שכל הציפיות אכן התקיימו באמצעות `verify`. בנוסף, ל-`mock` יש את כל היכולות של `stub`.

4-mocks.js

```
const obj = {
  greet: () => 'hello!'
};

const mock = sinon.mock(obj);

mock.expects('greet').once().returns('hi!');

console.log(obj.greet()); // Outputs: 'hi!'

mock.verify(); // PASS
```

createSandbox

היות `sinon` דורס את המימוש המקורי של פונקציות ואובייקטים, נדרש להיות דרך להחזיר את האובייקט במצב המקורי בסיום הבדיקה כדי שינויי מבדיקה אחת לא ישפיעו על הבדיקות האחרות.

מסיבה זאת, מומלץ תמיד שלא להשתמש ב-`sinon`'s יישירות אלא קודם ליצור `sandbox` (ארגז חול) ע"י קראיה ל-`createSandbox`, ובסיום כל בדיקה (למשל ב-`afterEach`) לקרוא ל-`restore` כדי להחזיר הכל לסדר.

דוגמא:

5-sandbox.js

```
const sandbox = sinon.createSandbox();
```

```

const obj = {
  func: () => 1,
};

sandbox.stub(obj, 'func').returns(42);

console.log(obj.func()); // Outputs: 42

sandbox.restore();

console.log(obj.func()); // Outputs: 1

```

בדיקות עם קריאות http

מה אם המתודה שאנו רוצים לבדוק פונה לבקשת לקבלת מידע ולביצוע פעולות? כמובן שנייתן לדروس מוגדרות במודול `http` לצורך הבדיקות, הבעה היא שמאוד קשה "לזייף" שירות `http` אמיתי בגל המורכבות של הפרטוקול.

בשביל זה קיימת ספרייה בשם `nock`. עם `nock` ניתן לתפוס שאלות `http` לפני שהן יוצאות לשרת, ולהציג תשובה שנראית כאילו היא באה משרת אמיתי.

נתקין את `:nock`

`npm install --save-dev nock`

עכשו, נניח שיש לנו מוגדרת שמחזירה את מספר הכוכבים שיש לריפו של Elementor ב-GitHub:

github.js

```

const fetch = require('node-fetch');

async function getElementorRepoStarCount() {
  const response = await fetch('https://api.github.com/repos/elementor/elementor');

  if (!response.ok)
    throw new Error(`HTTP status ${response.status}`);
}

const json = await response.json();

return json.stargazers_count;
}

```

```
module.exports = { getElementorRepoStarCount }
```

ואנו רוצים לבדוק 2 תסרים; בהינתן ש-github מgive עם קוד 200, המתוודה שמתקיים רצויים תחזיר את הערך של stargazers_count שהוחזר, ובמקרה שמתקיים סטטוס שגיאת - המתוודה זורקת שגיאה.

עם nock נוכל לבדוק את זה כך:

6-nock.test.js

```
const nock = require('nock');
const assert = require('assert');

describe('nock demo', () => {

  afterEach(() => {
    nock.cleanAll();
  });

  it('should return the number of stars from github', async () => {
    // Arrange
    const { getElementorRepoStarCount } = require('./github');

    const expected = 123456;

    nock('https://api.github.com')
      .get('/repos/elementor/elementor')
      .reply(200, {
        stargazers_count: expected
      });

    // Act
    const actual = await getElementorRepoStarCount();

    // Assert
    assert.strictEqual(actual, expected);
  });

  it('should throw an error on status other than 2xx', async () => {
    // Arrange
    const { getElementorRepoStarCount } = require('./github');

    nock('https://api.github.com')
```

```

    .get('/repos/elementor/elementor')
    .reply(408); // Request Timeout

    // Act
    const actual = getElementorRepoStarCount();

    // Assert
    await assert.rejects(actual);
  });
}
)

```

לצורך ביצוע שאלות אינטראקטיביות בדוגמה השתמשנו בספרייה `node-fetch`. חשוב לציין ש-`nock` יעבד עם כל ספרייה `http` אחרת, כי בפועל הוא מתלבש על מודול `http` הסטנדרטי של `Node.js`.

כמו ב-`nock`, גם ב-`nock` חשוב להזכיר את המצב לקדמונו בסיום כל בדיקה ע"י `All`
ב-`each`. אך בניגוד ל-`nocks`, ב-`nock` כל זיופ מחזיק רק לקריאה אחת. לאחר קריאה אחת
לכתובת שעליה עשינו `nock`, הזיופ נבטול והקריאה הבאה תבוצע לכתובת הרשות האמיתית. את
הקריאה ל-`All` `clear` כדאי לעשות במקרה שבו שבדיקה תכשל לפני שהיא תספיק לבצע את
השאליטה, וכן ה-`nock` ישאר פעיל לבדיקה הבאה.

סוגי בדיקות

כשאנחנו מדברים על בדיקות תוכנה אוטומטיות, אנחנו מדברים על הרבה סוגים שונים של
בדיקות, גם במטרת הבדיקה, וגם באופן הביצוע שלהן.

בדיקות ייחודית

בבדיקות ייחודית אנחנו תמיד מתייחסים לתוכנה כאלו אוסף ייחודות - חלקים בלתי ניתנים לחלוקה,
ובודקים כל "יחידה" צאת בבודד משאר התוכנה כדי לוודא שהיא מתפרקת כמצופה.
הערך שבדיקות ייחודית נוטנות הוא כפוף: מעבר לעצם העובדה שבדיקות ייחודית מאשרות שכל
יחידה ייחודה פועלת כמו שצריך, הם גם גורמות לנו לתכנן ולכתוב את הקוד בצורה יותר מודולרית
כדי שנוכל לנכתב בבדיקה לכל חלק בלי תלות בחלקים האחרים.

למעשה קיים סגנון פיתוח שנקרא **Test Driven Development** (פיתוח מכון טסיטים) או בקיצור
TDD שבו כתבים את הבדיקות מראש ולאחר מכן כתבים את המימוש על מנת לגרום בדיקה

לעבור. TDD מחייב את המפתח לכתוב את הקוד בצורה מאוד מודולרית עם תלויות מוגדרות היטב. חשוב לציין שזו רק דרך אחת, ולא חייבים לעשות TDD על מנת לכתוב בדיקות ייחודית אינכוטיות.

עם זאת, לבדיקות ייחודית יש כמובן גם חסרונות. הן אמנם מודאות תקינות של כל ייחידה ויחידה, אך לא מבטיחות דבר לגבי התוכנה בשילמותה, בעיקר ככל שהMORECODES של התוכנה גדולה. בנוסף, זה לא טריוויאלי למצוא איזון טוב בין נמות ואיזורי הטסטים לבין אינכוט הכספי וצימוד בין הטסטים ליחידות אותן בודקים.

בכתיבת בדיקות ייחודית חשוב מאוד לוודא שבודקים את התוצאה של היחידה ולא את הדרך שבה התוצאה הושגה, אחרת קל להגיע במצב שכל שינוי בקוד ידרשו גם שינוי בבדיקה.

בדיקות קומפוננטה

סוג נוסף של בדיקות הוא בדיקות קומפוננטה. בניגוד לבדיקות ייחודית שבודקים את התוכנה מבפנים החוצה, בבדיקות קומפוננטה אנחנו מתייחסים לכל המודול כאלו ייחידה אחת, ומתקדים בבדיקה החוצה שהמודול חושף, ככלומר מבחוץ פנימה.

אחד מהבעיות העיקריות בבדיקות קומפוננטה היא עמידות לשינויים. ככלומר שינויים במימוש התוכנה לא אמורים "לשבור" טסטים קיימים, ואם טסט נשבר קרוב לוודאי שגם רגסיה ויש לתקן את המימוש.

בדיקות קומפוננטה מתאימות במיוחד לפיתוח זמייש (Angular). הן בתסրיט שבו אנחנו מפתחים backend לאפליקציית client, ובין אם אנו מפתחים שירות בארכיטקטורת מיקרו-סרביסים, תמיד יש ערך גדול ביכולת להגיד ממשקים ברורים בין הרכיבים השונים בשלב מוקדם בתהילה, ולאחר מכן לפתחים של הרכיבים האחרים להסתמך עליהם. כיסוי של אותם ממשקים בבדיקות קומפוננטה מאפשר בהשכמה לא גדולה לוודא שבגרסאות הבאות של השירות אותם ממשקים ימשיכו לתפקיד צפויו.

בנוסף, בבדיקות קומפוננטה הן נקודת הת Dönha טובה לכתיבת בדיקות, שכן בדרך כלל ניתן לנפות את ההתנגדות הצפוייה של שירות Node.js במספר לא גדול של בדיקות.

supertest

ב-node.js ניתן לבצע בבדיקות קומפוננטה בצורה מאוד אלגנטית באמצעות ספרייה בשם supertest. הספרייה supertest מאפשרת "לזייף" שאלות http לשירות דומה ל-client או שירות אמיתי אחר שפונה לשירות שבבדיקה, ולודא שתשובות עוננות על הדרישות שהציבנו.

לדוגמה נניח שיש לנו שירות כזה:

app.js

```
const express = require('express');
const app = express();

app.get('/greet', function ({ query }, res) {
  res.status(200).json({
    hello: query.name
  });
});

module.exports = { app }
```

השירות מקבל קריאת GET עם ערך name ב-query-string ומחזיר JSON עם מפתח hello והערך שהעבכנו ב-name. עכשווי בוואו נכתב בדיקת קומפוננטה פשוטה ע"י supertest:

0-supertest.test.js

```
const request = require('supertest');
const assert = require('assert');

describe('GET /user', function () {
  it('responds with json', async () => {
    // Arrange
    const { app } = require('./app');

    // Act, Assert
    await request(app)
      .get('/greet')
      .query({ name: 'Elementor' })
      .set('Accept', 'application/json')
      .expect('Content-Type', /json/)
      .expect(200)
      .expect(({ body }) => {
        assert.deepStrictEqual(body, { hello: 'Elementor' });
      });
  });
});
```

בבדיקה, אנחנו צורכים את האפליקציה שלנו בשלמותה, ובאמצעות המетодה get מזיהיפים שאלתת GET לנטייב greet עם ערך name בשדה header-i שמצין שאנוינו

מעוניינים לקבל JSON חזרה, ובעזרת המתוודה expect אנחנו מודאים שאכן קיבלנו JSON, עם סטטוס 200 והערך של hello הוא אכן Elementor כפי שהלחנו בשאלתא.

סוגים נוספים של בדיקות

עד כה המקדנו בבדיקות וולידציה - בדיקות אשר מודאות שהתוכנה פועלת בהתאם לציפיות. אבל גם תוכנה שעובדת להפליא לא שווה הרבה אם קשה להבין ולהזק אותה.

eslint

עד ראשון לקוד קריוא הוא היcmdות ל"תקן כתיבה" (Coding Standard) כלשהו. בבדיקות כאלה נקראות linting, ותוכנה שביצעת את הבדיקה הזאת נקראת linter. הלינטר המוכר ביותר ל- JavaScript הוא eslint והוא ספציפי ל-node.js ומתחייב ל linting של כל קוד JavaScript באשר הוא.

כדי להתחיל להשתמש ב-eslint יש לבצע 2 צעדים:

1. להתקין אותו ע"י npm install --save-dev eslint
2. להיות ותקן כתיבה הוא דבר די סובייקטיבי, צריך לספר ל-eslint מהו התקן שאנו רוצים ע"ז הריצה של eslint-init ומענה על כמה שאלות:

```
$ ./node_modules/.bin/eslint --init
```

? How would you like to use ESLint? **To check syntax, find problems, and enforce code style**

? What type of modules does your project use? **CommonJS (require(exports))**

? Which framework does your project use? **None of these**

? Does your project use TypeScript? **No**

? Where does your code run? **Node**

? How would you like to define a style for your project? **Use a popular style guide**

? Which style guide do you want to follow? **Airbnb:**
<https://github.com/airbnb/javascript>

? What format do you want your config file to be in? **JavaScript**

למענה סיפרתי לאשף שאני כותב קוד JavaScript על Node.js ואני אוהב את התקן הכתיבה של Airbnb. האשף שומר את ההגדרות שלי בקובץ `.eslintrc.js`, ובזה מסתיימות ההגדרות.

עכשו כדי לבדוק מה eslint נותן לנו נכתב תוכנה קצרה:

app-pre.js

```
var the_answer = 42;

console.log('The Answer to the Ultimate Question of Life, the
Universe, and Everything is ' + the_answer);
```

:eslint וריז את

\$./node_modules/.bin/eslint **/*.js

./app-pre.js

```
1:1 error Unexpected var, use let or const instead    no-var
1:5 error Identifier 'the_answer' is not in camel case  camelcase
3:1 warning Unexpected console statement            no-console
3:13 error Unexpected string concatenation      prefer-template
3:91 error Identifier 'the_answer' is not in camel case  camelcase
```

☒ 5 problems (4 errors, 1 warning)

2 errors and 0 warnings potentially fixable with the `--fix` option.

ואלה! מייד מזהה את הבעיה הבאות:

- השתמשנו ב-var כשעדיף להשתמש ב-let או const היותר בטוחים יותר.
- שם המשתנה the_answer הוא לא ב-camelCase כפי שהוגדר בתקן של Airbnb.
- אנחנו משתמשים באופרטור + כדי לחבר את הערך של the_answer לטקסט של ההודעה, כשיותר קריא היה להשתמש ב-template literal.
- השתמשנו ב-console.log - בכלל, בפודקשן אין סיבה להשתמש בו ולכן ההתראה.

עכשו נתקן וננסה שוב:

app-post.js

```
const theAnswer = 42;
```

```
/* eslint-disable-next-line no-console */
console.log(`The Answer to the Ultimate Question of Life, the
Universe, and Everything is ${theAnswer}`);
```

ביצענו את השינויים הבאים:

- שינוינו את שם המשתנה ל-theAnswer ב-camelCase.
- הכרזנו עליו עם const כי אנחנו לא מתכוונים לשנות אותו.
- השתמשנו ב-template literal כדי להזrik את הערך של המשתנה לגוף ההודעה.
- והוספנו הערת eslint-disable-next-line eslint面前 הקדימה ל-log.console כדי להסביר לנו מה שאנחנו משתמשים ב-console במודע וזה אינה טעות eslint.

נרים את eslint:

```
$ ./node_modules/.bin/eslint **/*.js
```

מצוין! עכשו הקוד שלנו קרייא יותר לנו, ולחברי הצוות האחרים.

eslint הוא כלי מאד פתוח לקונפיגורציה, שכן יש אינספור סגנונות כתיבה שונים. וכן הוא תומך בהמון אינטגרציות, בעיקר עם עורכי טקסט. לדוגמה אם אתם משתמשים ב-vscode לכתיבה הקוד, קיים תוסף eslint שמציג את הבעיות של eslint בתוך העורך - תוך כדי הכתיבה, ומאפשר תיקון שגיאות אוטומטי.

npm audit

כפי שהזכרנו בתחום הפרק, חבילות-Node.js נוטות להיות תלויות בהרבה חבילות אחרות כשהן בתורן תלויות בחבילות נוספות. מטבע הדברים, כל הזמן נמצאות חולשות אבטחה חדשות בחבילות השונות, ובעיקר בחבילות שנמצאות בשימוש נפוץ - מכיוון שהן נחקרו יותר. חולשות אלו בדרך כלל מתקנות תוך זמן קצר, אבל איך ניתן לדעת אם אחת מהחבילות הרבות שהשירות שלי תלוי בה סובלת מחולשת אבטחה?

בשביל זה קיימת ב-node פקודה audit.

כשMRIצים npm מול חבילת js.Node, הוא סורק את כל התלוויות של אותה החבילה ומדוחה אם מי מהתלוויות סובלת מבעיית אבטחה.

למשל כשMRIצים audit npm מול חבילת המכילה גרסה בעייתייה של הספרייה lodash נקבל את הפלט הבא:

```
$ npm audit
```

```
==== npm audit security report ===
```

```
# Run npm update lodash --depth 1 to resolve 1 vulnerability
```

```
High      Prototype Pollution
```

```
Package    lodash
```

```
Dependency of  lodash
```

```
Path      lodash
```

```
More info  https://npmjs.com/advisories/1065
```

found 1 high severity vulnerability in 387 scanned packages

run `npm audit fix` to fix 1 of them.

וכפי שモופיע בסיום הפלט כל מה שצריך לעשות כדי לתקן את הבעיה הוא להריץ:

```
$ npm audit fix
```

```
+ lodash@4.17.15
```

```
updated 1 package in 3.208s
```

fixed 1 of 1 vulnerability in 387 scanned packages

ונowiן npm audit אוטומטית מעדכן את החבילה הבעייתה לגרסה בטוחה.

از מה יוצא לי זהה?

בדיקות אוטומטיות פותחות את הדלת לפיתוח גמייש (אגילוי). על מנת לאפשר פיתוח מהיר, יכולת לבצע שינויים בקוד ולהביא אותם ללקוח בזמן קצר מבלוי לאבד מאיכות המוצר, חייבות להיות דרך "להקפייה" את המוצר החיצוני של הקוד - החוצה שלו מול הלקוחות, בצורה נזאת שעדין יהיה ניתן לבצע שינויים במימוש ולהוסיף יכולות חדשות בלי פחד מתמיד שהשינוי ישבור משהו שעבד עד כה.

בדיקות אוטומטיות בשילוב עם תהליך CD/I (תהליכי אוטומטי שMRIIZ בבדיקות לפני שכל פיסת קוד כניסה למערכת ניהול הגרסאות, וכן לפני שהיא נפרשת בשרת את נשלחת ללקוח) נוותנות לבדוק את היכולת הזאת.

כדי למקסם את היתרונות של הבדיקות האוטומטיות בפיתוח גמייש, חשוב לחת את הדעת על האיזורים אותם חשוב לבדוק בבדיקות מול איזורים שנמצאים פחות בשימוש או שהם פחות מורכבים, ולהתחליל את כתיבת הבדיקות דוקא מהמקומות המורכבים והנמצאים בשימוש תדר. בנוסף, כאשר שלמרות הכל נמצא באג בתוכנה, לאחר תיקונו, כדאי כתוב טסט שמנסה את הבאג הזה. כך נוכל לוודא שהבאג הספציפי הזה כבר לא ייחזר על עצמו ואיכות המוצר תשתרף.

ניתן לבדוק את הcoverage (coverage), כלומר את אחוזי הקוד (לפי כמה שורות) שמכוסות ע"י טסטים באמצעות הריצה של mocha עם המפתח coverage-. חומרת השבט גורסת שהכיסוי (coverage) האופטימי של בדיקות הוא עד 85% מהקוד, משלב זה והילך הוספה של בדיקות חדשות לא מחזירה ערך שווה להשקעה.

באמצעות כיסוי בדיקתי של קטיעים קריטיים לפעולת המערכת, ניתן להגיע למחוזרי שחרור קיצרים עם אחוזי רgresיה נמוכים מאוד, מפתחים רגועים ולקוחות מאושרים.

שינויים בין גרסה 1.2.0 לגרסה 1.2.1

נוספו הסברים על CommonJS ו ECMAScript Modules (ESM) והבדלים ביניהם. ההסברים הם בתת פרק ב- `Require` ומודולים ובתת פרק בפרק המורחב על `chalk`.

בפרק המורחב על `chalk` בוצע תיקון – קביעה של הגרסה של `chalk` לגרסה 4.

נוסף הסבר על איך לומדים עם Chat GPT בתחילת הפרק.

תווך המחיר של Heroku.

תודה רבה על הרכישה של הספר!

עבדתי מאוד קשה על הספר זהה: שעות רבות של כתיבה, הגהה, תיקונים ומעבר על תוכרי העריכה. יותר מ-1800 אנשים תמכו בספר זהה ואיפשרו לו לצאת לאור.

הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהקינדל, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתוך תקווה שהרוכש והותמן לא ינצל את האמון שנטתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפצה שלו. אני מאמין שרוב האנשים הוגנים.

העתיק זהה נמכר ל:

nivbuskila@icloud.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נחברים פרטי הרוכש באופן שקוף למשתמש. כדאי מאוד להמנע מהעתיקה של הספר ללא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיהקו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!