

תודה רבה על הרכישה של הספר!

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

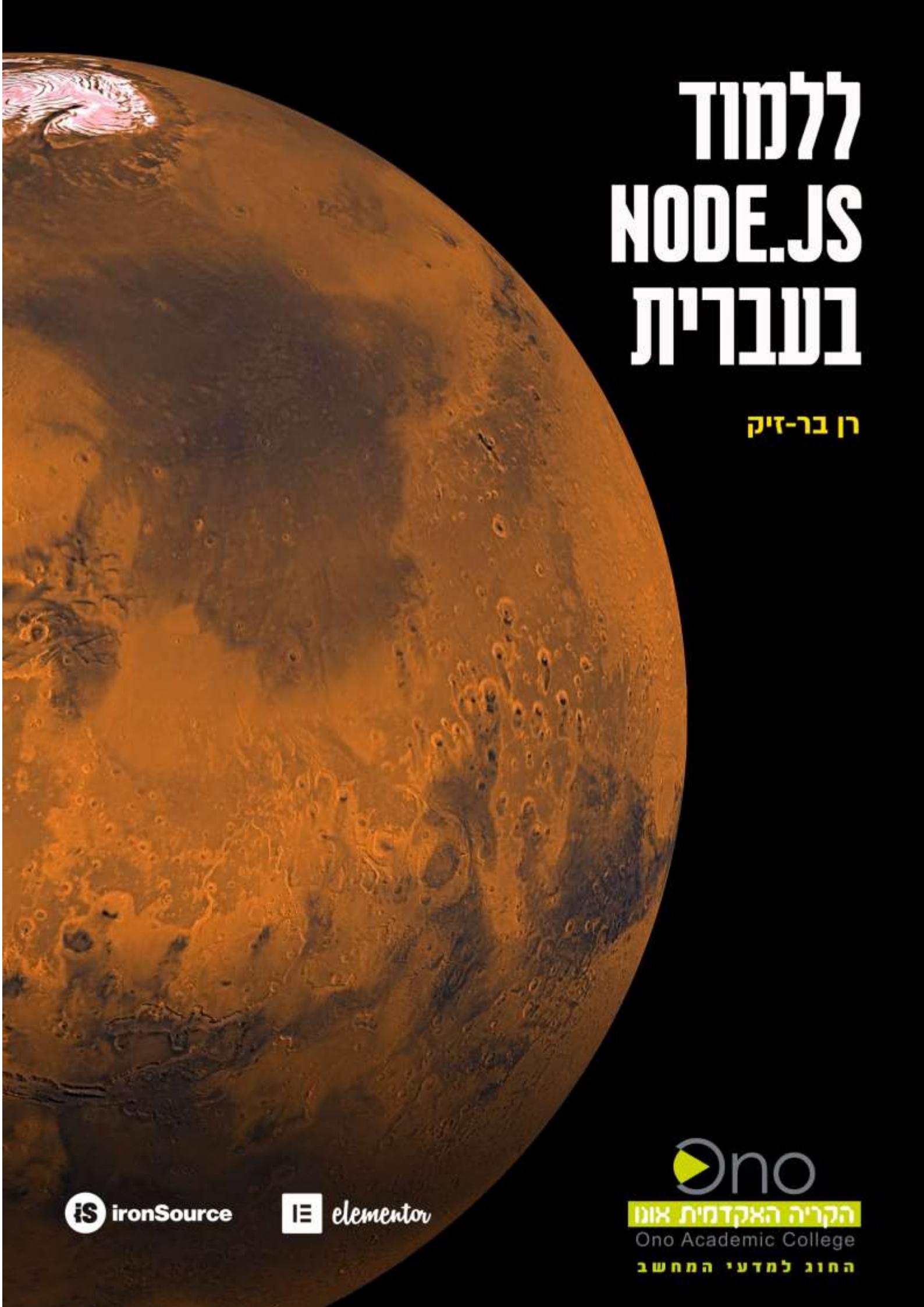
הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהקינדל, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתוך תקווה שהרוכש וה透מך לא ינצל את האמון שנתי בלהעתיקה סיטונאית של הספר לאנשים אחרים והפצה שלו. אני מאמין שרוב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

העתיק זהה נמכר ל:

beninson@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלי - כלומר בתוך דפי הספר נחברים פרטיו הרוכש באופן שקוֹף למשתמש. כדאי מאוד להמנע מהעתיקה של הספר ללא רכשו שלו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטיהם שלכם באתר ומיחקקו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!



# ללמוד NODE.JS בנברית

ח בר-זיק

# לימוד Node.js בעברית

## REN BAR-ZIK

מהדורה: 1.2.0



כל הזכויות שמורות © רן בר-זיק, 2019.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קיבלת רשות ל-A-בלעדי, לא-ייחודי, אישי, בלתי ניתן להעברה (למעט על פי דין), ובלתי ניתן להסבה לעשות שימוש אישי בספר זה לצרכים לימודים בלבד.

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

מותר לך לצלט קטעים קצרים מהספר במסגרת השימוש הוגן, ככלומר פסקה או שניים, 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 בתוך תוכנות שתפתח. אם אתה רוצה להכנסו אותן לפרויקט שלך, שלח מייל ונדבר על זה.

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

הפקה: כריכה – סוכנות לסופרים  
[www.kricha.co.il](http://www.kricha.co.il)



## תוכן עניינים

10	על הספר .....
10	על המונחים בעברית .....
11	על המחבר .....
12	על העורכים הטכניים .....
12	בנג'מין גרינבאום .....
12	גיל פינק .....
15	הקדמה – מה זה ואיך זה המתחיל .....
17	דרך הלמידה .....
18	התקנת סביבת עבודה ועבודה עם טרמינל .....
20	התקנה על חלונות .....
24	התקנה על מק .....
24	התקנה על לינוקס .....
25	תקלות נפוצות .....
25	כנתיבת התוכנה הראשונה .....
30	<b>Require ומודולים</b> .....
35	היכרות עם הדוקומנטציה של <i>Node.js</i> .....
43	גרסאות סינכרוניות למתודות אסינכרוניות .....
49	<i>package.json</i> – הכרה ראשונית והפעלה של <i>NPM package</i> .....
50	יצירת <i>os.js.package</i> לפרויקט שלנו .....
51	התקנת המודול הראשוני .....
53	שימוש במודול חיצוני .....
58	עבודה אסינכרונית ומעבר מקבוקים לפורמייסים ול- <i>let</i> - <i>await</i> .....
62	מודולים-ב- <i>js.Node</i> שתומכים בפורמייסים באופן טבעי .....
65	<b>אירועים</b> .....
68	כיבוי מאזין .....
68	הפעלת יותר מאירוע אחד .....
70	הצמדת כמה פונקציות מאזיניות לאירוע אחד .....
72	העברת נתונים באירועים .....
75	יצירת שרת <i>HTTP</i> בסיסי .....

<b>84</b>	<b>Node.js של Event Loop</b>
<b>84</b>	<b>מетодות הטימרים</b>
84	תורץ כשאי אומר לך – setTimeout
85	תורץ מיד עם קולבק – setImmediate
85	הלוואה הקבואה setInterval
<b>86</b>	<b>טור הקריאה</b>
<b>95</b>	<b>Streams</b>
<b>97</b>	<b>סוגי הסטרים השונים</b>
97	סטרים טרנספורמציה
<b>99</b>	<b>אירועים בסטרים</b>
<b>106</b>	<b>אריזת הקוד שלנו כמודול</b>
<b>112</b>	<b>קביעת גרסאות</b>
113	גרסאות סמנטיות
116	קביעת גרסאות סמנטיות ב-package.json
<b>120</b>	<b>התקנה גלובלית ו-CLI</b>
<b>124</b>	<b> כתיבת bin והתמכשות עם ה-CLI</b>
<b>134</b>	<b>Sockets</b>
<b>142</b>	<b>קריאה משאים באמצעות מודול path</b>
<b>147</b>	<b>package.json scripts</b>
<b>149</b>	<b>סקרייפטים עם שמות</b>
<b>149</b>	<b>משתני סביבה</b>
150	קביעת משתנה סביבה דרך הסкриיפט
151	קביעת משתנה סביבה דרך הגדרות מערכת הפעלה
153	קביעת משתנה סביבה דרך קובץ
<b>154</b>	<b>dev dependencies</b>
<b>158</b>	<b>אקספרס</b>
<b>159</b>	<b>טיפול במетодות של בקשות HTTP</b>
<b>165</b>	<b>ראוטינג</b>
<b>168</b>	<b>MiddleWare</b>
<b>170</b>	<b>URL דינמי</b>
<b>173</b>	<b>tabניות</b>
<b>177</b>	<b>חיבור ל-MySQL</b>
<b>178</b>	<b>חיבור ראשוני</b>

<b>180</b>	<b>שאילתת בסיסית.....</b>
<b>181</b>	המרת הקוד לעובודה עם פרומיסים ולא עם קוילבקים.....
<b>183</b>	<b>Prepared Statement</b>
<b>186</b>	<b>עליה לפורודקشن.....</b>
<b>186</b>	עליה לפורודקشن עם שרת.....
<b>187</b>	עליה לפורודקشن בענן.....
<b>191</b>	<b>סיכום.....</b>
<b>191</b>	<b>מיטאפים.....</b>
<b>191</b>	קבוצות דיוון.....
<b>191</b>	גיטהאב.....
<b>191</b>	התנדבות בעמותות ובמיזמים.....
<b>193</b>	<b>נספח: בדיקות אוטומטיות ב-<i>Node.js</i>.....</b>
<b>193</b>	מה זה בדיקות אוטומטיות?.....
<b>195</b>	בדיקות אוטומטיות.....
<b>197</b>	<b>Mocha</b>
199	describe
199	.....it
201	מחזור חיים.....
203	מבנה בדיקה.....
<b>204</b>	<b>פרימורק בבדיקות.....</b>
205	assert.ok(value)
205	assert.notStrictEqual(actual, expected) \ assert.strictEqual(actual, expected)
206	assert.notDeepStrictEqual(actual, expected) \ assert.deepStrictEqual(actual, expected)
206	assert.throws(fn)
<b>207</b>	<b>ספריות נוספת.....</b>
207	mock
<b>209</b>	<b>1-spies.js</b>
211	mock(obj)
212	בדיקות עם קריאות http
<b>214</b>	<b>סוגי בדיקות.....</b>
214	בדיקות יחידה.....
214	בדיקות קומפוננטה.....
<b>216</b>	<b>סוגים נוספים של בדיקות.....</b>
216	eslint
218	npm audit
<b>219</b>	<b>איך מזמין מזמין?.....</b>



## 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: חברותים של חברות ענק ועד תוכנות תחזקה ופיתוח שונות. `Node.js` היפה בשנים האחרונות לאחת התשתיות החשובות ביותר של הרשת ועולם הפיתוח. גם אנשים המתכנתים על גבי פלטפורמות אחרות ושפות אחרות משתמשים בתוכנות מבוססות `Node.js` למטרות שונות – בין אם בדיקת הקוד שלהם, ריצת בדיקות או כל שימוש אחר.

לימוד `Node.js` מחייב הכרה עמוקה עם שפת ג'אוوهסקריפט. בספר הקודם, "לימוד ג'אוوهסקריפט בעברית", למדתי ג'אוوهסקריפט ברמה המסיפה להתחלה הקיירה בספר זה. הספר מלמד `Node.js` ומתחילה במבנה סביבת העבודה והתקנת הפלטפורמה. הוא ממשיך בהקניית 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 אורך ונכבד המלמד על אקספרס, המודול הפופולרי לבניית שרת רשת. אנו לומדים גם על העלאת האפליקציה שלנו לענן באמצעות "הרוקו". בסיוםו של כל פרק רלוונטי יש תרגילים והסבירים מפורטים הכללים גם שרטוטים.

הספר מיועד לכל מתכנת ג'אוوهסקריפט שמעוניין למדוד על העולם המופלא של `Node.js` ולמתכנתים המכירים את `Node.js` אך זקוקים 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" מלאה אותי תמיד. מצד אחד, האקדמיה ללשון העברית מספקת לנו מונחים רבים בעברית. מצד שני, בתעשייה ההייטק, שמננה אני מגייע, איש לא משתמש ברבים מהמונחים האלה. אם תגינו לראיון בעבודה ותגידו: "במפגש המתכנתים האחרון שמעתיך על דרך חדשה לבצע הידור שבודק הזרחות במנשך מבוסס הבטחות", סביר להניח שלא תקבלו את העבודה. אבל אם תגידו "במיטאף האחרון שמעתיך על דרך חדשה לבצע קמפול שבודק אינדנטציה ב-API מבוסס פרומיסים" – יבינו על מה אתם מדברים. זו הסיבה שלא תמצאו מילים כמו "הידור", "מחלקה" או "מרשתת" אלא "קמפול", "קלואס" או "אינטרנט". המונחים שבהם השתמשתי הם המונחים שבהם משתמשים בתעשייה בפועל. בכל מקום שבו אני משתמש לראשונה במונח בעברית, אני מספק גם את הגרסה שלו באנגלית, כדי שתוכלו להכנסו אותו לחיפושים שלכם בגוגל.

חשוב לציין שאיני בז כליל לאקדמיה ללשון וshall ממהונחים שלא אכן נכנסו לשפה המדוברת במרכז הטכנולוגיה השונות (למשל: קובץ או מסד נתונים), אבל בכל מקום שהייתה לי ברירה בין להיות מוכן לבין לעמוד בכללי הלשון, העדפתתי להיות מובן.

## על המחבר

REN BAR-ZIK הוא מפתח תוכנה משנת 1996 ב מגוון שפות ופלטפורמות ועובד כ מפתח בכיר במרכז פיתוח של חברות רב-לאומיות, מ-HP ו עד Verizon, שם הוא מפתח בטכניקות מתקדמות הן בצד הלוקה הן בצד השירות, ושם דגש על בניית תשתיות פיתוח נכונה, על שימוש ב-CDI ו\_COMBOן על אבטחת מידע.

נסוף על עבודתו כ מפתח במשרחה מלאה, REN הוא עיתונאי ב"הארץ" במדור המחשבים, שם הוא מסקר נושאים הקשורים לטכנולוגיה ולאבטחת מידע וכותב על אינטרנט ורשתות.

משנת 2008 מפעיל REN את האתר "אינטרנט ישראל" ([internet-israel.com](http://internet-israel.com)), שהוא אתר טכני המכיל מדריכים, מאמרים וסבירים על תכנות בעברית, ומתעדכן לפחות פעם בשבוע.

REN הוא מחבר הספר "לימוד ג'אווה סקורייפט בעברית".

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

# על העורכים הטכניים

## בנג'מין גרינבאוֹם

בנג'מין גרינבאוֹם הוא מתכנת מנוסה, מומחה לג'אווהסקרייפט בעל רקע עשיר של עבודה במגוון חברות רב-לאומיות ובמגוון תפקידים ובוגר תואר ראשון למדעי המחשב באוניברסיטה העברית. הוא מפתח בצוות הליבה של Node.js ובמסגרת תפקידו הוא כותב קוד של Node.js ממש, מציע ומצביע על פיצ'רים בשפה ושותף בהחלטות השונות הרלוונטיות ל-Node.js.

בנג'מין היה שותף כעורך טכני לשורה של ספרים מוביילים בתחום נושא ג'אווהסקרייפט, כגון *Node.js Know JS* ו*Exploring ES6* . הוא מרצה בכנסים בארץ ו בחו"ל וחבר מוביל בקהילות פיתוח בארץ ובעולם.

## גִּיל פִּינְק

gil fink הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert . sparXys Microsoft Developer Technologies MVP .  
כיום הוא מייעץ לחברות ולארגונים שונים, שם הוא מסייע בפיתוח פתרונות מבוססי אינטרנט ו-SPAs .  
הוא עורך הרצאות וסדנאות ליחידים ו לחברות המעונייניות להתרמהות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט ( Microsoft Official ) "Pro Single Page Application Development (Course MOC ) , מחבר משותף של הספר "AngularUP" (Apress) ושותף בארגון הכנס הבינלאומי .

לפרטים נוספים עלgil: <http://www.gilfink.net>

# על החברות התומכות

## אלמנטור

אלמנטור מפתחת פלטפורמת קוד פתוח לבניית אתרים שמשנה את הדרך בה בונים אתרי אינטרנט בשוק המזקיע. אלמנטור מעניק למעצבים את החופש ליצור עמודי אינטרנט ללא צורך בקוד ולפתחים את החירות לדוחוף את הgebung, לרענן ולהרחב את המערכת בצורה קלה ומהירה באמצעות API ידידותי למפתחים, ובכך לחסוך זמן פיתוח ולהיות ייעילים ורוחניים.

עם מיליון+ אתרים הפעילים על אלמנטור וצמיחה חודשית מדיהימה, התגבשה סבב הפלטפורמה קהילה חזקה המונה מאות אלפי חברי, מפתחים, משווקים ומעצבים, המקיים מיטאפים בכל רחבי העולם. מידי יום האלמנטוריסטים מייצרים וצורכים אלפי שעות של הדרכות, סרטוי השראה ובלוגים מעמיקים, ומפתחים תורמים קוד ורעיוןות באמצעות GitHub. האקויסיטם המזקיע של אלמנטור מתפתח ללא הפסקה והוא אוצר המושך ומעשיר את היכולות של כל יוצר אינטרנט.

באלמנטור אנחנו משתמשים בטכנולוגיות קוד פתוח מתקדמות לפיתוח כל אינטרנט חדשניים ו מהירים. אם גם אתם רוצים להיות חלק מהטכנולוגיה שמשנה את חווית האינטרנט בעולם ויש לכם את הידע כדי לבנות עולם יפה יותר אנחנו מוחפשים אתכם, מעצבים UI&UX, מפתחים Full Stack, מהנדס DevOps ו Big Data עם מומחיות בניהול Kubernetes על פלטפורמות הענן של AWS.

## ironSource

חברת ironSource, הנחשבת לחברת מובילה בכל הקשור לmontezuma באפליקציות ולפלטפורמות פרסום בוידאו, וחולשת על יותר ממיליארדים וחצי שחקנים ברחבי העולם, ה奏音 in בפרסומות על גבי תשתיות החברה. החברה עוזרת למפתחים לחת את האפליקציות שלהם לשלב הבא, זאת גם בזכות רשות הווידאו העצומה שלה - והמספרים מדברים בעד עצם, עם יותר מ-80,000 אפליקציות המשמשות בטכנולוגיות של החברה כדי לפתח את העיסוק שלהן.

# הקדמה – מה זה ואיך זה התחיל

js. Node הופיע בשנת 2009. מדובר בסביבת הריצה של ג'אווהסקרייפט בסביבת שרת. סביבת ההריצה הזאת בנויה יכולה על מנת הריצה של כרום 87. מדובר במנוע חזק ומהיר מאוד שמשתמשים בו בכרום. ב-js. Node הרצה היא מוחזק לדף, אך מנוע 87 מאפשר לג'אווהסקרייפט לרווח מהר מאוד ויעיל מאוד. מרכיב נוסף של js. Node היא ספריית `uvlib`, הכתובה ב-C ומאפשרת הריצה של פעולות קלט ופלט ב מהירות רבה.

מתכנת בשם ריאן דאל רצה לבנות סמן התקדמות של טיענת קובץ. הוא ניסה לעשות זאת בשרתים הקודמים, ובראשם Apache, אך לא הצליח לעשות כן בגלל בעיות ביצועים. הוא החליט לבנות שרת מבוסס על 87 המהיר, עם דרכי פשוטות לבצע קלט ופלט למערכת ההפעלה ועם ג'אווהסקרייפט.

בניגוד לנסיבות הריצה אחרות של שפות אחרות, שבהן המשתמש נדרש לנוהל את התהיליכים של המעבד, ב-js. Node הקוד של המשתמש רץ על תהליך אחד של המעבד ואינו חוסם אותו כאשר הוא מוחכה לנוטונים שמאזינים. תהיליכים נוספים מנהלים אוטומטית דרך ספריית `uvlib`. דרך הפעולה הזאת מאפשרת ל-js. Node לעבוד מהר מאוד עם פלט וקלט, כיון שגם היא מבצעת בקשה כלשהי לשרת אחר, מערכת קבצים או מסד נתונים, התהיליך אינו נחסם אלא הבקשת נשלחת ו-js. Node ממשיכה לרווח. זה מאפשר בಗל האсинכרוניות המובנה שיש ב-js. Node והופך את סביבת הריצה הזאת לטובה מאוד בקלט ופלט.

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

js. Node פופולרית להדרה. בשעת כתיבת ספר זה (יוני 2019), יש יותר ממאה אלף חבילות תוכנה בקוד פתוח שזמןנות למשתמשים ב-js. Node לשימושים שונים. שרתים רבים נכתבם בעולם על js. Node ומשתמשים בתוכנות מבוססות js. Node בכל מקום: אפליקציות מובייל ועד אפליקציות דסקטופ, כלי עזר לשרתים באמצעות ה-CLI ולשפות אחרות ועוד. js. Node נמצאת בכל מקום.

כיום מי שmobiled את js. Node הוא מודד ללא כוונת רוח שנקרה OpenJS Foundation – מוסד שבנו על פי עקרון "הamodel הפותוח" וכל אחד שיש לו מספיק רצון יכול להשתתף בדיונים ולהשפיע על ההתפתחות העתידית של סביבת הריצה.

js. Node היא לא שפה, השפה היא ג'אווהסקרייפט. js. Node היא סביבת הריצה. קל לכל מתכנת או מתכנתת ג'אווהסקרייפט לעבוד היטב עם js. Node. ספר זה אינו מלמד ג'אווהסקרייפט ואני יוציא

מנקודת הנחה שהקוראים מכירים היטב ג'אווהסקרייפט ובדגש על ג'אווהסקרייפט מודרני ואסינכרוני. אם איןכם מכירים היטב את השפה זו, אני ממליץ לכם לקרוא את ספרי הקודם, "לימוד ג'אווהסקרייפט בעברית", שיצא בהוצאה הקדמית אוננו. למוד של הספר הקודם יביא אתכם למצב שתוכלו להבין את הספר הזה היטב.

בספר נלמד `js` משלב ההתקנה ועד השלב שבו נדע לשЛОט בה באופן מושלם. הדבר החשוב ביותר שחייב שיכדי לזכור ב-`js` הוא שבעשר הספריות העצום שלה בעצם חוסך המון מזמן הכתיבה. אנו נלמד פה למשל איך מקיים שירות `HTTP`, איך הסיכוי שתCTRוכו לעשות את זה בח"ם האמ"ת"ם הוא אפסי, כיצד שהמודול הפופולרי `Express` משתמש את רוב המתכנותים ליצור שירות `HTTP`. כמו כן תוכלו להשתמש במידע שתלמדו בספר זה כדי להוסיף למודולים קיימים או לכתב אפליקציות של ממש או שירותים של ממש שימושיים במודולים של `js`. ברגע שתבינו איך עובדים עם ג'אווהסקרייפט על סבב הרצה זו – השמיים הם הגבול. כאמור, משתמשים ב-`js` `Node` בכל מקום. גם במקומות שבהם כתבים בעיקר בשפות תכנות אחרות, כיצד שהכוח של `Node` הוא יכולת שלה לפעול בכל מקום, גם בנסיבות תחזקה וגם בנסיבות של אבטחת מידע.

## דרך הלמידה

דרך הלמידה היא פשוטה ביותר — קריית הפרק ותרגול של התרגילים שנמצאים בסופו. התרגול הוא קריטי, בגלל זה חשוב מאוד להשיקיע זמן בפרק הראשון ובננות את סביבת העבודה שלכם. ללא בנייה של סביבת העבודה ותרגול — הקרייה לא תהיה אפקטיבית ממש. ראשית יש להבין את החומר, לקרוא פעם, או פעמיים או שלוש, ואז לבצע את התרגילים. אחריו 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 ללמד Node.js ללא לכלי הידיים וכטיבה אמיתית. הספר הוא כל עזר, הוא לא יחליף את ההקלדה שלכם. בדרך כלל הקושי האמיתי הוא בבנייה של סביבת עבודה יציבה וטובה, لكن הפרק הראשון שעוסק בהתקנת סביבת עבודה הוא קריטי.

לא תמיד כל הסבר המופיע בספר הוא קולע או מתאים. אם קראתם את הפרק פעם ופעמיים ושלוש פעמיים ועדיין לא הבנתם — הבעיה לא בכם אלא בהסביר. לא להתייחס — פה כדאי להתייחס בקחילה של ג'אווהסקריפט ויש לא מעט כאלה בפייסבוק ובמקומות אחרים. גם חיפוש בגוגל לפעם יכול להוציא אתכם מבוֹז אמיתי. נתקעתם? אל תתייחסו — הבעיה לא בכם. Node.js קלה אבל יש בה כמה חלקיים קשים. נתקעתם? לא לדאוג — בקשו חילוץ. חפשו בגוגל, שחקו שוב ושוב עם הדוגמאות ובסוף זה ישב. אם אני הצלחתי — כל אחד יכול.

# התקנת סביבת עבודה ועבודה עם טרמינל

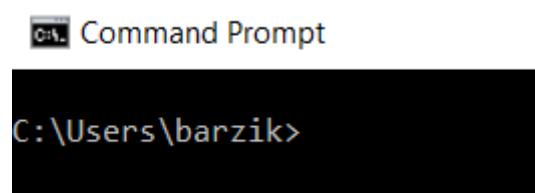
כאמור, `js` היא סביבת הריצה, וכדי שהיא תוכל לדרוש ציריך להתקין אותה על המחשב, ממש כמו כל תוכנה אחרת. מה שההתקנה הזאת עשוה הוא פשוט למד — היא מאפשרת לנו להפעיל את `Node.js` כמו כל תוכנה אחרת. זה הכל. אנו רגילים לפתח תוכנות באמצעות איקונים, אבל חלק מהתוכנות עובדות באמצעות הטרמינל. מה זה טרמינל? מקום שבו אתם יכולים להקליד פקודות. הוא קיים בכל מערכת הפעלה. בחלונות מגעים לטרמינל באמצעות לחיצה על הזוכחת המגדלת (בחולנות 10) והקלדה של `cmd` — ראש תיבות של `command`. נגע לחלון שנראה כך:



פתחו את החילון הזה, הקלידו `notepad` ולחצו על `enter`. יפתח ה-`Notepad` של חלונות. ברכותי! הפעלתם תוכנה באמצעות שורת הפקודה. הקלידו `calc` ולחצו על `enter`, תוכנת המחשבון תיפתח. זו עוד תוכנה שהפעלתם באמצעות משקל הפקודה. המשק זהה, או הטרמינל בשפת העם, הוא סביבת העבודה של `Node.js`. מפעילים את `Node.js` באמצעות הטרמינל. זה בדיק מה שקרה בשרת "אמיתי". זכרו ששרת בסופו של דבר הוא מחשב — "תכן שמחשב ללא מערכת הפעלה גרפית אלא רק עם טרמינל — אבל מחשב שմבוסס על חלונות או על לינוקס. "תכן שהשתתח חזק בהרבה מהמחשב הביתי שלכם — אבל עדין מדובר במחשב לכל דבר. כאמור, `Node.js` רצה היבט על שרתים מבוסס חלונות ועל שרתים מבוססי לינוקס. לצורך העניין, המחשב שלכם בעצם הוא שרת.

צריך להכיר מעט את משקל הפקודה של הטרמינל ולהתמצא בו. כיוון שהטרמינלים שונים בין חלונות לlinokס, יש שוני קטן בין הפקודות. כיוון שהлонנות היא מערכת הפעלה הנפוצה, ומשתמשי לינוקס בדרך כלל מ信נים יותר בטרמינל, אני מסביר מה על הפקודות בחלונות. בסוף הפרק יש טבלה קטנה שבה מובאות הפקודות בלינוקס ובחולונות.

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



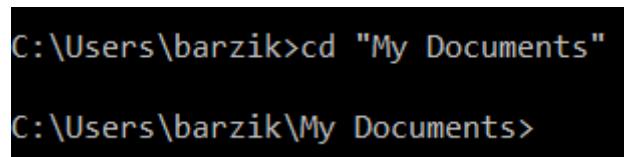
Command Prompt

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

אפשר לראות שאין נמצא בכוון C, בתיקייה Users ובתת התיקייה barzik, שהיא שם המשתמש שלי בחלונות. אם אני אפתח את סיר הקבצים, אני אוכל לנוט לתיקייה זו. הטרמינל הוא פשוט דרך נוספת לשוטט במחשב ולפעול בו – דרך שהיא לא גרפית, אבל כל מה שאני יכול לעשות במחשב הגרפי אני יכול לעשות בטרמינל.

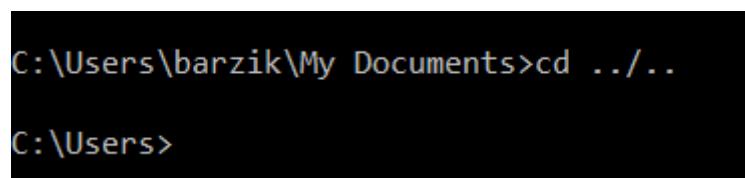
כדי לראות את רשימת הקבצים בתיקייה, אני צריך להקליד dir. הקלדה של dir ואז enter תראה לי את רשימת הקבצים שיש בתיקייה שבה אני נמצא. רשימת הקבצים זהו תהיה זהה לוחוטין לרשימת הקבצים אני רואה בסיר הקבצים כשאני נכנס לאותו מקום. אם אצור קובץ או תיקייה בסיר הקבצים ואקליד שוב dir בטרמינל כשאני באותו המיקום של סיר הקבצים, אוכל 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 רוחם בשם התיקייה, אני צריך להקליד אותו בມירכאות. אם אני משתמש ב-TAB הוא יעשה זאת עבורו.



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

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



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

ההפעלה של js.Node נועשית תמיד דרך הטרמינל. יש תוכנות שעוטפות את js.Node (כמוALKטרו) שלא מחייבות אותנו לעשות את זה, אבל אנו לא נתיחס לכך בספר זהה.

פקודה	פקודה בחלון/מק	פקודה בחלוןנות	פקודה בLINUKS/מק
הציג את רשימת הקבצים והתיקיות בתיקייה	dir		ls -a
מעבר לתיקייה אחרת	cd	cd	
יציאה מהטרמינל		exit	exit
נקוי המסך		cls	clear

לאחר שנאנו יודעים איך לעבוד עם הטרמינל, נתקן את js.Node. ההתקנה שונה במערכות הפעלה שונות אבל בכלל היא קלה למד'. בחרו את מערכת הפעלה שלכם והתקינו את js.Node לפי ההוראות.

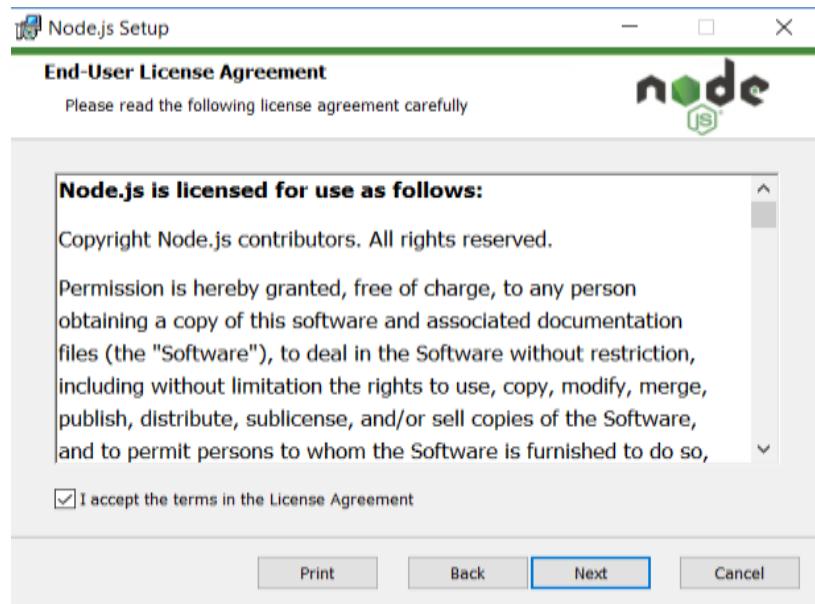
## התקנה על חלונות

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

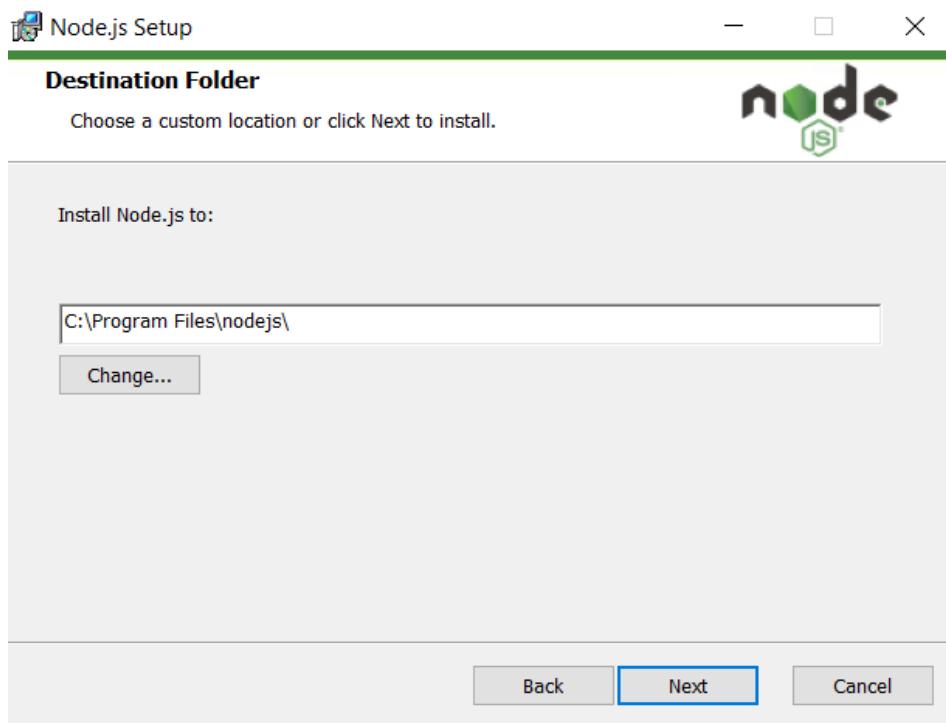
אנו נבחר בגרסה LTS – ראשית תיבות של "גרסה לטוווח ארוך", ונבחר במערכת הפעלה שלנו – אם מדובר בחלוןנות, יש לנו installer נוח. מורדים, לוחצים על התוכנה שיורדת:



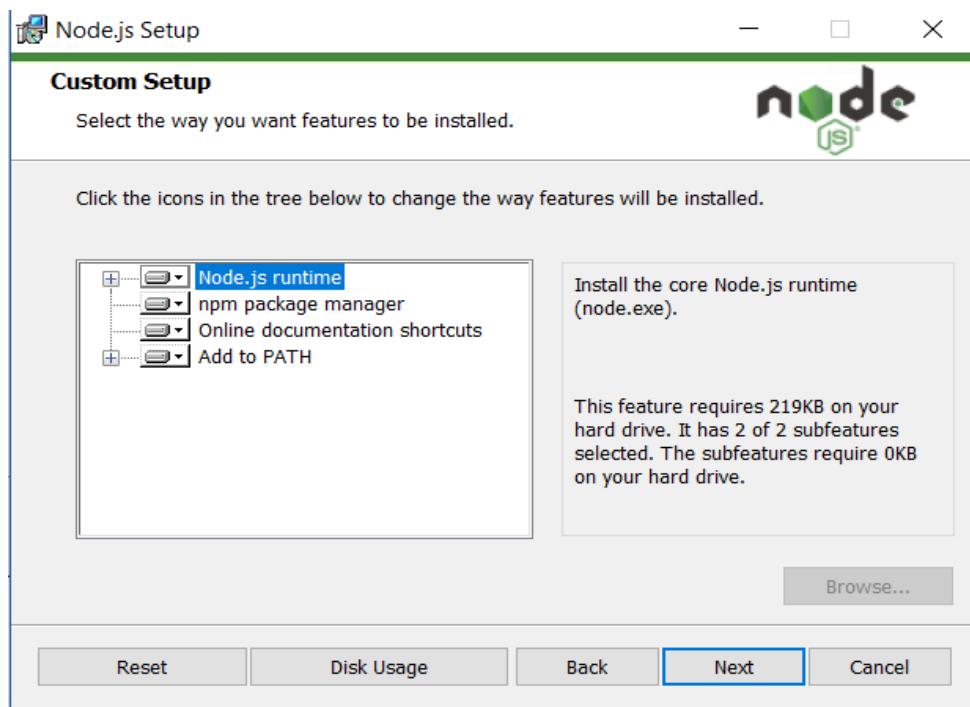
מקבלים את התנאים:



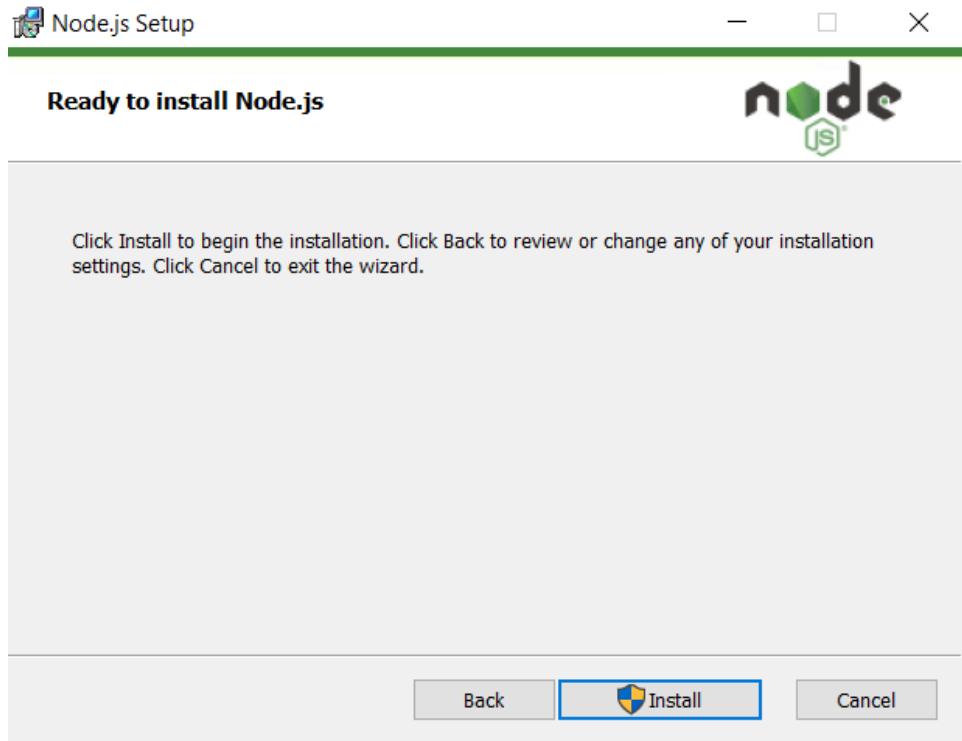
לחיצים על next:



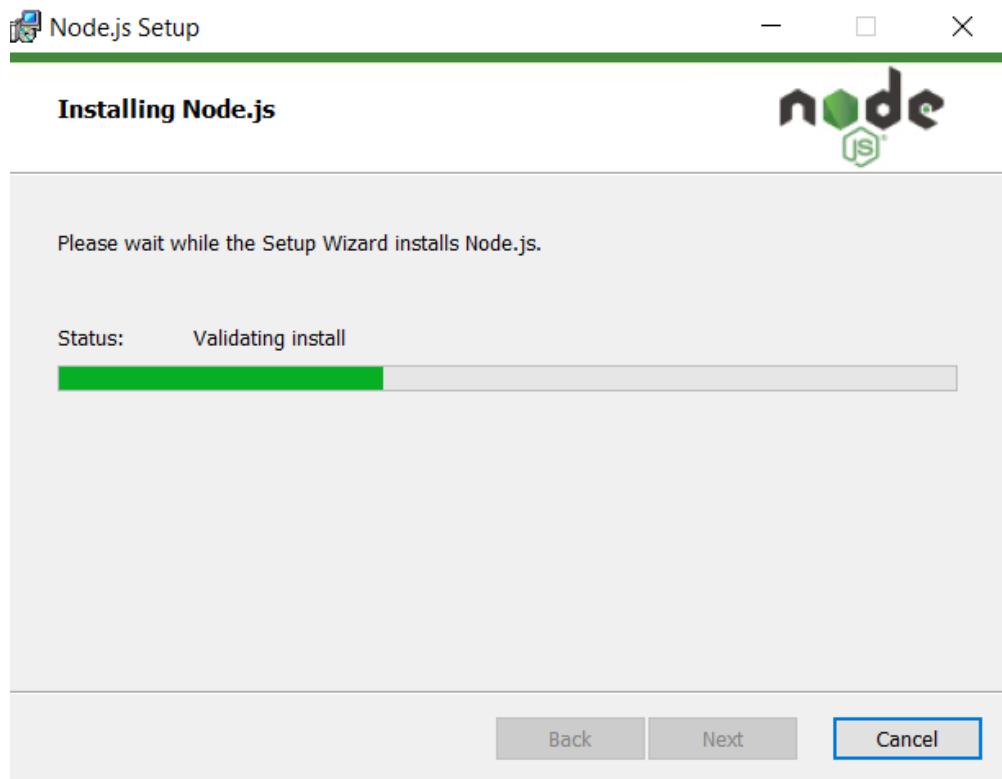
ושוב על next:



לחיצה על Install תתקין לבסוף את התוכנה:



כל מה שנותר הוא לחכות לסיום ההתקנה:



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 נראת את מספר הגרסה של .Node.js

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

C:\Users\barzik>
```

## התקנה על מז

ההתקנה של Node.js על מז היא פשוטה מאוד. נקליד בגוגל Node.js Download או ניכנו אל: <https://nodejs.org/en/download/>

אנו נבחר בגרסת LTS – ראשית תיבות של "גרסה לטוויה ארוך", נבחר במרק – ירד קובץ dmg שהוא אפשר להתקין כמו כל תוכנה אחרת בהנחה שהמחשב שלכם הוא לא מחשב ארגוני שמנע התקנות מהאינטרנט. ההתקנה היא פשוטה ביותר.

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

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

## התקנה על לינוקס

אם אתם משתמשים בדביאן, אז בדרך כלל ברוב ההפצאות sudo apt-get install node יטפל בהתקנה, אך אתם עלולים להתקין גרסה ישנה של Node.js זהה לעול להוות בעיה. למרות הפיתוי, היכנסו אל הקישור וקראו לפני ההתקנה את המדריך המלא לכל ההפצאות של לינוקס, שסביר על ההתקנות.

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

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

## תקלות נפוצות

זה נשמע מצחיק, אבל זה שלב הקשה ביותר שיש בכל למידת שפה חדשה, סביבה חדשה או כל' 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 מסטר התקלה או טקסט מהודעתת השגיאה וחפשו ברשת – סביר מאוד להניח שמשהו 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 שזה יצליח. אני מבטיח לכם ש-Node.js שווה את זה.

## כתיבת התוכנה הראשונה

נפתח תיקייה חדשה – למשל `node_projects` – ונכנס 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 נכתוב:

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

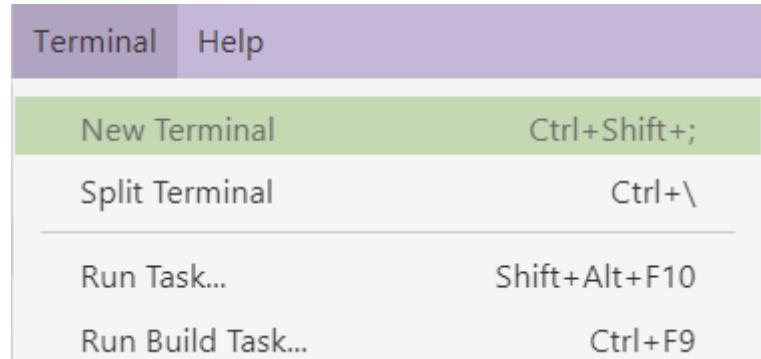
נשמר ואז נחזור לטרמינל ונכתב `node hello.js` או נראה שמודפס לנו המשפט

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

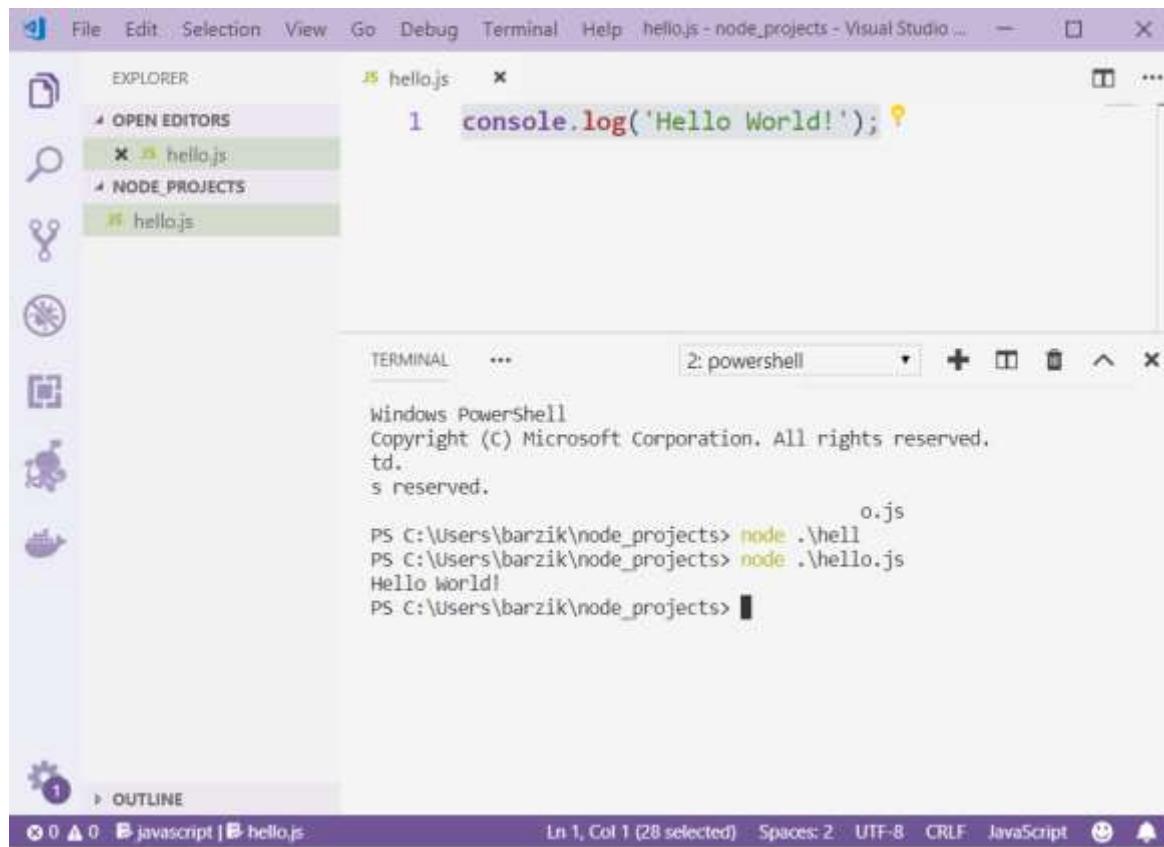
איזהipi כתבתם את תוכנית ה-Node.js הראשונה שלכם!

חשוב! אני יוצא מנקודת הנחה שאתם יודעים ג'אווהסקריפט וידיעים מה זה `console.log` ומה זה IDE ואפיו כבר מותקן לכם `Atom`, `WebStorm` או `Visual Studio Code`. אם החלטת לתרגם את התutorial מה-Node.js ל-Node.js, אומנם תקווה שיתרנו לך מושג את ה-IDE.

**הערה חשובה נוספת:** ב-`Visual Studio Code` וגם בסביבות העבודה האחרות הטרמינל מוגן ב-IDE. חפשו בתפריט העלון `Terminal` ולחצו עליו. בחרו ב-`New Terminal`. יפתח לכם בתחתית המסך טרמינל במיקום של הקבצים שלכם.



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



### תרגילים:

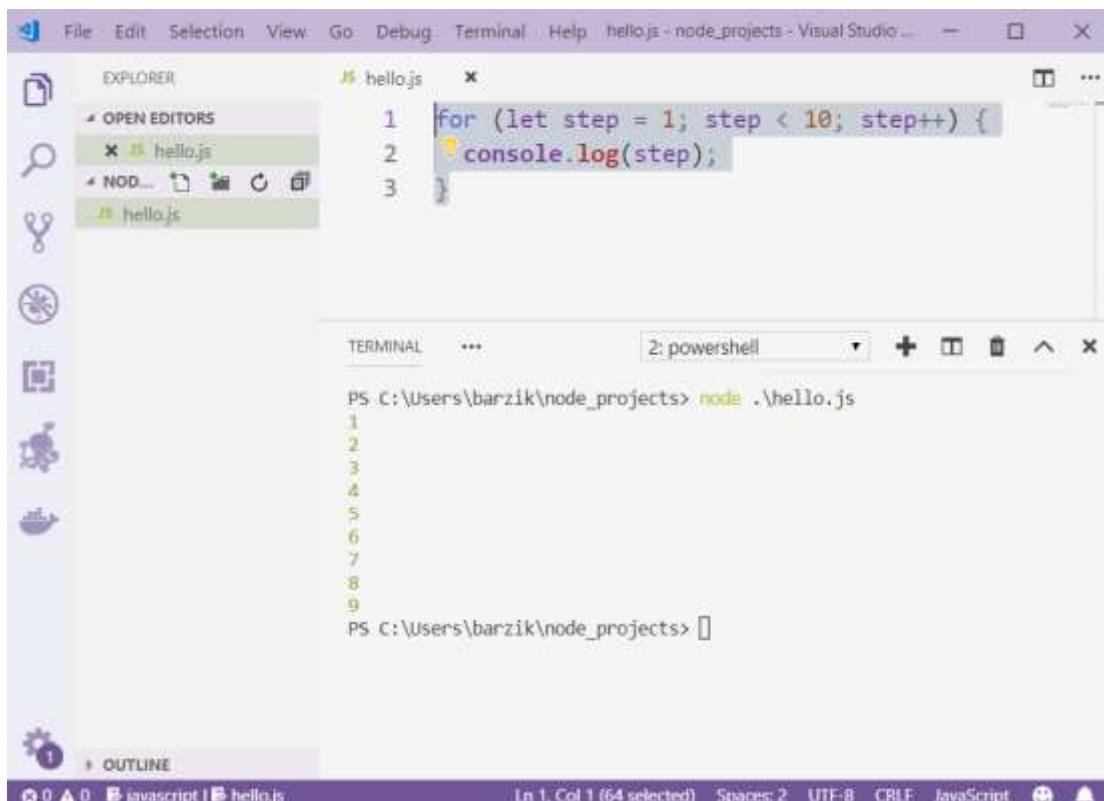
כתבו לולאה שרצה מ-1 עד 10 ומדפיסה את המספר של הולאה. הריצו אותה ב-node.js.

### פתרונות:

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

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

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



שימוש לב שהשתמשתי כאן ב-TAB. ההשלה האוטומטית הוסיפה את התווים \. שמסמנים "תיקייה נוכחת".

### תרגיל:

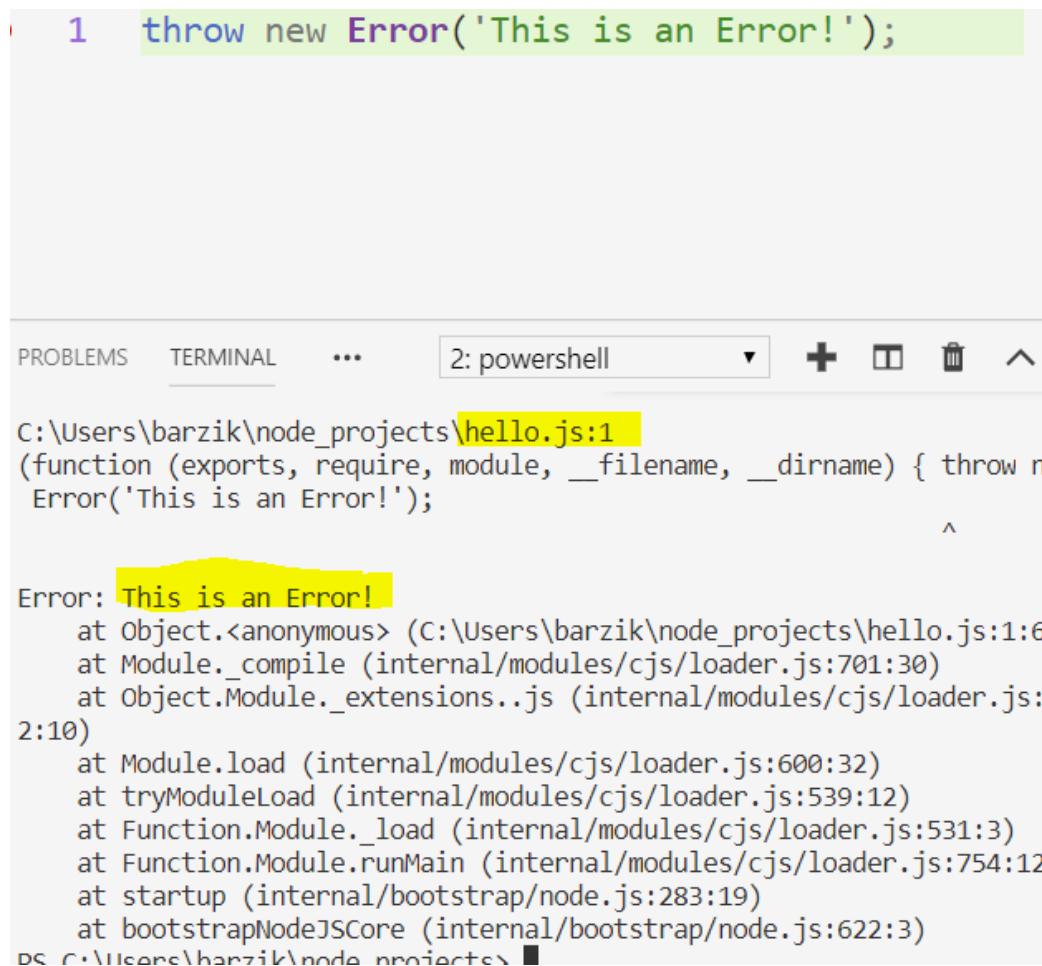
כתבו קוד שזורק הערת שגיאה. הריצו את הקוד.

### פתרונות:

בתיקית 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 התקינה, באמצעות הטרמינל במערכת הפעלה שלי או באמצעות הטרמינל ב-IDE שלי. אני מעדיף לווידאו שאני בתקינה שבה נמצא הקובץ וכותב node hello.js. אני רואה שגיאה וגם את ה-stack trace – השרשרת של הפקודות 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 new 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:710:10)
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)

במהלך הלימודים, אתם תראו את ה-stack trace של השגיאות לא מעת. הוא אמור לעזור לכם להבין איפה טויתם בהקלדה ואייפה שגיתם בסינטקס. אין מה להתבלבל או להיבהל. במערכות מורכבות הוא מסיע מאוד להבין מה הבעיה בדיק. כאן זרקנו שגיאה בקובץ אחד, אך נראה את המקור בשורה הראשונה. במערכות מורכבות יותר הבעיה האמיתית תופיע יותר בחתית. אבל העיקרון הוא אותו עיקרון – שגיאה נראית כר.

פרק 1

# REQUESTS וМОודולים REQUIRE



# Require ומודולים

הכוח הגדול של Node.js הוא חבילות התוכנה שלו. ל-Node.js יש יותר ממאה אלפי חבילות תוכנה שכל אחד יכול להשתמש בהן. איך משתמשים בהן? באמצעות `require`. אחד ההבדלים בין Node.js לבין ג'אווהסקרייפט בסביבת דפדן הוא `require`. הוא כמעט ייחודי ל-Node.js (יש ספריות נוספות שימושísticas בו, אך ללא ספק הוא סימן היכר שימושי של Node.js) ומשמש ל'יבוא' ולשימוש בחבילות תוכנה. ל-Node.js יש חבילות תוכנה שבאותו איתו כברירת מחדלanova ומשמש בהן.

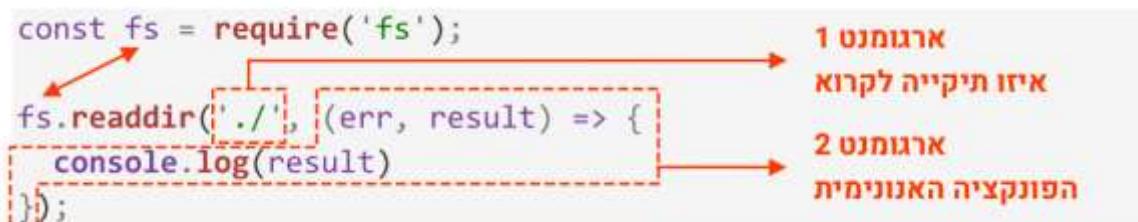
חבילת התוכנה הראשונה שאנו משתמש בה היא `fs`, שידועה גם כ-`system`. זוהי חבילת תוכנה שבאה תמיד עם Node.js (אי-אפשר להתקין את Node.js בלבד) ומשמשת לנו לטפל במערכות הקבצים של המחשב. יש לה מתודות רבות שמתפעלות את מערכת הקבצים. מתודה אחת שאחריה היא `readdir` – מתודה שמקבלת שני פרמטרים. הראשון הוא הנטייב של התיקייה שאנו רוצה לבדוק והשני הוא פונקציית קולבק. פונקציית הקולבק נקראת על ידי `readdir` בגמר הפעולה ומחזירה שני ארגומנטים – אובייקט שגייה (אם היא מותקנית) ואובייקט תוצאה שמציג את הקבצים והתיקיות שיש לנו בנטייב.

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

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

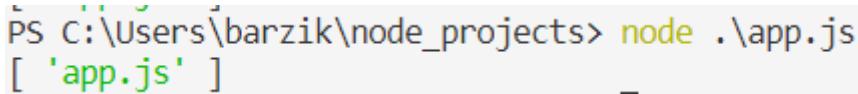
fs.readdir('./', (err, result) => {
  console.log(result);
});
```

מה הקוד הזה בעצם אומר? הדבר שלא אומר להיות ברור לתוכנת ג'אווהסקרייפט מן השורה הוא `require`. כאן אני בעצם קורא לחבילת התוכנה או לモול שנקרא `fs`. מדובר בעצם באובייקט כמו כל אובייקט שאנו מכירים, שיש לו מתודה שנקראת `readdir`. זוהי מתודה אסינכורונית שמקבלת ארגומנט ראשון של התיקייה שבה מחפשים וארוגמנט שני של קולבק. בקולבק יש שני ארגומנטים – אובייקט שגייה ואובייקט תוצאה. זהו פורמט סטנדרטי של קולבקים ב-Node.js ואני אראחים על כך בפרק על פרומיסים ב-Node.js.



כאמור, מתוכנת ג'אווהסקרייפט אמרור להבין איך קוד אסינכורוני עובד ואיך קולבקים עובדים. אם זה נראה לכם כמו סינית, זה הזמן לחזור על החומר.

אריך את האפליקציה שלי באמצעות `js.node app.js`. מה שאראה בקונסולה הוא את רשימת הקבצים שיש בתיקיה – במקרה זה `js.node app.js` בלבד.



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

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

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

אם תשמרו את הקוד הזה ב-`js.node` ותריצו אותו, תראו שנוצר קובץ בשם `test.txt`. אם תפתחו אותו, תראו שה תוכן הוא `hello world!`

אפשר לבצע `require` לכמה מודולים בו-זמנית. מודול נוסף שבא יחד עם `Node.js` הוא מודול `os`, שנוטן מידע על מערכת הפעלה. מתוודה אחת מתוך `os` היא מתוודה `os` – המתוודה הזאת לא מקבלת ארגומנטים, ומחזירה את `תיקיית` ה"בית" של מערכת הפעלה. אם אני למשל בחלונות, `תיקיית` הבית שלי היא `Users\barzik\c`. אם אני בلينוקס, `תיקיית` הבית שלי היא `/home/barzik`. אם אני כותב סקrypt של `Node.js`, אני רוצה שהוא יעבוד בלי קשר למערכת הפעלה ואני לא מעוניין לדעת מה היא. שימוש-ב-`os` הוא הדרך.

כך אכתוב את הקוד:

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

const homeDirectory = os.homedir();

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

אפשר לראות שפשות עשיית require ל-os והשתמשתי במתודה homeDir. מדובר במתודה סינכרונית שלא מקבלת קולבק, אז אין בעיה מהותית להשתמש בה. ה-require הוא לא קוסם או וודו אף. מדובר בקבלה של מודול, וברגע שקיבلت אותו אני יכול להשתמש בו בדיק כmo שאני משתמש בכל מודול אחר בג'אויסקייפט. כך למשל, אם אני רוצה ליעל את הקוד הקודם ולהוסיף שורה, אני יכול לבצע require ל-os וממיד לקרוא למתודה, וכן לחסוך משתנה:

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

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

אני לא חשב שמדובר בדוגמה שליל, אבל היא אמורה להבהיר לכם שלא מדובר בקסם. באחד מהפרקם הבאים אנו נראה מקרוב איך ה-require עובד כאשר נכתב מודול משלנו.

#### תרגום:

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

#### פתרון:

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

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

הדבר הראשון שאני עושה הוא require ל-fs. אני יוצר את הקובץ עם מתודת writeFile ואני מעביר לה שלושה ארגומנטים. ארגומנט ראשון הוא שם הקובץ שהוא אני יוצר, הארגומנט השני הוא hello world והשלישי הוא קולבק. פונקציית הקולבק נקראת אחרי שהקובץ סיים להיווצר. בתוכה אני אבצע קראיה 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

הפונקציה נמצאת  
בתוך הקולבק

פרק 2

# היכרות עם הדוקומנטציה של Node.js



# היכרות עם הדוקומנטציה של Node.js

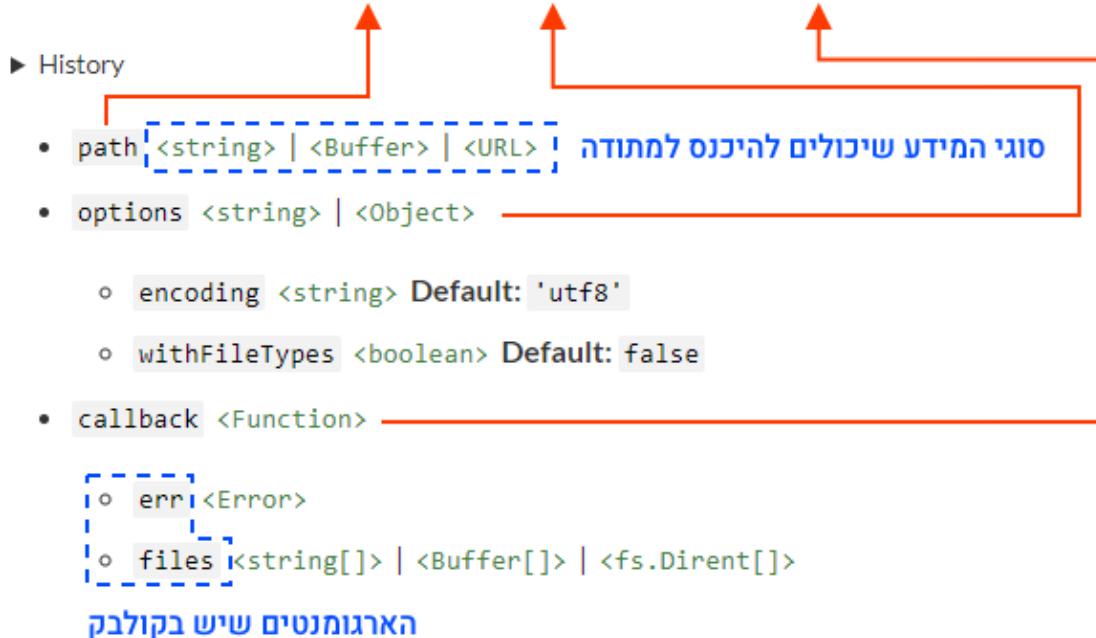
בפרק הקודם הסבירתי על המודולים של ברירת המחדל `fs` ו-`os`. מהיכן הכרתי אותם? הידע הזה לא בא לי בחלום אלא הוא כתוב בדוקומנטציה המפורטת של Node.js. בדוקומנטציה הזו יש פירוט של כל המודולים שבאים כברירת מחדל עם Node.js.

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

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 שאני יכול להעביר לה שלושה ארגומנטים. הראשון הוא `path`, השני הוא `options` שסבירו שיש סוגרים מרובעים כדי לرمץ על כך שהוא אפשרי ולא חובה. השלישי הוא הקולבק.

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

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

השלישי, הוא הקולבק שלנו. זהוי פונקציה (מקובל להזכיר פונקציית חוץ) שמשופעת לאחר שהפעולה מסתיימת. ככלומר ברגע ש-`fs.readdir` מסיימת לקרוא את תוכן התיקייה, היא לוקחת את הפונקציה שהעבירה ארגומנט שלישי ומפעילה אותה. כשהיא מפעילה אותה היא מאלסת את שני הארגומנטים בקולבק. אני יכול לטפל בקולבק ברגל ברגל ברגל 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' ]
});
```

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

באו לבדוק 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](https://nodejs.org/api/fs.html#fs_fs_readfile_path_options_callback) משהו 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 המתוודה `readdir`. גם כאן יש שלושה ארגומנטים. הראשון הוא `path` שיכל להיות מחרוזת טקסט (ויכל להיות גם סוג אחר), השני הוא אובייקט אפשרויות שהוא לא חובה, והשלישי הוא הקולבק. הקולבק הזה אמרור להיות מופעל כשהמתודה `readFile` מסיימת את תפקידה. היא תפעיל את הפונקציה הזו ותעביר אליה שני ארגומנטים, שגיאה (אם קיימת) ואת המידע.

כתבה של המתוודה הזו גם היא מזכירה מאוד את `readdir`. אני אזכיר קובץ בשם `js/app`, אכנים לתוכו את הקוד הבא:

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

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

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

אבל כשאני מפעיל את התוכנה הזו, אני רואה שהוא לא צפוי. במקומות לראות את כל הטקסט, אני רואה שהוא צזה:

```
<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 לעין בדוקומנטציה אני רואה שבאוף 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 אמור להיות תוכן הקובץ. אבל הלאה מוסבר שאם לא מפורט שם `encoding`, אנו מקבלים `Buffer` זהה בדיק מה שקיבלנו. איך אנו בעצם פותרים את הבעיה? גם זה מוסבר בדוקומנטציה (ואפיו יש דוגמה). פשוט להعبر קידוד:

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

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

ההרצה של הקוד הזה כבר תציג לי את תוכן הקובץ כמו שאני מyps להו:

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

### תרגיל:

אתרו בדוקומנטציה את המתודה `fs.mkdir` והשתמשו בה בסקריפט של Node.js על מנת ליצור תיקייה 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](https://nodejs.org/api/fs.html#fs_fs_mkdir_path_options_callback)

אם נסתכל עליה נראה שהוא זהה למתודות של `readFile` ו-`readdir`. גם היא מקבלת שלושה ארגומנטים:

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

► History Argument three Argument two Argument one Callback [No callback]

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

ARGINENT ראשון הוא המיקום שבו רוצים ליצור את התיקייה החדשה, ARGUMENT השני הוא אפשרויות, והARGINENT השלישי הוא הקולבק שנקרא לאחר השלמת הפעולה. הקולבק זה מקבל אריך ווק אובייקט שגיאה אם הפעולה נכשלה. כדאי לשים לב שהוא תמיד מופעל לאחר הצלחת הפעולה ואם אין שם אובייקט שגיאה, אני יכול להניח שהפעולה הצליחה.

כרגע כותב את הסקריפט. בחרתי לכתוב אותו בקובץ `fs.js` בתיקית העבודה שלי.

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

fs.mkdir('./Hello', (err) => {
  if (err) throw err;
```

```
  console.log('Directory created!');
});
```

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

#### תרגיל:

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

#### פתרון:

גם כאן קל להשתמש בדוקומנטציה של `js.Node` על מנת לאתר את המתודה הזו. 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 ורך אובייקט שגיאה אם הפעולה נכשלה.



הקוד שלי יראה 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 שיצרתי קודם תימחק.

פרק 3

# גושאות סינכראניות למטריות אסינכראניות



## גרסאות סינכרוניות לMETHODS אסינכרוניות

כתבתי בתחילת הספר ש-`Node.js` היא אסינכרונית וזו הכוח שלה. `Node.js` רצה על הליך אחד במאובך ואנו נדרשים לפעולה מסוימת כמו קריית קובץ, התוכנה לא עומדת וממתינה לקובץ זהה, אלא ממשיכה הלאה. אם אני כותב משהו כזה למשל:

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

console.log('Before readdir');

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

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

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

מה שאני רואה בטרמינל הוא זה:

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

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

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

ואיפה מוצאים את הפעולות הסינכרוניות האלו? בדוקומנטציה כמובן! אם חיטטתם בדוקומנטציה, הייתם יכולים לראות שיש METHODS שזהות לMETHODS שאוון תרגלונו, אבל מוצמד להן `Sync` לשם. METHODS האלו נמצאות בעיקר בMODULE `File System`. כך למשל, יש לנו `readdir` ויש לנו `readdirSync`.

לMETHODS סינכרוניות אין קולבק והן מוחזירות את התוצאה שלهن בדומה לקולבק. כך למשל `readdirSync` תיראה כך:

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

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

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

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

console.log('Before readdir');

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

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

זה מה שאני אקבל בהרצה:

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

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

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

console.log('Before readdir');

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

התוצאה של הרצת הקוד זהה תהיה:

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

וכמובן הסקריפט לא יתפוצץ עם שגיאה ו-stack trace, אם לא יהיה .try-catch. כמעט לכל מתודה שטפלת בקבצים יש הגרסה סינכרונית שלה. הינה רשימת המתודות שלמדו עד כה:

תיאור המתודה	גרסה אסינכורונית	גרסה סינכרונית
יצירת תיקייה	<code>fs.mkdir(path[, options], callback)</code>	<code>fs.mkdirSync(path[, options])</code>
קריאה תוכן תיקייה	<code>fs.readdir(path[, options], callback)</code>	<code>fs.readdirSync(path[, options])</code>
מחיקת תיקייה	<code>fs.rmdir(path, callback)</code>	<code>fs.rmdirSync(path)</code>
קריאה קובץ	<code>fs.readFile(path[, options], callback)</code>	<code>fs.readFileSync(path[, options])</code>
יצירת קובץ	<code>fs.writeFile(file, data[, options], callback)</code>	<code>fs.writeFileSync(file, data[, options])</code>

מאוד לא מומלץ להשתמש בגרסאות סינכרונית אלא אם כן אתם יודעים מה אתם צריכים — משתמשים בהן בדרך כלל לשימושים ייחודיים. מפתחה מודע, במיוחד אם לא סגורים עד הסוף על האסינכרוניות להשתמש בקוד זהה. אבל זה עלול להיות הרסני במקרים מסוימים כמו שרטטים.

אם אתם לא יודעים אסינכרוניות וקובוקים היטב — זה הזמן לבצע חזרה על כך. קובלוקים הם לא ייחודיים ל-`js.Node` ולא נלמדים בספר זה אלא נלמדים בספרים המלמדים ג'אווהסקריפט מאפס. בהמשך הפרק נלמד דרכי נוחות יותר לכתיבת קוד אסינכרוני, עם פרומיסים או עם `Async-Await`. שnochות בדיקן כמו קוד אסינכרוני. אבל כך או כך — `js.Node` לא נכתב כקוד סינכרוני.

#### תרגילים:

צרו מחרוזת טקסט רנדומלית עם:

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

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

#### פתרונות:

ראשית אנו צריכים לארח את המתוודות של File System שנשתמש בהן. במקרה זה, `mkdirSync` שמשמשת ליצור תיקייה `-rmdirSync` שמשמשת למחיקת תיקייה. אני יכול להתבסס על הידע המוקדם שלי או לבדוק אותו בדוקומנטציה ולראות איך הוא עובד. במקרה זה הן פשוטות ומקבלות ארגומנט אחד – שם התיקייה. כל מה שאנו צריך זה לקבל את שם התיקייה ולהוסיף אותו למיקום היחסי /. של הקובץ שלי. זה נראה כך:

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

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

fs.mkdirSync(`./${randomString}`);
console.log(`>${randomString} Directory Created!`);

fs.rmdirSync(`./${randomString}`);
console.log(`>${randomString} Directory Deleted!`);
```

כדי לשימוש לב שני משלוחים פה בתבנית טקסט (הגרש העקום – backtick) כדי להציב משתנה בתוך מחרוזת טקסט.

#### תרגילים:

בצעו את התרגיל הקודם בעזרת פונקציות אסינכרוניות.

#### פתרונות:

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

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

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

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

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

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

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

```
const fs = require('fs');  
  
const randomString = Math.random().toString(36).substring(7);  
  
fs.mkdir(`${randomString}`, (err) => {  
  console.log(`${randomString} Directory Created`);  
  
  fs.rmdir(`./${randomString}`, (err) => {  
    console.log(`${randomString} Directory Deleted`);  
  });  
});
```

הראשון  
נעשית בתוך הקולבך  
תתקייה

קולבך הראשון  
מופעל לאחר יצירת  
תתקייה  
  
קולבך שני  
מופעל לאחר מחיקת התיקייה

פרק 4

# PACKAGE.JSON הנדשה ראשונית והפעלה של NPM



# package.json – הכרה ראשונית והפעלה של NPM

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

כל האצבע של `js.Node` הוא שימושיים דברים ללבת המערכת רק כשהם מוסףים יכולת לפלטפורמה. אם אני רוצה להשתמש ב-`js.Node` לדברים מתקדמים יותר, אני יכול להשתמש במודולים של ברירת המחדל כדי לבנות את הפונקציונליות הזו, אבל זה ייקח לי הרבה זמן.

במקום זה, אני יכול להיעזר במודולים מורכבים שאנשים יצרו על בסיס המודולים הבסיסיים וכן לחסוך זמן רב. זהו הרעיון שעומד בסיס תרבות הקוד פתוח או המערכת האקוולוגית, האקויסיטם, של `js.Node`. יש המון מודולים בקוד פתוח שכל אחד יכול להשתמש בהם לсистемת שלו.

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

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

כתובת האתר היא [npmjs.com](https://npmjs.com) ואם תיכנסו אליו תוכלו לראות שיש שם יותר ממאה אלף חבילות תוכנה. אבל מתוקן יש רק כמה אלפי חבילות תוכנה פופולריות. אפשר לחפש באתר ולבחן את המודולים השונים.

NPM אינו רק מאגר של תוכנות אלא גם כל שמייע לנו לנצל את החבילות שאנו משתמשים בהן בפרויקטים שלנו.

הכל הזה מותקן אוטומטית יחד עם `js.Node`. אם התקנתם אותו כמו שהסבירתי בפרק הראשון, תוכלו להשתמש בו כתעת. היכנסו לטרמינל שלכם (למדנו עליו באחד הפרויקטים הקודמים) ונוטו אל המיקום שבו נמצאת תיקיית הפרויקט שלכם. אפשר לעשות את זה מתוך `Visual Studio Code` או `npm`. כתבו 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 – כרשמי משתמש בה יכול גם הוא להתקין את כל המודולים שבחרנו להשתמש בהם בקלות רבה באמצעות פקודה אחת. הוראות התקינה האלו נמצאות בקובץ שנקרא `package.json`, שהוא אנו נדרש ליצור בתיקייה הראשית של כל פרויקט NPM.

בקובץ זהה יש לא מעט דברים אבל בראש ובראשונה יש בו רשימות של המודולים השונים שנותן משתמש בהם והגרסאות שלהם. כרשמי מתכוון אחר ירצה להשתמש בפרויקט שלנו, הוא לא נדרש לנחש באילו מודולים של NPM השתמשנו אלא תへיה לו רשימה מסודרת שלהם (עם מספר הגרסה שלהם, כפי שנראה בהמשך) ודרך להתקין אותם.

## יצירת `package.json` לפרויקט שלנו

יכנסו לטרמינל שלנו ונכנסו אל התקייה הראשית של הפרויקט שלנו. נקליד `npm init` ומקח ואז נקיש על אנטר. מיד תופיע לנו רשימה של שאלות. אנו יכולים ללחוץ אנטר כדי לחת את תשובות ברירת המחדל. לשאלות האלו יש משמעות שנдан 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 לפתח 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 אחסון מידע שמצויר את הפורמט של אובייקט בג'אווהסקרייפט, למעט כמה הבדלים פשוטים (למשל, שדות של אובייקטים חיברים להיות בין גרשים). משתמשים בו בדרך כלל לקובץ קונפיגורציה. בספר הלימוד "לימוד ג'אווהסקרייפט בעברית" יש הסבר על פורמט הנתונים זהה ומתכנתה ג'אווהסקרייפט אמורים להכיר אותו. הוא בניי כמו אובייקט ג'אווהסקרייפט אבל התוכנות שלו (למשל `name`, `version` וה-`description`) מוקפות במירכאות כפולות. גם מחרוזות הטקסט. כמו בג'אווהסקרייפט, גם ב-JSON תכמה יכולה להכיל אובייקט או מערך. פורמט הנתונים זהה קל לקרוא ומשתמשים בו ללא כמעט מוקומות, הן לקונפיגורציה של מערכת והן לצרכים אחרים.

כרגע אנו לא רואים שום רשיימה של מודולים, אבל זה די קל — אין מודולים חיצוניים שהותקנו, אז אין לנו רשיימה. מה שיש לנו הוא קובץ הגדרות פשוט שבפושוטים — זה הכל. מה שצורך לזכור זה שקל לייצר את הקובץ זהה ובכל פרויקט Node.js הוא קיים.

## התקנת המודול הראשון

אנו נבחר במודול הפופולרי ביותר ב-NPM, הלא הוא `lodash`. הוא נמצא בכתובת זו: <https://www.npmjs.com/package/lodash> אם תיכנסו אליו, תוכלו לראות הוראות התקינה, אבל הוראות התקינה של מודול זה פשוטות. מקלידים `npm install lodash` 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 הזו בטרמינל במקומם שבו יצרתם את ה-`json.package`, שזו התיק'יה 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 שנוספה לכמ' תי'יה בשם `node_modules` ובתוכה יש תי'יה נוספת שנקראת `lodash`. בתיק'יה `node_modules` אמרוים להיות המודולים החיצוניים שאותם משתמשים בהם, והם בלבד. זו תי'יה שומרה לכך. מאד לא מקובל לבצע שינויים ידנית על התוכן שנמצא בתוך התיק'יה הזו. מי שמבצע שם שינויים זו אף רוק NPM, באמצעות סט פקודות שקיים בה. לעיתים המודולים החיצוניים שלכם משתמשים במודולים חיצוניים אחרים. NPM מטפלת בהורדה שלהם וגם הם יופיעו בתיק'יה הזו.

שינוי נוסף שהתרחש הוא ב-`json.package` – 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.ណון על כך בהמשך.

אם אתם מעלים את התוכנה שלכם לאנשוהו, מעבירים אותה לחבר או שומרים אותה בGITהאב, לא מעלים יחד עם התוכנה את תי'יה `node_modules` אלא רק את `json.package`. 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 הזו מפעילה את מוקח שניגש ל-`json.package` ומוריד את כל המודולים שמפורטים ב-`dependencies` היפיך מ-`NPM`. זו הסיבה שמאוד חשוב להקפיד שככל המודולים החיצוניים שאתם משתמשים בהם מופיעים ב-`json.package`.

## 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 זהה – **reverse**. מה שהמتدודה עשוה הוא פשוט להפוך מערך. אם יש לנו מערך שהוא [1,2,3] היא תהפוך אותו ל [3,2,1] – די פשוט ונחמדה.

איך השתמש בו? נבצע לו `require` בדיק כmo מודול בירית חדש. עם השם שמויע ב-`hosj.js`:  
במקרה זהה `lodash`:

```
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 לחילוטין של משתנה. כיוון `lodash` הוא מודול פופולרי מאוד בתעשייה, זו הקונבנצייה לשימוש במודול הספציפי זהה. חשוב לציין שברוב החבילות לא מקובל להשתמש בסימנים מיוחדים בודדים. אם זה מאוד מפריע לכם יכולים להשתמש בשם אחר. איך משתמשים במודול זהה? בדיק כmo במודול של `fs` שלוינו למדנו קודם. ממש קל.

בואו נדגים עם מודול אחר. הפעם בחרתי במודול `chalk` שנמצא בכתובת [https://www.npmjs.com/package\(chalk\)](https://www.npmjs.com/package(chalk)) – אנו נראה שיש הוראות התקינה שם, אבל אנו לא צריכים אותן. לפי כתובת המודול אני רואה ששמו הוא `chalk`. אנווט לתיקיה של הפרויקט שלו במכשיר הטרמינל (או באמצעות הטרמינל של `Visual studio code`) ואכטבו:

```
npm i chalk
```

```
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
```

אחרי שאלחץ אנטר ואמתין שהפעולה תושלם, אני אראה שני דברים. ראשית, שבקובץ `hosj.js` מתווסף המודול `chalk` ומספר הגרסה שלו. באחד הפרקים הבאים אפשרי 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 יכולCut להשתמש בchalk.

פתח את הקובץ `sz.app` ורכניס אליו את הקוד הבא:

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

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

```
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 שיקרה פה הוא שגם lodash וגם chalk יעבדו במקביל, בלי בעיה.  
שימוש לב: מקובל להציב את כל ה-require בתחילת הקובץ שעליו עובדים.

בסוף דבר מודולים הם לא קיטם, הם לא וודו – גם הם קוד של Node.js שנכתב על ידי מתכנתים בשර ודם שהשתמשו במודולים אחרים או במודולים שבאים עם Node.js. אנחנו גם יכולים לכתוב, מאפס, מודולים שצובעים את הפלט של הקונסולהocab בעים מרהייבים או הופכים מערך. בסופו של דבר זה ג'אויסקריפט. אבל למה לנו? למה לי לכתוב שוב ושוב אותם מודולים שעושים אותם דברים כשאני יכול להוריד אותם בקלות ולנحال אותם עם NPM?

ככה עובדים עם מודולים חיצוניים זהה הכוח של NPM ושל האקואיסיטם של Node.js – שפע של מודולים חיצוניים שמאפשרים המון דברים ויכולת מדהימה של ניהול שלהם.

יש מנהל מודולים נוסף ל-Node.js, שנקרא yarn, והוא עובד בצורה זהה. אני לא מלמד אותו בספר זה. אפשר לקרוא עליו באתר שלו: [yarnpkg.com](http://yarnpkg.com)

**תרגילים:**

השתמשו במודול החיצוני `request` בכתובות `request` על מנת לבצע קריאה לעמוד הראשי של מנוע החיפוש גוגל ולקבל את ה-HTML שלו בקונסולה.

**פתרונות:**

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

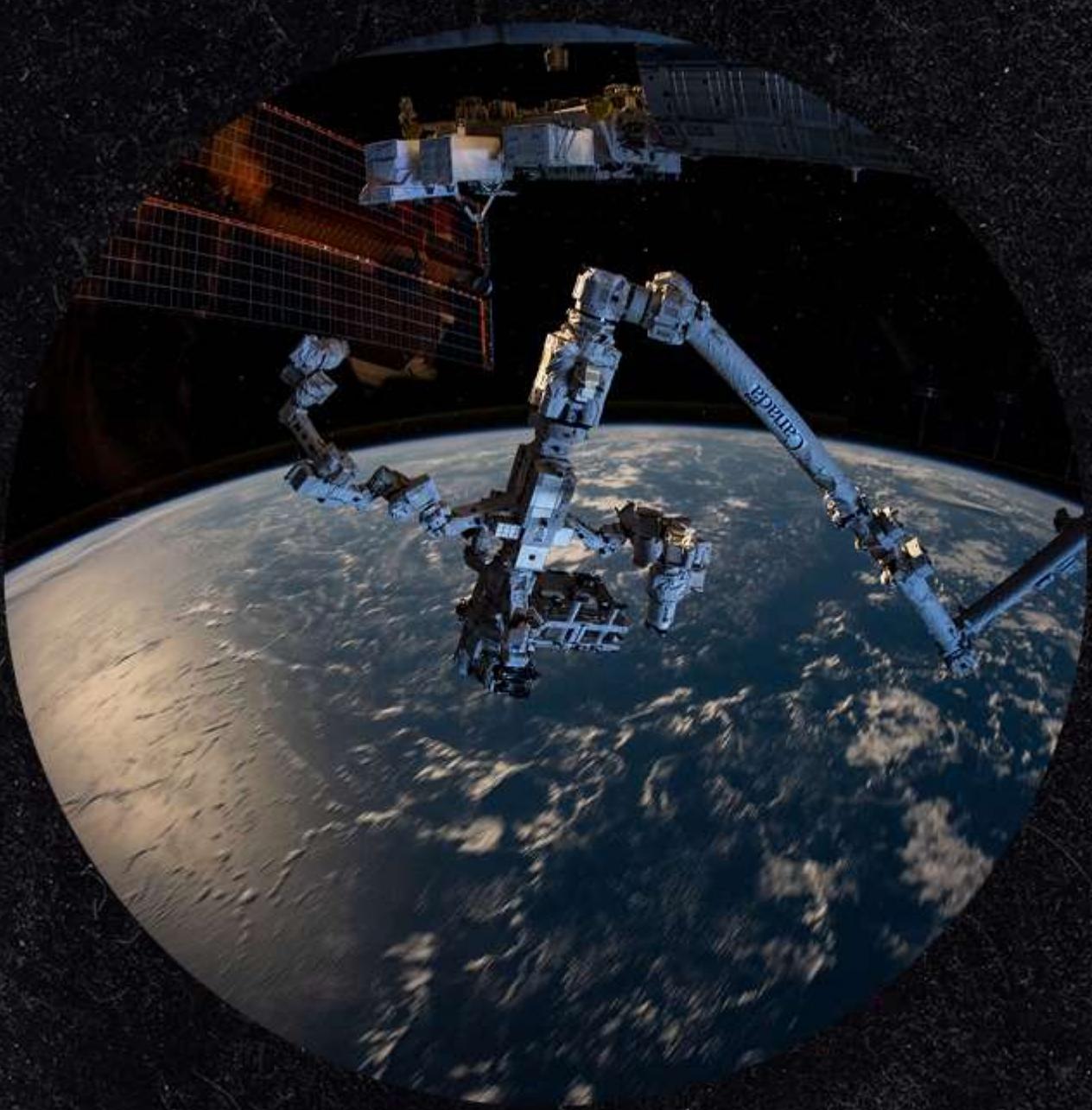
```
const request = require('request');

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 `node app.js` באמצעות הקלהה של `node app.js` בטרמינל (כאשר אנו נמצאים במקום של הפרויקט שלנו כמובן) ונראה את התוכנה עובדת.

פרק 5

# עובדת אסינכרונית ומטריך מהולבקים לפומיסים ול-`ASYNC/AWAIT`



# עבודה אסינכרונית ומעבר מ`колבוקים` ל`פרומים` ול-`async/await`

עד כה רأינו שאנו עובדים עם `kolbokim` פשוטים ב-`js.Node`. אבל כל מתכנת ג'אווהסקריפט יודע `kolbokim` זו דרך מסורבלת לעובדה עם קוד אסינכרוני. ב-`js.Node` יש דרכים טובות יותר מאשר `kolbokim` ואני נלמד בפרק זהה כיצד להסביר את הקוד שלנו והמודולים שלנו לעובדה עם `prumim` ועם `async/await`.

בספר זה אני לא מלמד על `kolbokim`, `prumim` 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](https://developer.mozilla.org) (שידוע ECMAScript 6-המודולים של `js.Node` כרגע באופן אסינכרוני עם `kolbokim` בלבד. כ-`Node` ES2015) יצא עם יכולת האסינכרונית של `prumim`, רבים רצו לעבוד איתם ועם המודולים שבאים עם `js.Node` כברירת מחדל, אבל לא היה אפשר. זו הסיבה שמתכנתים רבים עבדו עם מודולים חיצוניים כמו `birdsnest`, שהמירו את המודולים הסטנדרטיים של `js.Node` לעובדה באמצעות `prumim`. מ-`js.Node` גרסה 8 יש לנו דרך לעשות את זה עם מודול של `js.Node` וזה הדריך הפשטה והטובה לעבוד עם המודולים של `js.Node` ו`prumim` כרגע.

המודול שבו משתמשים ב-`js.Node` על מנת להסביר את המודולים שלו למודולים שתומכים ב`prumim` הוא `util`. מדובר במודול שכשמו כן הוא – מכיל המונח `util` עזר. אחד מכל העזר הוא מתודה בשם `promisify` שמקבלת כארגומנט מתודות אחרות של `js.Node`. המתודה הזאת מסייעת לנו לקחת מודול שמשתמש ב`kolbokim` ולהמיר אותו למודול שמשתמש ב`prumim`.

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

הינה דוגמה עם `readdir`. אפשר לראות שזה די פשוט:

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

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

אני קורא למודול `util` ולמודול `fs` עם `require` כרגע. בשורה השלישי קורא הנוס – אני בעצם עוטף את המתודה שאני רוצה להשתמש בה, כמו `fs.readdir`, במתודת `util.promisify` ומוחזיר את התוצאה לקבוע בשם `readdir` (אני יכול לקבוע איזה שם משתנה שאני רוצה, כמובן). מעכשיו הקבוע הזה הוא בעצם `fs.readdir` שלו ואני יכול להשתמש בו לכל דבר ועניין, בהבדל פשוט אחד – אני לא מעביר לו קולבך אלא מקבל ממנו פרומיס ועובד איתו כמו כל שירות שמחזיר לי פרומיסים.

זה יעבוד רק על מethodות של מודולים שמקבלים קולבך בהתאם הארגרמנט האחרון ובkulback יש `error` ו-`value` של חזרה, כמו `fs.readdir`. להזכירם, `readdir` נראה בדיקות כהה:

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

ולפיכך הוא ממש מתאים ל-`util.promisify`. הסטנדרט של קולבך אחרון נחשב סטנדרט מקובל מאד בתעשייה ולפיכך יתאים לרוב המודולים שיש ב-NPM.

ובמקרים שיש פרומיסים, יש גם `it` `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 איני אكتب אותו מהנקודה הזאת ולהלאה. אם הקוד הזה לא מוכך לכם, זה הזמן לעבור על `it` `async-await` – דרך מומלצת ומודרנית לכתוב קוד אסינכריוני בג'אוויסקורייפט.

מה קורא אם הקולבך מוחזיר כמה ערכים ולא רק ערך אחד? אdegים בעזרת המודול `DNS` שהוא מודול בירית מוחדר ב-`js.Node` ויש לו מетодה `shnkrat` `lookup`. המתודה הזאת מבצעת פעולה שנקראת `dns` `lookup` – פעולה שמחזירה את כתובת ה-IP מאחורי שם מתחם מסוים ואת הסטטוס שלו. הקולבך

שהmethod dns.lookup מפעילה מחריר שני ארגומנטים: הראשון הוא address והשני הוא family. מה קורה כאשר מושה לה ?promisify

```
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 במקומות ערך, חזר 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 את ה-IP של node.js, תיצור תיקייה מكتوبة ה-IP הזו ותדפיס את כל תוכן תיקיית האב.

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

```
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: `dns.lookup`, `fs.mkdir` וועטף אותן ב-`util.promisify` ואז אני יכול לעבוד איתן עם פרומיסים או עם `async-await`. במקרה זהה אני בוחר לעבוד איתן באמצעות `async-await`. הקוד הרבה יותר קרייא ונוח.

שיםו לב שהה נכתב כמו קוד סינכרוני אבל מתנהג כמו קוד אסינכרוני אמיתי. במקרה הזה כל מה שיש בפונקציית `init` זה תלוי זה בזה, אבל אם היו פונקציות אחרות, הן היו רצויות גם כן בזמן שהיא שি�ש ב-`init`. היה מוכחה לתשובה ממערכת הקבצים או מהרשף.

## מודולים ב-`js.Node` שתומכים בפרומיסים באופן טבעי

ב-`js.Node` החלו להכני תמיכה טבעית בפרומיסים במודולים שלהם גם ללא `promisify` או תוספות מלאכותיות. איך? באמצעות תכונת `promises` שנמצאת במודולים התומכים בפרומיסים. כשאנו עושים `require` לモודול התומך באופן טבעי בפרומיסים, נעשה אותו כך:

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

ואז אני אוכל להשתמש בו ממש כאילו עשיתי לו `util.promisify`. לשם הדוגמה, הקוד הקודם יהפוך להרבה 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 מדובר בפייצר ניסיוני.

**תרגילים:**

מודול DNS מכיל מתודה בשם `reverse` שנמצאת במקום זה בדוקומנטציה: [https://nodejs.org/dist/latest-v8.x/docs/api/dns.html#dns\\_dns\\_reverse\\_ip\\_callback](https://nodejs.org/dist/latest-v8.x/docs/api/dns.html#dns_dns_reverse_ip_callback)

המתודה הזו 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.

**פתרונות:**

```
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');
```

ראשית משכתי את המודול `utils` ואת המודול `dns`. עכשו אני יכול לעתוף את `dns.reverse` ב-`Promise` ואני יכול להשתמש בו ב-`it-async`. אוצר פונקציה אסינכרונית שמקבלת פרמטר שהוא IP ואשתמש במתודה העטופה כרגע. היא תחזיר לי את התוצאה כמו פונקציה רגילה ואוותה אוכל להדפיס בקונסולה. השו לשימוש בקולבקים:

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

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

פרק 9

# אידוניים



## אירועים

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

יש מוצרים שימושיים בפרומים כדי לתקשר ויש כאלה שימושיים באירועים. כל מערכת והצריכים שלה ושל מי שתכנן אותה. מערכות מבזורת יותר, שיש להן תוספים או מודולים חיצוניים, נוטות להשתמש באירועים. מערכות אחרות, במיוחד לקריאה ולכתיבת, נוטות להשתמש בפרומים.

המודול `events` עובד באופן שונה מהמודולים שאנו מכירים. הוא קל-aos שמכיל מתודות. אנו יכולים להשתמש במתודות האלו לשירות או לרשת ממנה – כלומר לבצע לו `extend`. אנו נלמד על הדרך השנייה – ירושה. אם אתם מכירים את ג'אווסקריפט לעומק, אתם בוודאי יודעים שהירושה הזו היא לא ירושה "אמתית" אלא פשוט סוכר סינטקטי מעלה אובייקט של ג'אווסקריפט. כך או כך, בוגד למודול רגיל שבו אנו עושים `require` ומשתמשים במתודות שלו באופן סטטי, כאן אנו אמורים ליצור את הקלאס שלנו ואז לרשת את `event`. ברגע שאנחנו עושים את זה, הקלאס שלנו, או המודול שלנו אם תרצו – מקבל את כל המתודות של מודול `events` ויכול לעבוד עם אירועים. בואו ניצור קלאס פשוטו, ניתן 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 שאנו עושים הוא לבצע `require` ל-`events`. מקובל לעשות `require` לתוך `EventEmitter`, למolute שאפשר לעשות לכל שם משתנה שנבחר. בוגד למודולים אחרים, אני לא משתמש לשירות ב-`events` אלא ירש ממנה בקלאס שלי, בקרה זהה `MyClass`. זהו קל-aos פשוט מאוד שאין בו כלום. אבל ברגע שאנחנו ירש מ-`events`, אני מקבל את כל המתודות שיש בו. המתודות מופיעות בדוקומנטציה:

[https://nodejs.org/api/events.html#events\\_class\\_eventemitter](https://nodejs.org/api/events.html#events_class_eventemitter)

איך מתקדים מכאן? עכשו אני יכול לבצע אימפלמנטציה להאזנה לאירוע פשוט. איך? אני משתמש במתודת חס, שכאמר נמצאת בקלאס שלי אחריו שירשתי מ-`events`, ו"נרשם" לאירוע. אני בעצם אומר

— שמע נא, js.Node חמוד, ברגע שיש אירוע בשם someEvent, תפעיל את הקולבק זהה. מה יש בקולבק? בקוד שלעיל סתם console.log(). אבל זה יכול להיות הרבה יותר.

במה שקדם אני פשוט מפעיל את האירוע באמצעות הפונקציה emit. הפעלת האירוע גורמת לקולבק להתקיים:

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

class MyClass extends EventEmitter { }

const myClass = new MyClass();

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

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



מתודת ה-`on` מקבלת שני ארגומנטים: הראשון הוא שם האירוע שני נרשם אליו והשני הוא הקולבק המופעל.

האירוע יכול להיות מופעל מכל מקום, לא רק מבחן! בדוגמה זו אני מכניס את ה-`emit` בתוך מתודה של הקלאס ומפעיל אותה באמצעות המילה השמורה `this` – שהוא רפנס לאובייקט עצמו. אני מבצע הפעלה של אירוע `this.emit`. בתוך הקלאס הוא שווה ערך ל-`myClass.emit` – פשוט אחד מבפניים 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 `this`. 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 להציג לארועים עם `on` מכל מקום ולהפעיל אותם מכל מקום שיש בו רפנס, מבנים או מבחוץ. אירועים/events דומים באופן כמעט זהה לארועים בג'אוויסקריפט בדף, אבל השינוי העיקרי פה – כיוון שאין דף – הוא שהוא יוצרים את האירועים בעצמם. קלומר לנו יוצרים את המazziים לארועים שעובדים כשהם קורים גם, במידת הצורך, את האירועים עצם.

לסיום אני אציג דוגמה נוספת, שבה גם הפונקציה המטפלת באירוע ושהותה אני מצמיד למאזין במאזין:

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

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

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

const myClass = new MyClass();
```

וכך בדרך כלל תראו מאזורים בתוך מודול. מובן שכל אחד מחוץ למודול יכול להתחבר לאירוע. פונקציות שמתפלות באירוע נקראות בדרך כלל handler ומאוד מקובל להציג את המילה handler לשם פונקציה שמתפלת באירוע.

## כיבוי מאזור

כמו שהצמדנו קולבק לאירוע, ככלומר גרמנו לו להיות מאזור באמצעות חס, אנחנו יכולים לכבות אותו באמצעות off. זה חשוב אם אנחנו לא צריכים את המאזור הזה יותר. עושים את זה באופן פשוט 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 הזה אני יוצר מאזור. בדיקן כמו בדוגמה הקודמת – הכל נמצא בתוך הקלאס. מייד אחרי היצירה של המאזור עם חס, אני מכבה את המאזור הספציפי הזה באמצעות שימוש בפקודה off והעbara שני ארגומנטים: שם האירוע ורפרנס לפונקציה שמתפלת בו. בדיקן כמו ב-חס. אם תריצו את הקוד הזה, תראו שדבר לא קורה למרות שעשית emit.

## הפעלת יותר מאירוע אחד

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

```
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 חס, 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 ומאזיניות שאנו רוצים לאירוע אחד. זה כל העניין באירועים. מתרחש `emit` אחד – אבל לאירוע הזה שמתקיים יכולים להיות אלפי מאזינים שיבצעו אלף 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`, אמן יוציאו יוץ קודם, אמן יכולים להשתמש במתודה `prependListener` שמקהה את הפונקציה המאזינה שמתקבלת וגורמת לה לירוץ קודם (למרות שכך לא יסתמך על זה). אם נסתכל על הקוד הזה ונರיץ 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`, איז למרות שהצמדנו את `someEventOtherHandler` לפני `someEventHandler`, היא תרוץ ראשונה. יש לנו גם את `prependOnceListener` שהוא מקבילה של `once`. כלומר היא מעבירה את הפונקציה המאזינה לראש התור, אבל היא תרוץ פעם אחת.

וכמובן יש לנו את `removeAllListeners` שבאמצעותה אנו יכולים להסיר את כל המאזינים לאיורע 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 לאירוע `someEvent`. בגלל שהסרנו אותו באמצעות `.removeAllListeners()`.

## 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 ב-`on` בקובק של הפונקציה המאזינה, והארגומנט יכול 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', 'Hellllloooo');
  }

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

const myClass = new MyClass();
```

מה שiodfō CAN כתוצאה מההרצה הוא:

someEvent occurred! with Helllllloooo

ה-oooooo Helllllloooo כМОן הגיע מהתווע עצמו.

### תרגיל:

צרו קלאס שירש מ-`events` ונקרא כלב. בכל פעם שמוועל אירוע של `food`, הקלאס כותב "נבייה" אל הקונסולה. הפעילו את האירוע מתוך הקלאס ומוחוץ 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();

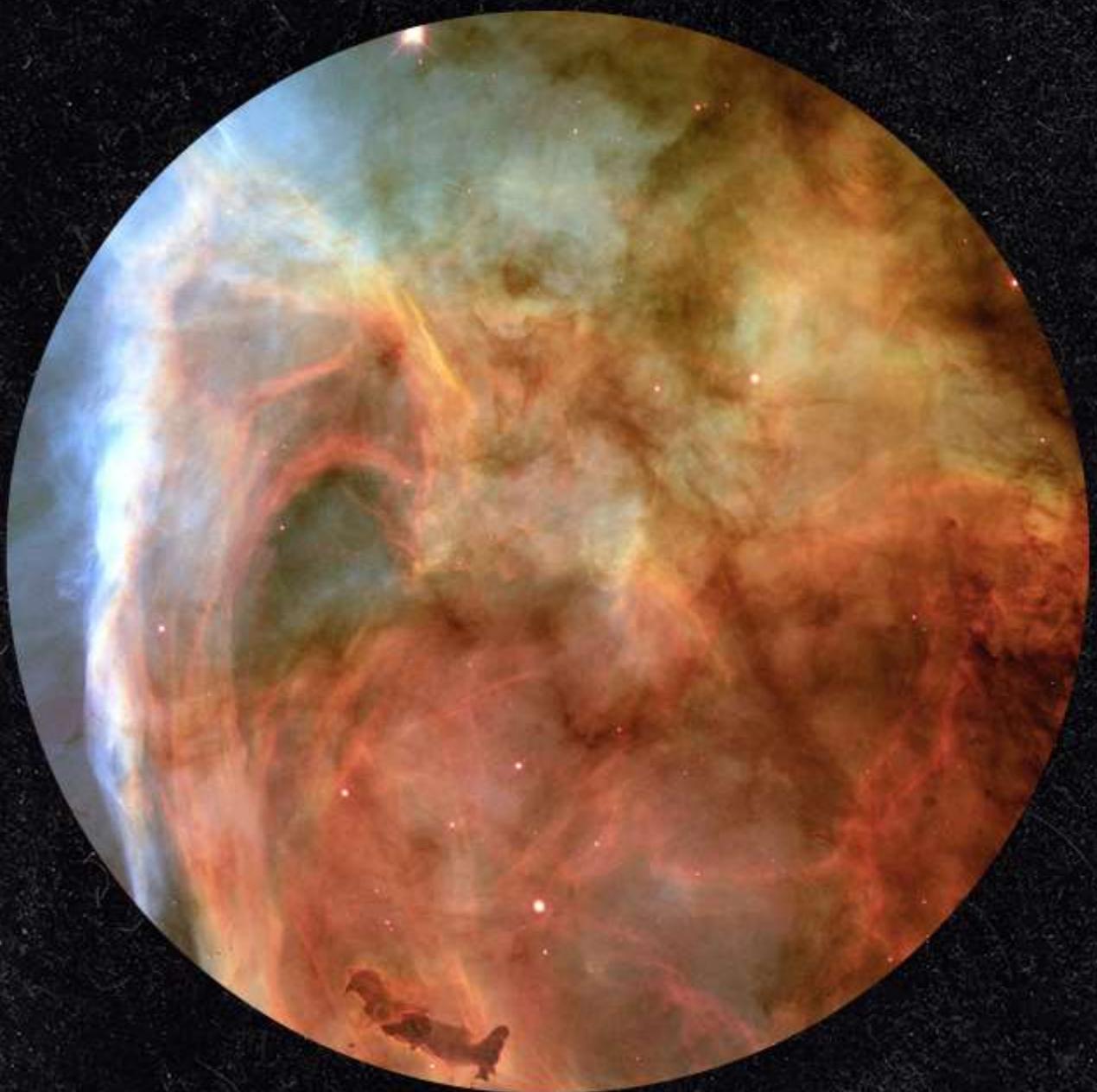
dog.emit('food');
```

הדבר הראשון שעשית הוא לבצע `require('events')`. מהרגע זהה אני יכול לבצע `extends` מהקלאס של, שקרأتي לו `Dog`, ל-`EventEmitter`. מהנקודה הזו לקלאסו שלי יש את כל המתודות של `EventEmitter` ואני יכול לעבוד.

יצרתי `constructor` והכنته בו `super` כיוון 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 מתודת `on`. מאיפה היא הגיעה? זוכרים שאני יורש מ-`EventEmitter`? ממש. 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 "נביות".

פרק 7

# ויצירת שורת הדר אבסיסי



# יצירת שרת HTTP בסיסי

אחרי שספרק הקודם למדנו על אירועים, אנו יכולים להשתמש במודולים שמשמשים את events ייורשים ממנה. המודול המפורסם ביותר הוא מודול `http`. המודול זהה מאפשר לנו בעצם לבצע פעולות מול הרשת, הן כיוזם התקשורת (מה שנקרא קלינט או לקוח) והן מקבל התקשורת (מה שנקרא שרת).

משמעות הראשית HTTP היא Hyper Text Transfer Protocol והדבר בפועל בuftprotוקול התקשורת הבסיסי של הרשת. כשאנו מפעלים דפדפן ונותנים לו פקודה להיכנס לאתר וצופים בתוצאה – הפקודה והතוצאה (זהה דף האינטרנט) מועברים על גבי הuftprotוקול זהה. 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 בבקשות שנשלחות אליו. אנו נתמקד בפרק זהה בשרת וניצור שרת זהה, שידעו לקבל בקשה של דפדפן ולהציג תשובה. כאמור, מודול השרת הוא `http`.

איך אני ידוע שמודול `http` משתמש באירועים? פשוט מאוד. זה כתוב בדוקומנטציה! אם תפתחו את הדוקומנטציה של `http` ותגלו אל `http.Server` תוכלו לראות באופן מפורש שהקלואס זהה יירוש מ-.

## Class: `http.Server`

Added in: v0.1.17

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

אם נלחץ על ה קישור נראה שבעצם הקלואס המרכזי שמןנו `https.Server` יירוש הוא `EventEmitter`.

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

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

- Class: `http.Server`

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

از ביצם `http` הוא מודול מורכב יחסית. הוא יורש ממודול אחר שנקרא `net`, ומודול `net` מכיל בתוכו את היכולת ליצור אירועים, יכולת שניתנה לו על ידי `events`. אבל זה לא מאוד מורכב. אותנו פחות מעוניין, כאשר אנו כתובים, ממי כל אחד יורש. אנחנו צריכים את הידע הזה כאשר אנו משתמשים בדוקומנטציה. מה שקרה הוא שלכל מודול יש המתודות והתכונות שלו.



כאמור, לא להיבהל. חמושים במידע זה, ניגש למלאת בניית השרת שלנו. תראו שזה יגמר בתוך כמה דקות.

ראשית ניצור `http.Server`. איך אנו יודעים שדווקא אותו? כיוון שהוא יורש מ-`net.Server` וכותב בມפורש שהוא מיועד ליצור שירותים שיודעים לקבל תנועת TCP, שזו התנועה הרגילה. איך אנו מבצעים `require`? כמו כל מודול אחר, אבל כיוון `http` יש כמה מודולים, אנו ניאלץ לבצע `require` מ-`http`. לא ממשו מלחיץ במיוחד, זה מצוי בມפורש בדוקומנטציה: `Class: http.Server`

ועכשיו? עכשו נבחן את המethodות שיש במודול `net` ונראה שיש מtodoת `listen` שמקבלת פורט. מה זה פורט? פורט הוא בעצם סוג של "ערוץ" שדרכו אנו מנהלים תקשורת. אנחנו רואים את זה לעיתים גם בכתובות אינטרנט. אם פעם יצא לכם לראות כתובות בסגנון: <https://some-sites.com:3456> אז המספר שמוצב אחרי הנקודות הוא בעצם הפורט.

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

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

אבל זה לא מספיק. מה יקרה כשהמשהו יתחבר לשרת שלנו? אנו צריכים שבכל מקרה של "חיבור", יוצג משהו למשתמש. עיון בדוקומנטציה של `http` ייתן את התשובה. אנו יכולים להשתמש באירוע מסווג `request`: [https://nodejs.org/api/http.html#http\\_event\\_request](https://nodejs.org/api/http.html#http_event_request) – האירוע הזה מפעיל את הקולבק עם שני ארגומנטים – הבקשה של הלוקוח, `request`, והותזאה שהשרת אמרו להחזיר, `response`. `response` הוא גם קלאס משל עצמו ואנו יכולים לבדוק את המethodות שלו.

[https://nodejs.org/api/http.html#http\\_class\\_http\\_serverresponse](https://nodejs.org/api/http.html#http_class_http_serverresponse)

מתודת אחת, שהיא `end`, היא מה שאנו מ Chapman – היא סוגרת את החיבור ומחזירה טקסט ללקוח. הקוד יראה 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-`js` קם בתיקיה שלכם, תפעילו אותו באמצעות כניסה עם הטרמינל למקומם של `js app` והקלהה של `js app` ו-`node` – תראו שהסקריפט של `js` לא עוצר. ה-`listen` משאיר אותו פתוח. אם אתם רוצים להרוג את התהילה, תלחצו על קונטROL + C.

אם תנווטו עם הדפדפן אל <http://localhost:3000/> תוכלו לראות>Hello World. בניתם את השרת הראשון שלכם!

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

```
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!')
  }
}
```

מה שעשינו כאן הוא לקרוא `http.Server` ליצור קללאו משלנו עם `super`. ה-`super` נדרש כאן כי אנו יורשים מקללאו אחר. אנו מוצאים האזנה לפורט 3000. במקביל, אנו מוצמידים את הפונקציה `requestHandler` לאירוע `request` בפרק הקודם. עשינו את זה בפרק הקודם. מה שנשאר לנו לעשות הוא לאתחל את הקללאו שלנו. פשוט ו נעים.

טוב, האמת שזה לא כל כך פשוט – הקוד הזה הוא בן 14 שורות, אבל בשביל ליצור אותו הינו צריכים לצלול לדוקומנטציות לא מעtot: לדוקומנטציה של `http.Server`, `http`, `listen`, `on`, `request`, `response` class וכמוון לדעת מעולה מה אנחנו עושים. זה באמת מסובך! אבל מה שיפה ב-`Node.js` זה שאפשר להעביר קריירה שלמה מבל' לקרוא את הדוקומנטציה או לכתוב שורות שמשתמשות במודולים הבסיסיים של `Node.js`, כיוון שגם תצרכו למשרת לא כתבו את השורות האלה אלא תשתמשו במודול קיים שנקרא `express` – מודול לבניית שירות אינטרנט שנלמד בהרבה בהמשך הספר. מבל' לשבור את הראש ולהיכנס לעומקם של דברים. אנו לומדים את זה כאן כדי להבין יותר לעומק איך אירועים עובדים ואייר לקרוא דוקומנטציה.

### תרגיל:

airoう `connection` שמתקיים בכל מסוג `http.Server` פועל בכל פעם שיש חיבור לשרת. צרו שירות התחברו לairoう זהה והדפiso לקונסולה עדכון על חיבור בכל פעם שימושו מתחבר לשרת שלכם.

### פתרון:

```
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 השירות קל ליצור, יש לנו את זה בדוגמה בפרק. מה שאנו צריכים לעשות הוא ליצור פונקציה שתפעל בכל פעם שיש לנו אירוע. ניצור אותה כחלק מהקלאס שלנו:

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

אחרי כן, ניצור פונקציה מואזינה עם `on` לאירוע `connection` ונחבר אותה ל-`connectionHandler`:

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

בגלל שזה בתוך הקלאס שלנו, ולא ישב בחוץ, אני משתמש ב-`this` כרפרנו לקלאס שלנו. קצת מסובך להבנה, אבל הקוד הרבה יותר קרייא.

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

כדי לשים לב שבדוגמה זו, כמו גם בשאר הדוגמאות, אנו מואזינים לפורט אחריה האירועים.

**תרגילים:**

צרו בתיקית הפרויקט שלכם קובץ txt בשם test.txt עם תוכן מסוים. אפשר גם תוכן ארוך במילוי (כמו למשל הטקסט הזה, המכיל את כל הקומדייה האלוהית מאות דנטה אליגייר): <http://www.gutenberg.org/cache/epub/8795/pg8795.txt> http://www.gutenberg.org/cache/epub/8795/pg8795.txt רמז: עשו זאת באמצעות fs.readFile שימצא ב-requestHandler.

**פתרונות:**

```
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 הקודם, אני יוצר קלאס שירוש את כל המתודות של httpServer. ב- constructor אני שם super, כי ג'אווחסקרייפט דורשת זאת ממנה. אחרי כן אני יוצר את השרת באמצעות методת listen ויאżן לאיורע request כפי שיש בדוקומנטציה – בהזנה, במקום להחזיר מחרוזת טקסט פשוטה, אני מכניס קריאה ל-`fs.readFile` כפי שלמדנו. התוכן שחוזר ממשם הוא התוכן שמוזכר במתודת `end`.

אשמור את הקוד הזה ב-`app.js` וапעיל אותו באמצעות ניווט בטרמינל אל מיקום הקובץ, הקלדה של `node app.js` וaz הקשה על אנטר. אני אוכל לראות את תוכן הקובץ.

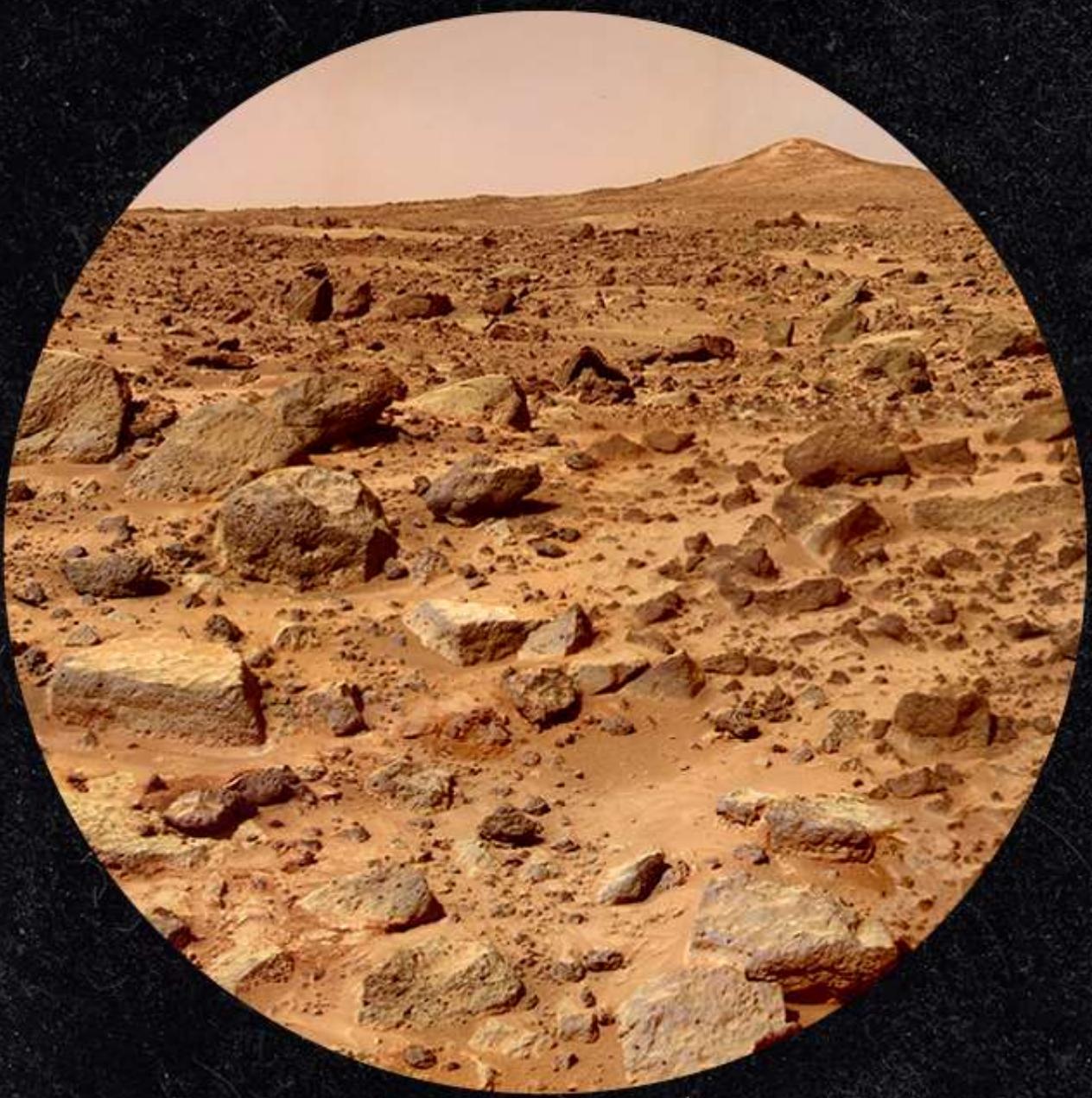
אם אתם רוצים לעשות זאת באופן אסינכרי ללא קולבקים – עושים זאת כך:

```
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 שולן EVENT LOOP



# 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 עמוקים יותר פרק זה.

ראינו עד כה ש-Node.js עובדת בדרך כלל אוטונומית, אבל ג'אנושקרייפט היא אסינכרונית (כפי שלמדנו מוקדם יותר בספר – מרים את הפקודות לפי הסדר שבו הן נכתבו). הסיבה שבוטייה Node.js היא אסינכרונית היא בגלל הפלטפורמה של Node.js שמאפשרת לה לעבוד כר. הבנה של הפלטפורמה והדריך שבה היא מימושת את האסינכרוניות זו היא קריטית.

## METHODS TIIMERS

ב-Node.js יש לנו כמה טייםרים – פונקציות שמשייעות לנו בתזמון הקוד שלנו. משתמשים בהם לכל מיני שימושים שאדגים בהמשך, אבל בינתיים אסביר עליהם כאן.

### תץ כשאני אומר לך – `setTimeout`

מדובר במתודה שמקבלת שלושה ארגומנטים. הראשון הוא קולבק שרצ – הקוד שאמור לרוץ כשאני אומר לו. השני הוא מספר המילישניות שיעברו עד שהקוד רץ, והשלישי הוא ארגומנטים שנitin 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 כמוובן:

Callback with arg 1

## טרץ מיד עם קולבק – `setImmediate`

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

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

למה צריך פונקציה כזו? בהמשך נבון.

## הlolאה הקבועה `setInterval`

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

אם תרצו את הקוד הזה, תוכלו לגנות שהוא לא עצור עד שתאתם לא הרגים את התהלהך באמצעות קונטרול + 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');
```

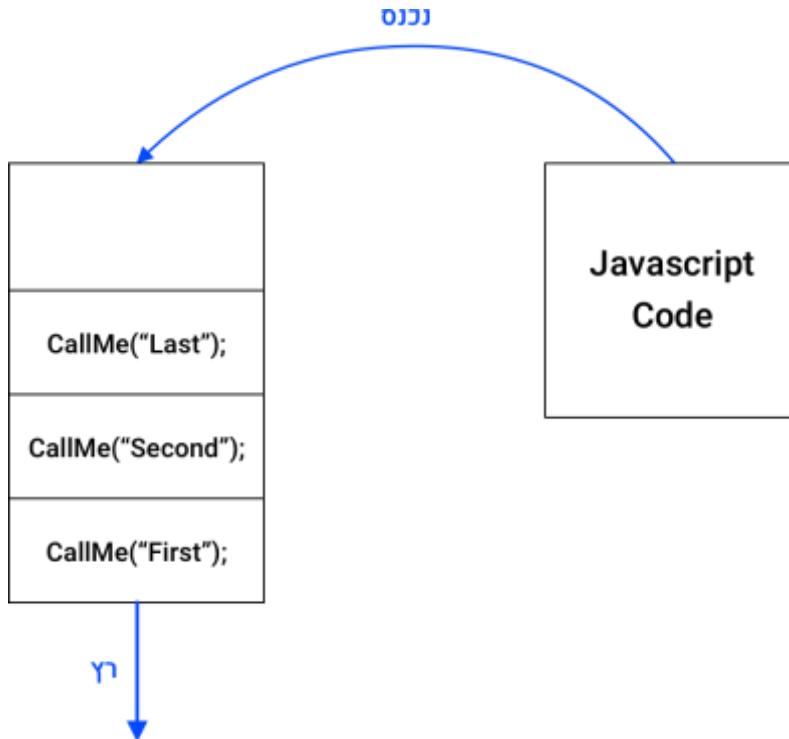
בפלט אני אראה:

```
First
Second
Last
```

מה קורה מאחוריו הקליים? איך ג'אווהסקריפט יודעת להריץ את הקוד? מדובר בקוד סינכרוני, קוד שרצץ לפי הסדר מלמעלה למטה והוא בולם את השימוש. ג'אווהסקריפט לא תמשיך לשורה השנייה לפני שהשורה הראשונה מסתיימת לרווח ולא תמשיך לשורה השלישית עד שהשנייה תושלם. התור הזה נקרא "טור הקריאה" והוא יעבד גם עם פונקציות כמו:

```
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 טימרים למשל (שלוש הפונקציות שראינו לעיל) או קטעי קוד אסינכרוניים שמתפלים בפלט ובקלט (O/A) כפי שלמדנו קודם. עם `File System`?-can נכנסת לפעולה לולאת האירועים של `js.Node`.

מה קורה כשיש לנו קוד מעורב זהה?  
ראשית, הפקודות הסינכרניות מורוצות לפני הסדר. אם יש קריאה למשק חיצוני, כמו קובץ או רשת – הן מורוצות והקובוקים שלהן נכנסים לתוך הקובלוקים של הפלט/קלט. אם יש פונקציות של טימרים, הן נכנסות לתוך הטימרים. אחרי שכל הפקודות הסינכרניות מורוצות, רץ ה-`loop` על כל חלקיו:

בחלק הראשון רצים כל הטיימרים שזמן הגיע. בחלק השני – הקובלוקים. החלק השלישי הוא פנימי ובחלק הרביעי רצות כל הבקשות שמגיעות מבוחץ (למשל שרת `http`).

כדי להמחיש את זה נבחן את הקוד הבא:  
צרו קובץ בשם `test.txt` בפרויקט שלכם, פתחו את `app.js` והדביקו 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` מ-`console.log` סינכרוני שנמצא בקוד. הם לא נמצאים בתוך קולבקים, לא נמצאים בתוך טיימרים. סתם זרוקים בקוד. אך המנווע של `Node.js` מרים אותם מייד. את הקראיות לקובץ הוא גם מרים, אך לא ממחכה לתוכאה (מדובר בפעולה אסינכרונית). הוא מכניס את הקולבקים ואת התוצאות שלהם כשייגעו, לתוך תור הקולבקים ב-`Event Loop`. הוא מכניס את הטיימרים שיש לנו – ויש לנו אחד – אל התור של הטיימר.

ועכשיו הולא נכנסת. קודם קודם כל `Node.js` בודקת את ענייני הטימרים. מה יש בתור? רק `setTimeout` אחד – אם הזמן הגיע, הוא רץ. מה זה הזמן הגיע? האם עברו מספר המילישניות מרגע הרצת הקוד? במקרה זהה צינתי אפס, אז הטימר הזה יורץ. ועכשו? עכשו לטור הקולבקים. אם יש קולבקים הם יורצו לפי הסדר שבו הם הושלמו. זו הסיבה שלא תמיד נראה 1,2,3,4 בפלט. אלא זה תלוי בסדר שבו מערכת הקבצים השלימה את הקרייאות השונות. כל קולבק נכנס לטור לפי הסדר שבו הוא מושלם ולא לפוי הסדר שבו הוא נקרא.

הכל קורה באופן אסינכרי, כלומר אין שהוא שבולט את מעגל האירועים של `Node.js` אלא אם כן בקולבקים שהוא קוד כבד במיוחד – וזה אחת הסכנות הגדולות שבגלן אני מלמד את זה בספר – בקולבקים לא אמרו להיות קוד כבד כי אחרת הוא יחסום את המעגל וכל המעגל יתעכבר בגלן קוד כבד. קוד כבד לא אמר להתקיים ב-`Node.js` ובמיוחד בשרטים המבוססים על `Node.js`. אם מונחים יותר מדי את הקוד, המעגל ייתקע, התור של הקולבקים יתנפח ומתישהו הסקריפט יקרוס מוחסר זיכרון.

יש שלב במעגל האירועים שנקרא `check` – במעגל הזה `setImmediate` 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 ירוץ הפלט הסינכרוני:

First

Last

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

```
setTimeout #1 Finished
```

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

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

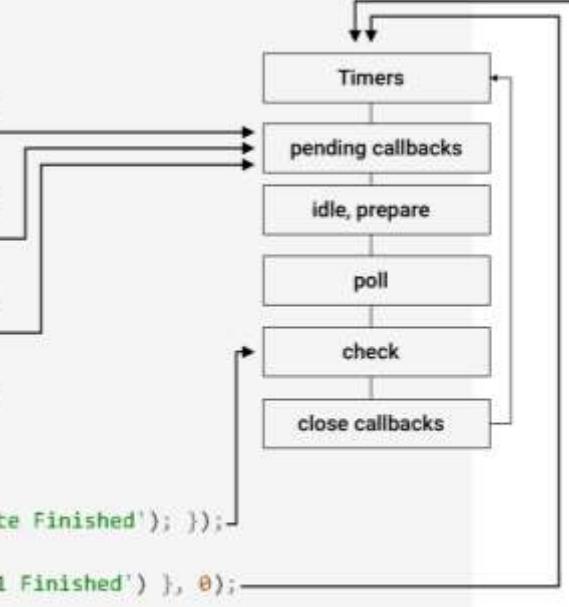
console.log('first');

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 ואנו מגיעים לשלב ה-check, שבו ה-setImmediate עובד ונראה בפלט את:

`setImmediate Finished`

הלוואה סוגרת את מה שנותר לסגור ומתחילה מההתחלה. שלב הטימרים מגיע. אם בשלב זהה עברו 5 מילישניות (ה-setTimeout השני אמר לרוץ 5 מילישניות לאחר הפעלה שלו) אנו נראה:

`setTimeout #2 Finished`

אם לא, הקוד ירוץ מיד אל Pending callbacks וירץ את מה שחרור ממערכת הקבצים לפי סדר החזרה. קלומר נוכל לראות שם:

Reading data 3  
Reading data 1

אחרי שהתור התרוקן, הלוואה ממשיכת לפעול, check ריק אך חוזרים לטימרים. אם ה-setTimeout השני עדין לא רץ, הוא ירוץ עכשו כי בטח עברו 5 מילישניות. ואז הלוואה אל ה-Pending callbacks.

יש שם משהו? מעולה. Node.js תritz את הכל. איז? מצוין. חזרה לlolאה עד שכבר אין שום קולבק שמחכה לחזור ואז הסкриיפט ימות. העבודה הסתדרה והlolאה רצתה במלואה.

**תרגיל:**

מה יהיה לפי דעתכם הפלט שתראו אם תritzו את הקוד הזה?

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

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

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

**פתרון:**

```
First
Last
Callback with arg
```

התשובה היא ברורה למדוי – ראשית הקוד הסינכרוני רץ, ורק אחריו כל מה שנכנס לlolאת האירועים של Node.js. במקרה הזה רק .setTimeout.

**תרגילים:**

מה יהיה לפי דעתכם הפלט שתראו אם תרצו את הקוד הזה?

```
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 הסינכרוני רץ, שתי הפקודות האחרות עוברות ללולאת האירועים. setTimeout נכנס לחולק של הטימרים -setImmediate לחולק של ה-`check`. הלולאה ריצה בפעם הראשונה. כיוון ש-setTimeout אמר לפעול 0 מילישניות לאחר תחילת הסקריפט, הוא יעבד מיד. הלולאה ממשיכה לחולק של ה-`check` ומricane את `setImmediate`.

פרק 9

# STREAMS



# Streams

בפרק על ייצרת שרת בסיסי הראיתי 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 לאיירע `request`, וברגע שהוא מתקיים הוא מבצע `fs.readFile`. הבעיה עם הקוד הזה היא שאם `fs.readFile` לוקחת הרבה זמן, אם הקובץ גדול מדי, זה ייקח זמן. בעצם מה שיקרה פה זה שה-`fs.readFile` תיכנס לולאת האירועים של Node ועד שהקובץ לא יישלם, הולולה תרוץ וזה עלול להושיב את הלקוח מול מסך לבן במשך הרבה זמן.

זה בדיקן כמו להוריד סרט לחולוטין ורק אז לראות אותו. דבר אפשרי בהחלט, אבל מייאש מאוד ועלול לוקחת הרבה זמן. במצבות אלו מורידים בכל פעם חלק קטן של הסרט ומתייחסים לראות אותו לפני סכל הסרט ירד. כך אנחנו לא צריכים להמתין. אותה התנהגות יכולה לקרות פה בעזרת Streams. במקרה הזה אנו מונעים מהלקוח לצבץ זמן יקר ולחכotta לקובץ `txt` שיקרא 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 קצת שצופה בקובץ או איזושה תוכנה אחרת.

סטרימים הם לא המצאה ייחודית ל-Node.js. למעשה, כל פלטפורמת צד שרת עשו את זה, אבל זה משווה שעובד מעולה ב-Node.js היישר מהkoposha.

הבה נדגים ונראה איך זה עובד. הקוד הבא הוא של שרת שקורא קובץ בעזרת סטרימים:

```
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`. זהה מתודה שמקבלת בדוגמה ארגומנט אחד בלבד – שם הקובץ שהוא אנו קוראים. המתודה הזאת נמצאת באופן טבעי ב-`File System`, ממש כמו `readFile`. גם הארגומנטים שהוא מקבלת זהים לחולstein `readFile` שלו כבר למדנו, אבל בኒיגוד `readFile` אני לא משתמש בקורסוק אלא מעביר את התוצאה באמצעות שיפוק אל ה-`response` שהוא תגובה השירות. זה הכל!

כשאני יוצר סטרים אני בעצם יוצר ברצ שמזרים מים, שאני חייב לחבר אותו לאנשאנו עם `pipe`. ממש כמו אינסטטטור. אני יוצר את ברצ הנתונים המזרים נתונים מ-`test.txt` ומעביר אותם אל ה-`readFile`. בקצת `response` יש שיפוק.

אבל מה שקרה מאחורי הקלעים הוא שהקובץ לא נטען במלואו לזיכרון אז נשלח במלואו אל הליקות דרך השירות, אלא הוא נטען בחלקים, וזה אומר שלולאת האירועים שלנו לא תיתקע כל כך מהה.

בואו נדגים שוב. הפעם עם שני סטרים. סטרים אחד שקורא – קלומר מספק את זרם הנתונים, וסטרים אחר שכותב – קלומר מנתב 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 לקרוא, והשני הוא סטרים שכותב את הנתונים, סטרים לכתיבה. בדיק כמו אינטטלו אני לחבר את הzinor שמצויה נתונים אל הzinor שמצויה להם. למעשה מתבצעת העתקה של קובץ אחד לקובץ שני, אבל באופן שמקל על הזיכרון. במקרה לטען קובץ שלם אל הזיכרון אז להעתיק אותו, אני עושה זאת בחלקים באופן שקויף וקל יותר.

## סוגי הסטרים השונים

יש כמה סוג סטרים ב-`js.Node`, שניים מהם כבר רأינו: `Readable Stream` הוא סטרים לקרוא נתונים, זהה שהוא מקור הנתונים. הוא נקרא `fs.createReadStream`. הדוגמה שהשתמשנו בה היא `httpServer.response`.

השני הוא סטרים לכתיבה הנתונים, זהה שידוע מה לעשות עם הנתונים שmaguiim אליו. הוא נקרא `Writable Stream`. ראיינו שני סוגי אלו, `fs.createWriteStream`. נוסף על הסטרים שכבר ראיינו יש סטרים נוספים: סטרים דו-כיווני (Duplex) לכתיבה וקריאה כמו סוקט, שעליו נרחב בפרק על סוקטים, וסטרים שמבצע טרנספורמציה, כמו ר הוא מקבל מידע, משנה אותו ו מעביר אותו להאה.

## סטרים טרנספורמציה

כמו אינטטלו שמחבר כמה צינורות, אני יכול לחבר שלושה צינורות: אחד שקורא את המידע ושולח אותו להאה, השני שמקבל את המידע, מבצע עיבוד שלו ושולח אותו להאה, והשלישי שמשתמש במידע. למשל – סטרים שקורא קובץ טקסט, סטרים טרנספורמציה שלוקח את הקובץ, מכועז אותו ב-`z` ו מעביר אותו להאה וסטרים שכותב אותו. המודול הטבעי של `js.Node` שעוסק בכיווץ נקרא `zlib` והוא מכועז בפורמט `z` ואנו יכולים להשתמש בו כדי לבצע כיווץ של קבצים, עם סטרים. איך זה נראה? בפשטות כר:

```
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 לכם קובץ בשם `out.gz` שהוא קובץ מכווץ. מה קורה פה? פשוט מאוד – סטרים אחד קורא, סטרים שני מבצע כיווץ והשלישי כותב. הדבק ביניהם נעשה באמצעות `readable.pipe(writeable)`. זכרו את מטפורת האינסתלטור. זה בדיק מה שאותם עושים.

## AIROUIM BOSTRIMIM

סטרים מכילים אירועים שאפשר להשתמש בהם בנסיבות כדי לבצע פעולות שונות בתחילת העברה, תוך כדי העברת הנתונים ובסוף העברת הנתונים. האירועים משתנים מסטרים קרייה לסטרים כתיבה.

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

הairyous המשמשים בירורם בדרך כלל `end\finish\drain\data`.  
 איך משתמשים באירועים? בדיקן כמה בכל מקום. יצרת סטרים? עם המתוודה חס ושם האירוע אני יכול להאזין למידע. יש אירועים שמעבירים מידע (כמו `drain`) ויש כאלה שלא. איך אנו יודעים אילו? בדוקו מנגנון 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 מענינית:

```
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 בינארי או לא בינארי ממש בקלות ובלית התאמץ יותר מדי, מבליל לחשב באפרים (Buffer). באפר זה בעצם "אזור המתנה" של הסטרים. כשאנו יוצרים סטרים, הנתונים שלא מספיקים להיכנס אליו "מחכים" בעצם באפר. אבל כאמור אנו לא נדרשים להבין עמוק את עניין הבאפרים. יוצרים סטרימים ומחברים אותם עם פ'יפים, ובאמצעות אירועים 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` באמצעות סטרימים והכניסו אותו לקובץ בשם `js.app`. הפעילו אותו באמצעות `node ./js.app`. מהטרמינל ובדקו שנוצר קובץ `out.txt` שהתוכן שלו הוא בדוק כמו `test.txt`.

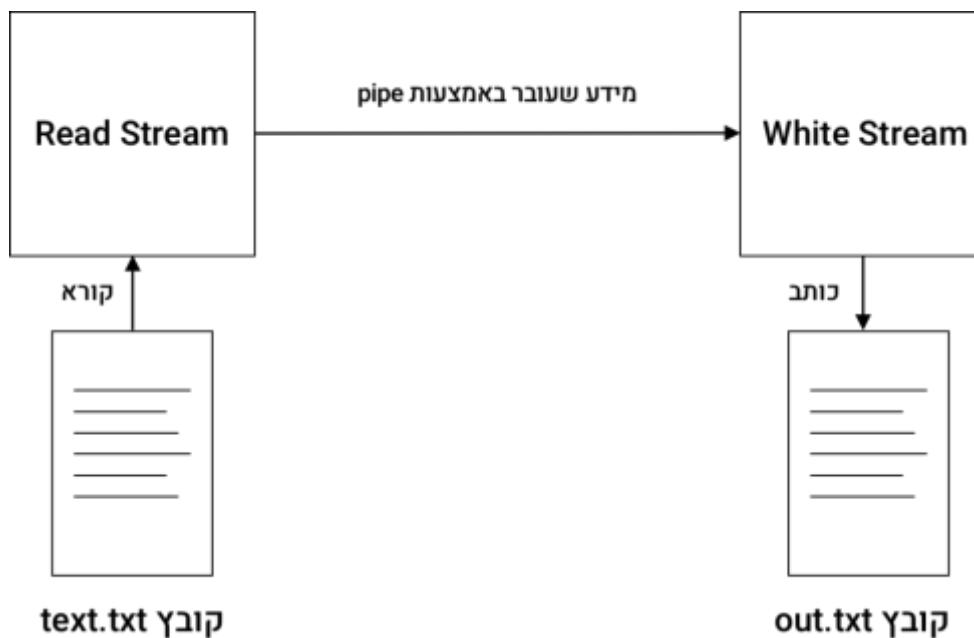
**פתרון:**

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

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

readStream.pipe(writeStream);
```

**הסבר:** ראשית קראתי לモודול `fs`, כי שלמדנו בפרק ואנו רואים בדוקומנטציה, יש שתי מethodות: `createReadStream` ו-`createWriteStream`. המethodה הראשונה `createReadStream` מקבלת שם הקובץ `test.txt` שמננו אנו קוראים. המethodה השנייה `createWriteStream` מקבלת את שם הקובץ `out.txt` שאליו אנו כותבים. כל מה שנוצר לעשות הוא לחבר את המethodות השונות באמצעות `pipe`. הстрימים הקורא מתחבר לстрימים הכותב.



**תרגילים:**

יש צורך בהצפנת 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 הפעלה `createCipheriv` והצפינו את הקובץ `test.txt` מ-`encrypted.txt`

**פתרונות:**

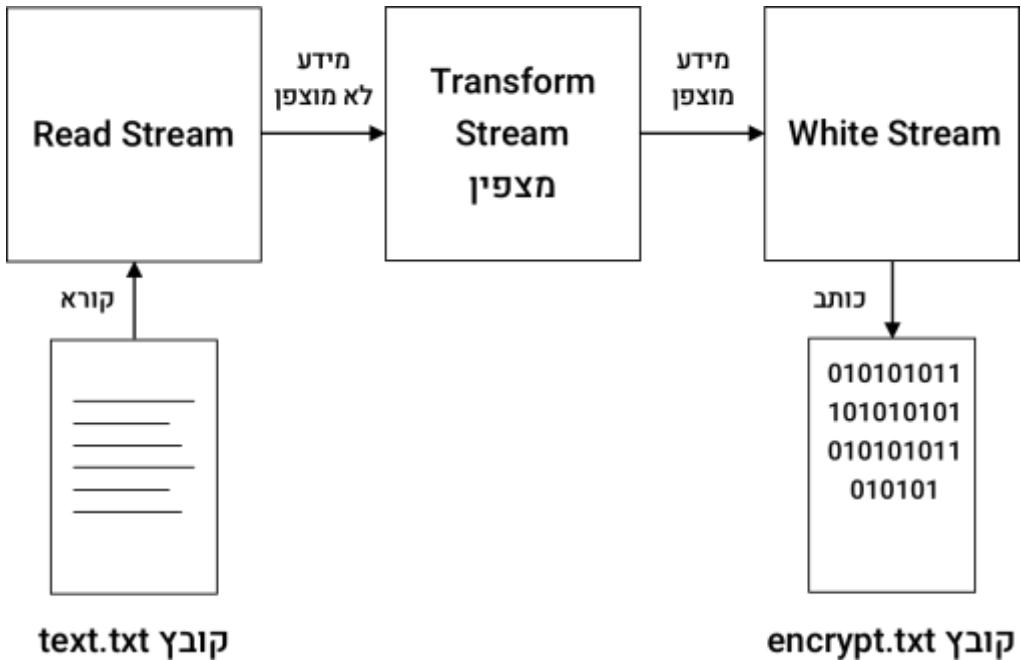
```
const fs = require('fs');
const crypto = require('crypto');

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 בתרגילים: `algorithm`, `key` ו-`iv`. אם תעמיקו בקריאה תוכלו לראות ש-`createCipheriv` יוצר `cipher` שעובד סטרים. כל מה שנותר לעשות הוא ליצור סטרים בדומה לדוגמה של ה-`zlib` שיצרנו ולחבר את הכל עם `pipe`.



כדי לשימוש ב-`sh-cryptSync` היא פונקציה סינכרונית חוסמת. אם נzieב פונקציה כזו בחיים האמיתיים, נסתכן בבעיית מהירות ויציבות.

**תרגילים:**

קחו את הקובץ המוצפן שיצרתם, קראו אותו באמצעות `readStream` וענחו את הצופן באמצעות `createDecipheriv` אל קובץ `./decrypted.txt` אל סטרים `fs.createWriteStream('./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);

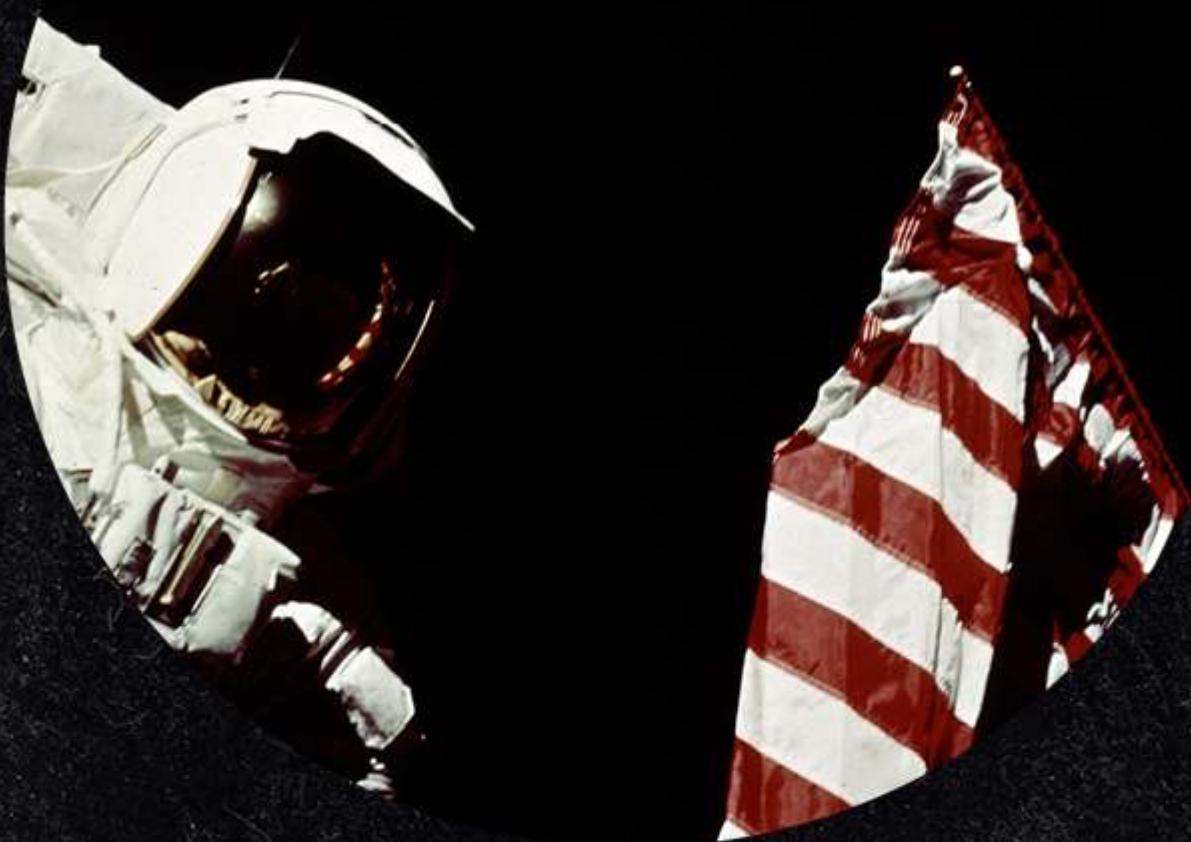
const readStream = fs.createReadStream('./encrypted.txt');
const decryptStream = crypto.createDecipheriv(algorithm, key, iv);
const writeStream = fs.createWriteStream('./decrypted.txt');

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

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

פרק 10

# אריזת הירוד שלנו נמושול



# אריזת הקוד שלנו כמודול

עד כה השתמשנו במודולים – בין אם מדובר במודולים חיצוניים או במודולים של Node, הקוד שלנו הוכנס לתוכה `js.js` והפעילו אותו ישירות. אבל תמיד מתקבל לארוז את הקוד שלנו בתוך קליאס נאה שאפשר לעשות לו `require`. הוא לא קוסם אף. הוא מבצע קריאה של קובץ ומביא את מה שאותו קובץ מיצא.

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

```
const myNumber = 42;
```

אני רוצה להגיע למספר שאם אני עושה ל-`require module.js` פקודה require בקובץ אחר, כמו `js.app.js`, אני אקבל 42. מהו בסגנון זהה:

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

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

אני מיצא דברים באמצעות אובייקט מיוחד `Node.js` מיוחד ל-`module.exports` שנקרא `module`. הוא נמצא ב-`js.module.js` כאובייקט גלובלי ומה שנכנס אליו יוצא `require`. – אם אני אכנס את המשתנה שלי – זה שנמצא בקובץ המודול שלי `js.module.js` – ל-`module.exports`, אני אקבל 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.js`, תיכנו לקובץ אחר ותכתבו:

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

תראו שב-`myModule` או בכל משתנה אחר שמקבל את ה-`require` מקבל 42. ואפשר כמובן 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 `require('myModule')`, אני אוכל לראות את האובייקט הזה.  
אני יכול לעשות `export` לקליאו שלם, כמו למשל הקלאס שיצר שרת אינטרנט, שהראיתי את הקוד  
שלו בפרק הקודם. אם אני אשימ אוטו ב-`myModule`, אני אוכל ליצא 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 אציגו אותו? בדוק כמו משתנה, ככה:

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

myServer = new myModule();
```

אני יכול לבצע את הייצירה של השרת 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');
```

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

**תרגילים:**

קחו את הפתרון לתרגיל בפרק על בקשת http והפכו אותו למודול שיישב 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();

והפעלה מתבצעת עם:

```
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 והוא יתבצע כמו שיעשה require לקובץ זהה, יקבל בחזרה את:

```
new GoogleCaller();
```

כאילו כתבתי את הcola באותו קובץ.

פרק 11

# קביעות גרסאות



## קביעת גרסאות

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

אם אני מתקין 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 שמדובר בלקוח, בחר נוסף בצוות או בשרת – אני לא צריך להעתיק את כל קובצי המודולים, אני רק צריך לוודא שלפרויקט שלי יש `package.json` בתיקיית האב שלו. הלקוח יוכל לעשות `npm install` ותוכנת NPM תסתכל על `package.json` ועל דוד.   
לקראת ממנה את גרסאות המודולים והשמותיהם שליהם ולבצע התקינה נוספת.  
על מנת לבדוק זאת, העתיקו את הקובץ `package.json` זהה אל תיקיה כלשהי 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 התקינה וכתבו שם `npm init`. תראו איך תוכנת NPM מתקינה את כל המודולים האלו בתיקיית `node_modules`.  
אנו יכולים לקבוע את הגרסאות של המודולים השונים וכמו כן גם את הגרסה של המודול שלנו.

## גרסאות סמנטיות

לכל תוכנה שהיא יש גרסאות – בין שמדובר במודולים של js.js או במודול שהוא כתוב בעצמו, או בכלל בתוכנות שלא קשורות לג'אווהסקרייפט. באקויסיטם של js.js אנו עובדים לפי גרסאות סמנטיות. מדובר בסטנדרט של ממש שיש לו גם תרגום לעברית באתר הרשמי. אתם מוזמנים להיכנס ולקראן 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**, במרקחה הזהה 4, הוא הינו **למייג'ור** (Major) – גרסה ראשית. כמשמעותה צו – שוברים API. כלומר מושנים/מוחקקים מתודות או התנהוגיות. כמשמעותם מודול בגרסה ראשית, נראה שהוא יישבר בתוכנה שלנו. הדגש הוא על "כנראה". לא תמיד שוברים API בקידום גרסה ראשית; יש כאלה המתכוונים את ה-API שלהם בצורה נכונה וგמישה ויכולים לעשות שינויים גדולים גם בלי לשבור או לשנות התנהוגות. אבל בקידום גרסה ראשית המפתח של המודול מאותת למי שבונה על הגרסאות הללו שצורך להיערכן לכך.

**החלק השני**, במקורה הזה 17, הוא הינו למיינור (Minor) – גרסה משנית. כشمקדמים גרסה צזו מבעים שונים מהותי בתוכנה – אבל עם תאימות לאחר. כלומר משחו לא ישבר בתוכנה שלנו אם נקדם את הגרסה הזו, אבל יש סיכוי שנוראה אחרת.

**החלק השלישי**, במקורה הזה 11, הוא הינו לפאטץ' (Patch) – תיקון באגים. כشمקדמים גרסה צזו מתקנים באגים/בעיות אבטחה או משפצים פונקציונליות – כלומר קידום של הגרסה הזו לא יעשה לנו בעית.

בואו נדגים. נניח שיש לי מודול צזה:

```
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 יצירת מודול משלנו. הגרסה שלנו היא:

1.0.0

אם החלטתי לתקן העריה כלשהי בקוד, למשל במקום:

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

לכתוב:

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

از שאני אוציא את הגרסה שלי-L-NPM או לגיטהאב או לכל מקום אחר, אני אתקן את הגרסה לגרסה 1.0.1, ככלומר אקדמי את הגרסה בפואט', כי לא השתנה ממשהו מהותי.

אם החלטתי לשנות את המתודה המרכזית שלי-`callGoogle` ל-`call` שמקבלת פרמטר `google` (אול', כדי לתרום באתרים נוספים בעתיד) אבל אני כן שומר את `callGoogle` כדי שתיה תמייה למשתמשים אחרים, ככלומר ממשהו צזה:

```
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 הטראה שהמתודה עומדת להשתנות. אני שיניתי בעצם קוד מהותי באפליקציה אבל עם תאיימות לאחר. זהו שינוי מיינור. אני אשנה את הגרסה שלי ל-1.1.0 במקומן 1.0.0.

כשאני אחליט להוריד את `callGoogle` לחלוטין, מי שנסמך על המתודה הזאת במודול שלי יצרר לשנות אותו. במידה שלא, הקוד לא יעבד. כלומר ביצעת שינוי מהותי בלי תאיימות לאחר. פה אני אהיה חייב לפחות את גרסת המיג'ור שלי ל-2.0.0.

אם אתם מייצרים מודול LNPM, ככה אתם-Amorim לעבוד וככה המודולים האחרים עובדים. כלומר – אם אתם מסתמכים על `request` – אין לכם בעיה לעדכן פאצ'ים, אין לכם בעיה לעדכן מיינורים – אבל כן כדאי שתתבדקו היטב בדוקומנטציה של המודול ובקוד אם אתם רוצים לעדכן מיג'ור.

## קביעת גרסאות סמנטיות ב-`json.package`

וחזרה ל-`json.package`. אם ב-`json.package` יש גרסה מדויקת של המודול, כלומר מספר לא תוספת של כובע או טילדה או כל סימן אחר – כאשר אני או כל משתמש אחר עשו ? ומקח לך שלכם, LNPM תתקן את הגרסאות המדויקות ביותר.

אם יש כובע (הסימן ^ או caret באנגלית) זה אומר LNPM תתקן את כל הגרסאות שתואמות לגרסה המצוינת. כלומר אם יהו שינויים בפאצ' או מיינור, היא תתקין אותן. מה זאת אומרת? זאת אומרת שם יש לי פירוט גרסה כזה:

```
"lodash": "^4.17.11",
```

אם אני כותב ? ו-NPM תראה שהגרינה של `lodash` היא 4.17.13, היא תתקין את הגרסה האחרונה. גם גרסה X.4.17. (X אומר כל מספר), כיוון זהה לא שובר פונקציונליות. אבל אם הגרסה האחרונה היא X.X.5 אז LNPM לא תתקין אותה אלא את הגרסה האחרונה של X.X.4. אם יש טילדה (הסימן ~ או tilde באנגלית) זה אומר LNPM תתקין את הגרסאות של הפאצ'ים בלבד, ולא של המינורים. כלומר, אם יש לי פירוט גרסה כזה:

```
"lodash": "~4.17.11",
```

אם אני מבצע התקינה מאפס של המוצר עם `npm` תראה שהגרסה של `lodash` היא גרסה 4.17.12 ומעלה, היא תתקן את האחורה, אבל היא לא תתקן את X.4.18. ובודאי לא את X.5.X. אפשר למכת על הקצה ולצין בפני `NPM` להתקן את הגרסה האחורה באמצעות הטקסט `:latest`:

```
"lodash": "latest"
```

זה אומר ש-`NPM` תבצע התקינה של הגרסה החדשה ביותר ואז אתם משתמשים שהcoil ישבר, ומאוד לא מקובל בתעשייה לעשות שיטות כאלה.

יש עוד אפשרות לקביעת גרסאות, אך אלו הנפוצות ביותר. רוב האנשים משתמשים בברירת המחדל של `NPM` – ה"כובע" – עדכון של המינורים והפאייטים בלבד.

**תרגילים:**

קבעו את הגרסה של lodash בפרויקט שלכם ל-3.3.0. התקינו אותה עם `npm` ובדקו את הגרסה במאזעות כניסה ל-`node_modules`, משם לתיקית `lodash` והצצה במספר הגרסה ב-`lodash.json` של `package.json`.

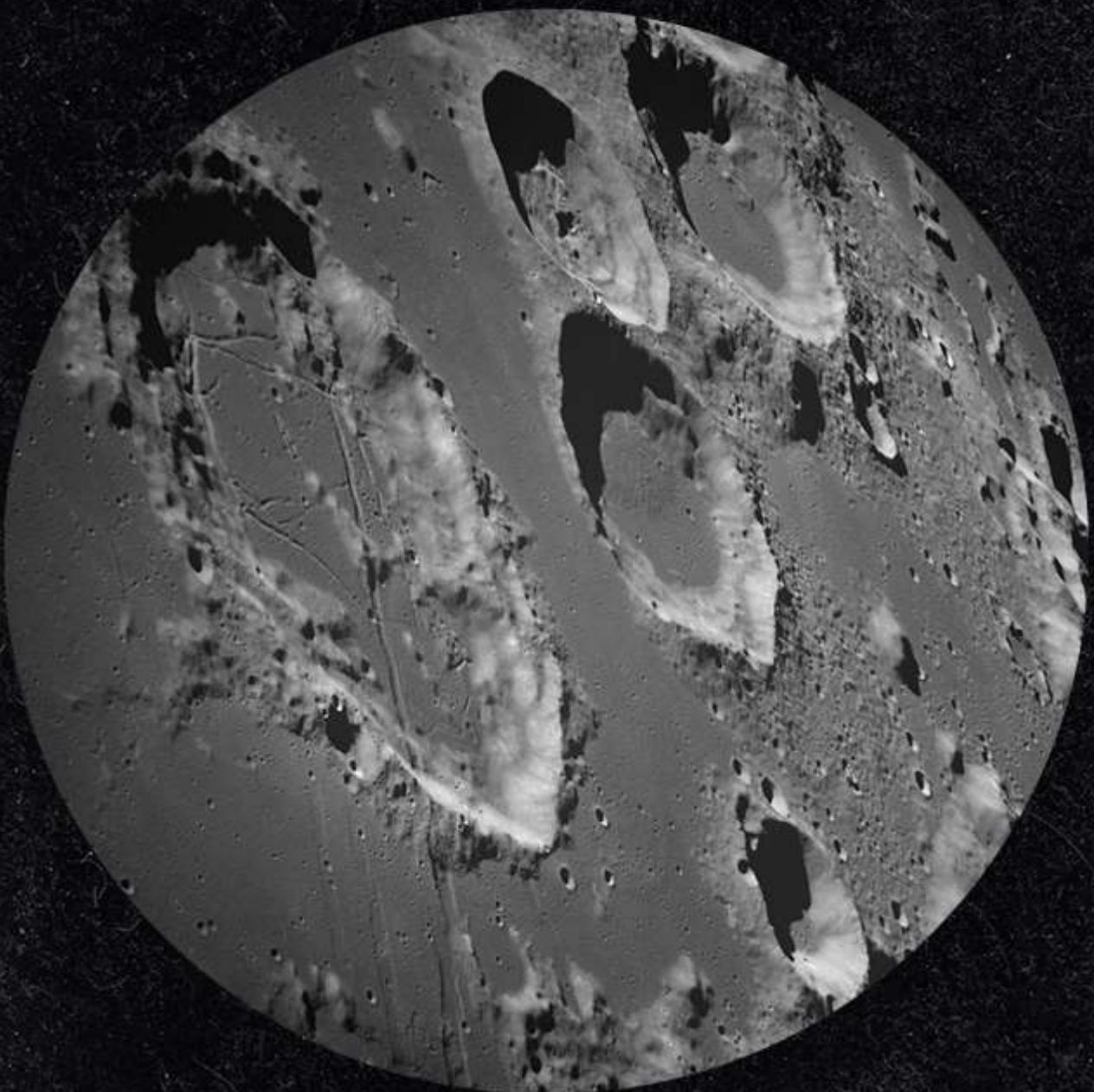
**פתרונות:**

```
{
  "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, בעוד מkapid להיות במיקום של הפרויקט שלי, את הפקודה `npm`, ואז אנטר. NPM תבדוק את מספר הגרסה ותתקין 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-ונ-



# התקנה גלובלית ו-CLI

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

חלק גדול מכל ה-CLI יכולים להיכתב על ידי `node`. אחד הכלים המפורסמים ביותר הוא `eslint`,CLI שבודק תקינות קוד ג'אווהסקריפט. בואו נתקין אותו בפרויקט שלנו. נכתוב בפרויקט:

```
npm i eslint
```

מיד תתווסף ל-`package.json` שלנו ול-`dependencies`, הגרסה האخונה של `eslint` עם "כובע". כלומר אם אני אעשה שוב בעתיד `i npm` – הגרסה האخונה של `eslint` שתואמת לגרסה המצוינת תותג.

כדי להפעיל את `eslint`, אני חייב להפעיל את הטרמינל. הרץ `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 .\app.js --no-eslintrc
```

אנו מתחילה ב-`node` כדי להריץ הכל בסביבת `js.Node` ואחרי כן אנו מקלדים את הントיב של `eslint` שהותקנה ב-`node_modules`. הントיב זהה הוא CLI של `eslint`. אחר כך יופיעו שם הקובץ שהוא לנו רוצים לבדוק (במקרה זה `js.app` שאנו מניח שיש לכם) והאפשרות `-eslintrc`-`-no` – שאומרת לא להסתמך על קובץ קונפיגורציה. אם תכתבו קצת ג'יבריש בשורה הראשונה בקובץ, הקלהת הפקודה הזו תיתן לכם שגיאה.

מפתחי 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 גלובלי. בהתקינה הגלובלית אנו בעצם הופכים מודול js.Node שעובד ב-CLI לגלובי. כלומר אני יכול להפעיל אותו מכל מקום בטרמינל. מכל מקום! איך עושים את זה?  
כותבים (בכל ספרייה שהיा) בטרמינל:

```
npm i eslint -g
```

הפלאג (flag) הוא כינוי לפקודה בטרמינל שבאה לאחר התו "-". -g אומר ל-NPM להתקין את eslint באופן גלובלי. NPM מתקינה את המודול לא בתיקייה של הפרויקט, אלא בתיקייה גלובלית במחשב ומקשרת (עם PATH בחלונות או סימלינק בلينוקס) את התקייה הזאת למערכת ההפעלה כך שאפשר יהיה להפעיל את המודול מכל מקום שהוא.

התקינו את eslint באופן גלובלי, ו>Show להתקינה שב-NPM מתקינה את המודולים הגלובליים. מוצאים אותה באמצעות הקלהה של

```
npm list -g
```

הפקודה הזאת מדפיסה היכן המודולים הגלובליים נמצאים ו呱 מילויים הותקנו (עם התלוויות שלהם). אתם יכולים לגש את אל התקייה הזאת ויראות eslint הותקנה בה. מה זאת אומרת? עכשו תוכלו להקליד eslint מכל מקום. הקלידו -u eslint בכל תיקייה שהיוא ותוכלו ליראות eslint-utf8 עובדת מכל תיקייה באופן גלובלי. עכשו אתם יכולים להשתמש בכל זהה בכל מקום במחשב שלכם.

יש לנוCLI באקזיטם של js.Node, כולל כלים שמייצרים סביבות מלאות לاجرוי כמו -create react-app או cli angular. קל להתקין אותם, קל לעובד אותם והם חלק מהכח של js.Node. זהה נחדר.

מצד שני זה עשוי להיות בעיתי – לملא את המחשב שלנו במידע שאנו לא צריכים. נוסף על כך המודולים האלה לא נכנסים ל-`json.package`. לפיכך התקנה גלובלית אינה מומלצת. מה הבחירה? אקח.

דרך נוספת לעבוד עם מודולים גלובליים היא באמצעות פקודה של NPM שנראית כך. הפקודה הזאת מופעלת באמצעות אקח 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 כמה פרויקטים משתמשים בגרסאות שונות של `eslint` או CLI אחרים. כדי שהזיכרון קודם, מומלץ מאוד להשתמש ב-אקח ולא בהתקנה גלובלית.

#### תרגום:

התקינו את מודול סx באופן גלובלי. מדובר במודול שדומה ל-`eslint`. בדקו שהוא עובד על ידי הריצה של סx מכל תיקייה שהיא. הדוקומנטציה של המודול נמצאת [פה](https://www.npmjs.com/package/xo):

#### פתרונות:

התקנת המודול נעשית באמצעות `g - ox` או `mk`. מהרגע שהקלדתם את הפקודה הזאת והקשתם על אנטר, תוכלו להשתמש ב-`g` באמצעות הקלדה של סx בלבד בכל מקום שהוא. אם אתם משתמשים בחלונות, יש סיכוי שתצטרכו לסגור ולפתח את הטרמינל כדי שהשינויים ב-`PATH` יעבדו.

פרק 13

# כתייבת און והתמונושיות עם ה-CLI



# כתיבת bin והتمמשקות עם ה-CLI

קל מאוד לכתוב CLI ב-Node. יש לא מעט מודולים שימושיים בתהילך ומאפשרים ליצור דיאלוגים, אнимציות וצבעים בטרמינל. אחד מהם כבר הכרתם בפרקם הקודם: המודול chalk שמאפשר לצבוע את הטרמינל בצבעים עלייזם.

כasher אנו כותבים CLI, אנו צריכים להتمמשק עם שורת הפקודה. בדרך כלל אנו כותבים את הפקודה ואז עוד ארגומנטים. למשל הפקודה:

cd ..

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

על האלגוריתם של ההצפנה כבר עברנו בפרק על הסטרימים. 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 נארוז את זה באופן שונה, בטור קלאס שעומד כמודול עצמאי לגמרי. למדנו את זה בפרק'ים הקודמים והקוד הבא אמרור להיות מובן שונה. בקובץ `src/module.js` ניצור את הקלאס שמבצע את ההצפנה. זהו אותו קוד בדיאק, 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 `EncryptionCLI`. ב-`constructor` אנו מגדירים את המשתנים שנצטרכם בהמשך: `iv`, `key` ו-`algorithm`. במתודה `encrypt` יש ארגומנט אחד – שם הקובץ. אנו לוקחים אותו ומשתמשים בו בקריאה ובכטיבה – בסטרים. בסוף הכתיבה נאצין לאיירע "סוף" ונדפס `Finished`.  
 נבנה את הקלאס באמצעות `new` וنعביר אותו לכל מי שעוזה `require` באמצעות `module.exports`. על מנת לבדוק שכל מה שכתבנו עובד, ניגש ל-`node app`, נוצר את המודול זהה באמצעות `require` ונשתמש בו. באותה תק"יה של `node app` ניצור קובץ `test.txt` ונכנס בו כמה מילים. ב-`node app` נוצר את המודול ונשתמש בו על קובץ `test.txt`:

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

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

אם הכל יעבד, כאשר אני אכתוב `node ./app`. `node` ייצור לי קובץ בשם `test.txt.encrypted` וANI אראה `finished` בטרמינל. הקובץ `test.txt.encrypted` יוכל מידע 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 שארצה!) והוא יבצע לי את ההצפנה. איך אני עושה את זה?

ראשית אני אוצר תיקייה בשם `ch0d`. נהוג, במקרים רבים של `Node`, לשים את כל הקבצים הקשוריםCLI ל-CLI בתיקייה זו. אני אוצר שם קובץ. הוא יכול להיות בכל שם שהוא אבל אני אבחר בשם `.encrypt.js`.

כדי לסתן ל-`js.Node` שמדובר בקובץ שנייתן להרצה מה-CLI אני חיב 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 מזמן אתכם לחפש אותה וללמוד עליה. אם אתם משתמשים במערכת הפעלה שאינה חלונית, אתם יכולים לנסות כבר עכשיו להקליד את שם הקובץ ללא `node` ותראו `sh.js` תרץ אותו כailo `node`.  
`./bin/encrypt.js`.

עכשו לעבודה. אנו יכולים לצרוך את המודול שלנו כמו ב-`node app`, בדיק כר:

```
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 כר:

```
└──bin
└──src
└──app.js
```

ואם אני נמצא בקובץ בתיקיה `ch06`, ואני רוצה לבצע `require` לקובץ בתיקית src שבה נמצא המודול שלי – אני נדרש לעלות לתיקית האב ואז לרדת לתיקית src. אם זה מסובך לכם, כדאי לכם להשתמש ב-`VSCode` שמאוד מקל את הניווט כי יש לו `auto complete`.

אחרי שיצרכתי את המודול שלי, אני צריך להשתמש בו. בשביל זה אני צריך להשתמש בשם הקובץ. כאמור, בתחילת הדוגמה הסבירתי שאני רוצה לקבל את שם הקובץ מהארגון בשורת הפקודה. איך? באמצעות המשתנה הגלובלי `process.argv`

ראשית, `process` הוא אובייקט גלובלי שימושי של `js.Node` שמחזק בתוכו מידע רב על התהיליך שמריץ את `js.Node` וגם אפשר לבצע פעולות רבות באמצעותו. בפרק זהה 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'אואסהקרייפט' שבו אנו פועלים. בשאר האיברים הוא מכיל את הארגומנטים בשורת הפקודה. אנו רוצים רק ארגומנט אחד, וזהו האיבר השלישי במערך (כלומר זה שנמצא במקום [2] – אנו זוכרים שמערך מתחילה מ-[0] ב'אואסהקרייפט'). אותו נשלח למетодת `encrypt` בדיק כפי שעשינו ב-`node app`:

```
#!/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 app, אבל הפעם עם ארגומנט – כלומר עם שם הקובץ שאנו רוצים להצפין:

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

אם הכל עובד כנדרה, נראה שנוצר לנו קובץ מוצפן בשם test.txt.encrypted. אם רוץ השם test.txt.encrypted 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 מעת מידע על המודול שלנו ועל המודולים שהוא משתמש בהם וברשותו. אבל הוא גם אמור להכיל מידע על ה-CLI שלנו, אם יש לנו. אנו נוסיף ל-JSON שיש ב-JSON שישי ב-package את הtekסט:

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

**bin** הוא החלק ב-package שמטפל בהרצה מתוך שורת הפקודה. באובייקט זהה יהיו כל הפקודות שהמודול שלנו מכיל. כרגע זו רק פקודה אחת בשם encrypt. הערך שלו יהיה מחרוזת טקסט שמכילה את הנתיב אל הקובץ של ה-CLI. כיוון ש-JSON נמצאת בתיקייה הראשית של הפרויקט, הנתיב הוא נקודה (כלומר המיקום הנוכחי), תיקיית bin ואז שם הקובץ.

אבל כדי שנוכל לכתוב encrypt מכל מקום, אנו צריכים לגדודו למודול להיות גלובל. כיוון זהה לא מודול ב-PM (עדין), אני לא יכול להתקין אותו באופן גלובלי עם -i ומקח או עם ak. אבל יש דרך לגדוד מודולים להפוך לגלובליים גם אם הם נמצאים רק במחשב שלי. איך? אני אקשר אותם באמצעות קישור סימבולי אל סדרית התיקיות הגלובלית שיש במחשב שלי, בדיקן כפי ש-d-i ומקח עשו, וזאת באמצעות הפקודה:

```
npm link
```

אני אקליד link ומקח בתיקייה הראשית של הפרויקט ואז אלחץ על אנטר. מהנקודה הזו אני יכול להקליד encrypt ושם הקובץ מכל תיקייה שיש בטרמינל שלי והקובץ יוצפן בבדיקה כפי שרציתי!

כרגע יוצרים CLI עם `node`. כרגע המודולים הגלובליים סוחרים `eslint` וכל מודול אחר שרצה עם CLI עובדים. אם תינקחו את המודול שלכם ותפרסמו אותו ב-`NPM`, הוא יעבד עם `g-` ו-`npm` – אבל עד אז גם `npm link` יופיך.

**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 שלה מהפרק על הסטרימים, שם היא מפורטת. הקלאס שלי יראה כעת כך:

```
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 משהו שאמור להיות ברור: יצירת סטרים של קריאה ממש קובץ שmagick ארגומנט, יצירת סטרים של פענוח עם הנתונים מה-`constructor`, יצירת סטרים של כתיבה ממש קובץ שmagick ארגומנט, הוספת `decrypt` והאזהה לאירוע "סיום".

עכשו ניצור בתיקיית `cho` קובץ בשם `decrypt.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 בנתיב – כיוון שאינו בתיקיית `cho` ואני צריך להגיע לתיקיית האחות `src`, אני עלה לתיקיית האב עם `..` וזה יורד לתיקיית `src` ומשם לקובץ `module.js` שהוא קובץ ה'אואהסקריפט' שבו נמצא המודול שלו.

השלב החשוב ביותר הוא לנקות את הארגומנט באמצעות האיבר השלישי של `process.argv`. המערך שיש ב-`process.argv` הוא מערך של כל הארגומנטים. האיבר הראשון הוא `node`, האיבר השני הוא שם הקובץ הרץ והאיבר השלישי הוא הארגומנט הראשון. בהמשך – הדפסה מהירה של מה שקיבלתי בקונסולה לבקרה וזה קריאה למוגדרת `decrypt` של המודול שלי עם מה שקיבלתי מהארגומנט.

החלק הנוסף הוא הוסף decrypt ל-bin שיש ב-package.json

```
{
  "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 הקלדה

npm link

בתיקיה הראשית של הפרויקט. הקלדה זו בעצם מאפשרת למודול להיכנס למודולים הגלובליים ולקשרים למערכת הפעלה. זהה. אם אני אכתוב decrypt ובארגון השם 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. בחים האMITים מומלץ מאוד להשתמש במודולים קיימים ל-crypto. נספ על כך, אני מפנה את תשומת ליבכם אל scryptSync ועל הועבה שמדובר בפונקציה סינכרונית שהיא בעייתית בכל מקום אחר שהוא לא CLI.

פרק 14

# סוקטים - SOCKETS



# Sockets – סוקטים

עד כה למדנו תקשורת עם פרוטוקול HTTP, הכולמר שרת אינטרנט, זהה שמאזין לכל ניסיון תקשורת ומוחזיר תשובה. כל מי שגולש באינטרנט מכיר את הפורמט זהה. אני נכנס לאתר, האתר שולח לי את המידע זהה. הפרוטוקול של HTTP הוא Stateless, זאת אומרת שהקשר בין בין השירות מסוים ברגע שהользоватל נשלח חזרה אל מבקש הבקשה.

אבל בראשת המודרנית לפעמים אלו אין צורך לשמור על קשר עם השירות. יש לנו כל מיני דרכים לעשות את זהה. הראשונה היא "עוגיה". קובץ טקסט שנשמר בדף (צד הלוקה) ונסלח בכל בקשה אל השירות וכן מהשירות. כך השירות יוכל ליצור בין המשתמש לפועלות קודמות שלו באתר. השנייה היא XMLHttpRequest, בקשות שנשלחות באמצעות ג'אוויסקרייפט מהדף אל השירות (עם עוגיה או בלי) וambilות לנו מידע. אם יש לנו אתר חדש למשל, שורצוה להעתיקו כשהוא פתוח, סקרייפט ג'אוויסקרייפט יdag לשלוח כל פרק זמן מסוים בקשה לשרת ברגע לחידשות האחוריות. השירות יחזיר לו את החידשות האחוריות והוא יחליף את התצוגה של החידשות באתר. דרך נוספת וינה יותר היא `polling long`, שבה בעצם השירות היה מעכבר אצל השלמת בקשה HTTP.

אבל בסופו של דבר, ב-AJAX, שזו השיטה הפופולרית ביותר, השירות סטטי – אלו תמיד מאזינים לתקשרות. לעולם לא שולחים. רק הדף או שירות אחר מסוגל לשולח בקשות לתקשרות, באמצעות טעינה של הדף (הקלדה של שם המתחם או פשוט ללחיצה על F5) או באמצעות שליחת בקשה AJAX. יש להזה יתרונות, כי קל מאד לנצל כך כמות גדולה של משתמשים. אך יש מערכות שפותחו ה-AJAX פשוט לא יעבד עבורי. חשבו למשל על מערכות רפואיות או כלו שחשוב שתהיה בהן תקשורת בזמן אמת. במערכות כאלה גם שיגור בקשה AJAX מדי שנייה עלול לא להיות מספק.

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

למה אנחנו צריכים צהו תקשורת? חשבו למשל על צ'אט, כמו פיסבוק מסנג'ר. אני יכול למשח צ'אט צזה עם בקשות AJAX. הכולמר הלוקה פותח את הצ'אט וזה הדף משגר בקשות לבדוק אם יש הודעות חדשות מדי דקות. זה אפשרי, אבל זה מאד לא יעיל. אם יש לנו אלפי משתמשים, הם ישגרו הררי בקשות לשרת, שרובן לא נחוצות. יש אפליקציות ניטור ובקלה שצרכות תקשורת מיידית. לא בטוח ש-AJAX יסיע, כי יש דילוי, למשל, בתוכנות מסחר בבורסה ושאר אפליקציות צרכיות תקשורת דו-כיוונית.

לא תמיד סוקט עדיף. יש כמה יתרונות ל-AJAX – קל להשתמש בו, אין מגבלה תיאורטית לכמות המשתמשים ובעולם השירותי הענן מואוד קל לפרש עוד שירות מאשר להשתמש בסוקט, שהזה תקשורת יקרה יחסית. סוקט הוא כל שחייב להכיר וללמוד, אבל הוא לא תמיד הפתרון האפשרי הטוב ביותר ביותר.

את סוקט אנו ממשמים באמצעות מודול `net`, שאותו אנו מכירים. זה מודול שמננו המודול `http` יורש. במודול `http`, להזכירם, השתמשנו לייצר שרת אינטרנט בפרק על יצירת שרת אינטרנט. המתוודות של `http`, כמו `http.createServer`, והארועים מגעים ממודול `net`. הם כבר אמרוים להיות מוכרים לכם. הכל מאד דומה לשרת `http`, למעט הבדל אחד ממשמעות: שרת `http` יוצר אובייקט שנקרא סוקט. האובייקט זהה הוא סטרים (על סטרים למדנו בפרק קודם) שאפשר לכתוב אליו ולקרא ממנה – כלומר סטרים של כתיבה ושל קראיה, זהה שונה ממשמעות משרת `http` שעבד עם סטרים שאלי כתובים בלבד. פה יש לי ערך תקשורות דו-כיווני שאני יכול לכתוב אליו או לקרא ממנו.

תקשורת TCP לא עובדת עם דפסן אלא עם תוכנות אחרות. יש מגוון עצום של תוכנות שאפשר לעבוד מולן. אני אציגים באמצעות תוכנה שנקראת `telnet`, שמתחברת לסתוקטים. אם אתם משתמשים בליינקס, התוכנה זו כבר קיימת אצלכם כברירת מחדל. אם אתם משתמשים במק, אפשר בקלהות להתקין אותה עם `brew` באמצעות הפקודה `brew install telnet`.

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

```
pkgrmgr /iu:"TelnetClient"
```

לחברת מיקרוסופט, שעומדת מאחורי חלונות, יש הסברים מפורטים באתר שלהם על התקנת תוכנת `Telnet`. אם השורה הזו אינה עובדת עבורכם, תצטרכו לחפש בגוגל כיצד מתקינים `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 לבדוק אם התוכנה הותקנה בהצלחה, הקלידו בטרמינל שלכם:

```
telnet -help
```

עתה לחצו על אנטר. אם ההתקנה הייתה תקינה, תראו הסבר על פועלות התוכנה ותוכלו להמשיך. התוכנה מאפשרת לנו לפתח חיבורים שונים לשרתים שעובדים עם סוקטים. הנה ניצור שרת זה.

כאמור, אנו משתמשים במודול של net בדיק כפי שהשתמשנו במודול זהה כדי ליצור שרת HTTP. נכניס את השירות שלנו למודול משלנו. ניצור אפליקציית js.js עם package.json. ניצור לה תיקייה src ובדור תיקייה \_\_-cs \_\_ ניצור תיקייה שנקראת 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)` כדי שנדרש ממנו תמיד כשאני יורש. אני מזין לפורט 6969 (בדיקה כמו במודול `http`) ומזין לאירוע שנקרא `connection`. האירוע הזה מופעל כאשר מישו מתחבר לשרת שלנו, בדיק כמו האירוע המקביל ב-`http`.

האירוע מנהל עם `connectionHandler`. פה יש הבדלמשמעותי מ-`http`. בעוד המודול של `http` עובד עם `request` (התגובה של השירות) ו-`response` (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 בדיק כמו כל סטרים דו-כיווני שלMANDNO עליון בפרטים הקודמים. מה אני עשה זה לכתוב לסטרים הזה "ברוך הבא לשרת שלי" ולהזין לאירוע `data` שמוחפעל בכל פעם שmagiu איזשהו מידע מהמשתמש. זהה הדבר הייחודי שיש לנו בסוקטים – היכולת לעשות דברים כשהמשתמש מקליד. כשההמשתמש מעביר משהו, השירות מפעיל את מתודת `dataHandler` – במקרה זה היא רק מדפסה לკונסולה.

אני מבצע Now לקלואס זהה ומוחזיר אותו ב-`module.exports`. זהו, יצרתי את המודול הזה. בתיקייה הראשית של הפרויקט שלי, אני אוצר קובץ של `js.app` בו אפעיל את המודול שיצרתי. איך? באמצעות `require`:

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

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

```
telnet 127.0.0.1 6969
```

הפקודה הזאת מורה לטלנט להתחבר ל-IP המקומי 127.0.0.1 ולפורט 6969. אם עשיתו הכל נכון, אני אראה את הכתובת `Welcome to my server` בתוכנה – אני אראה מיד בקונסולה של השרת, בטרמינל השני.

הכתובת המקומית שלנו היא תמיד 127.0.0.1 – זהה כתובות שמורה המיעדת למחשב המקומי. המחשב המקומי נקרא גם `localhost` ואנו יכולים להקליד גם:

```
telnet localhost 6969
```

זה יעבוד.

ב-`Visual Studio Code`, שעליו כאמור למדנו בפרק הראשון, אפשר לעבוד עם טרמינל מפוצל ואז כל מואוד לראות גם את צד השרת שיצרנו והפעלנו וגם את צד הלקוח.



The screenshot shows a split terminal window in Visual Studio Code. The left terminal window displays command-line output:

```
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

```

The right terminal window displays the text:

```
Welcome to my server
dddfgh█
```

מקובל ליצור שרת של סוקט כדי ללמד ולהבין איך סוקט עובד. בסופו של דבר, שרת סוקט נשמע מאוד מפוצץ 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 נוח.

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

**Socket.io** הוא מודול מרכזי שנמצא ב-NPM. יש לו גם אתר נאה שאפשר ללמוד ממנו וכל דוגמאות בכתובות – 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 וההסברים שמסופקים בדרך כלל ביד נדיבה ורחבה. יש כל כך הרבה מודולים מרכזיים ב-Node.js ספר אחד או אפילו סדרת ספרים לא יכולים לכטוט אותם. באתר, למשל, בזמן כתיבת שורות אלו, יש הסבר מكيف איך ליצור אפליקציית צ'אטים מキפה. אני ממליץ לכם, לאחר קראת הספר, לנסות לבנות אותה.

יש מתכנתים שיכולים להעביר קריירה שלמה ב-`js`-Node ללא שימוש של סוקט, אבל זה חלק חשוב מ-`js`-Node וגם סוקטים עומדים מאחוריו כמו אתרים ותוכנות חשובות וכך לתרגל ולהבין זהה לאקסם – אלה סוקטים.

#### תרגילים:

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, וצרו `setInterval` שמשדרת בכל שנייה את ה-`time`.

#### פתרונות:

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

בתוך `this.connectionHandler` אנו מבצעים קריאה לפונקציה `snickerat repeater`, ככלומר מפה זה ג'אווהסקייפט כמעט טהור. אנו יוצרים `setInterval` שמופעלת מדי שנייה. דנו בפונקציה זו בפרק על טימרים והיא נלמדת גם בספר "לימוד ג'אווהסקייפט בעברית". זו מקבלת שני ארגומנטים, אחד מהם הוא פונקציה שמבצעת כתיבה ל-`socket` ומדפיסה בכל שנייה את הזמן לפי הפונקציה שניתנה בתחילת התרגילים. שימושה ב-`toString` כדי להמיר את המספר למחוזת טקסט שרק היא יכולה לעבור בסוקט.

אנו נוצר את המודול זהה באמצעות `require` ב-`js`:

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

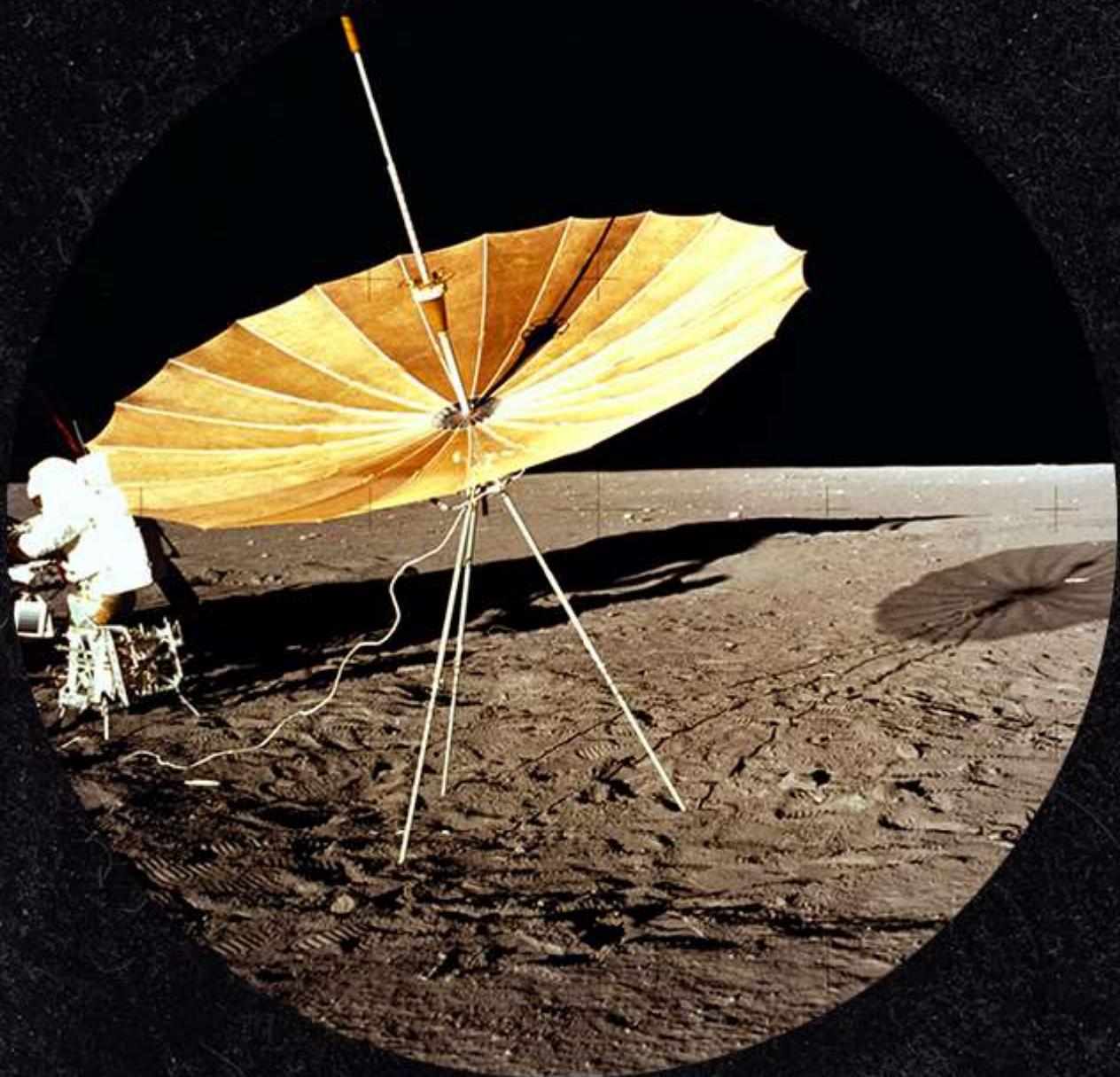
ונפעיל את השרת באמצעות `node ./app`. כדי לבדוק את השרת, נפתח טרמינל נוסף ונכתב בו:

```
telnet 127.0.0.1 6969
```

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

פרק 15

# קוריאת מושבים PATH באמצעות נזודול



# קריאה משאים באמצעות מודול path

עד כה, כשקרנו 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 שמציג קובץ `index.html`. בואו ניצור שרת פשוט שקורא לקובץ `index.html`.

ניצור תיקייה בשם `simple-client` ובתוכה ניצור תיקיית `src`. בתיקייה זו ניצור תיקייה שנקראת `modules`, שם יהיו כל המודולים שלנו. תיקייה נוספת תחת תיקיית `src` תקרא תיקיית `static` ושם ניצור את דף ה-HTML של האפליקציה שלנו. נתחיל את הפרויקט שלנו בהקלה `init` 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 זהה. איך עושים את זה? מרים שרת פשוט, בתוך מודול. כבר למדנו איך עובדים עם שירותים 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 גודלה: **WebServer**.

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

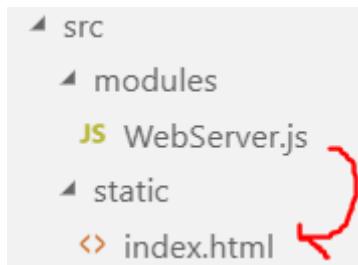
class WebServer extends httpServer {
  constructor() {
    super();
  }
}
```

```

  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 לקלואים של מודולים שלמדו לנו בעבר. שווה להתעכבר על ה-`fs.createReadStream`. אני צריך להקליד את הנתיב שבו הוא נמצא. המודול נמצא בתיקיית `modules`. הקובץ `index.html` נמצא בתיקיית `static`.

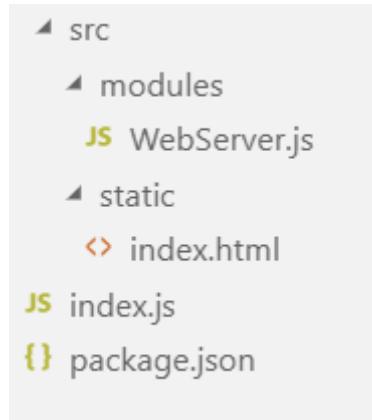


air אני מגיע אליו? פשוט שבסוטרים מהמקום הנוכחי שלי/. אני עליה באמצעות `..` תיקיית `src` ומשם יורד אל `index.html`. הנתיב שאני צריך להכין הוא:  
`./../static/index.html`

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

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

מבנה האפליקציה אמור להיראות כך:



נՐץ את האפליקציה כולה באמצעות `node index.js`. ניכנס ל-`localhost:3000` ונראה שהטרמינל מתפוץץ מהודעתה שגיאה. הודעת השגיאה טעונה שהשורה הזו:

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

לא עבדת כי המשאב לא נתען. איך זה יכול להיות? התשובה היא `fs.createReadStream` מחשבת את הנתיבים מנוקודת ההרצה. כלומר, אם אני מՐץ את הפרויקט מהתיקייה הראשית שלו, `Node.js` תחשב את הנתיב בצורה שונה מחלוטין.

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

הקבוע `__dirname` הוא קבוע שתמיד מחזיר את המיקום הנוכחי של הקובץ. אם אני מՐץ אותו בכל קובץ שהוא, הוא תמיד יביא לי את הנתיב המלא של הקובץ. אם הקובץ נמצא בcone C בתיקייה `barzik`, הוא יחזיר את המיקום זהה. זהו דבר שימושי מאוד כאשר אני רוצה להגיע למשאים בתת-תיקיות. במקרה הזה אני רוצה להגיע למשאים בתיקיות אחרות ועדיין רוצה להשתמש בנתיב יחסית כמו `../` למשל.

בדוק כאן יש `path.join`, שמקבלת שני ארגומנטים ומרכיבה משנייהם נתיב אחד, בעוד הארגומנט השני יכול להיות נתיב יחסוי. כלומר אם אני רוצה לקרוא לקבצים מתיקיות אחרות, אני צריך להשתמש ב:

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

וכמובן לא לשכוח לעשות `require` למודול `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();
```

למודול `path` יש עוד כמה מוגדרות חשובות שצדאי להציג בדוקומנטציה בעברית – אבל השימוש שלו עם `__dirname`, הקבוע הגלובלי שיש ב-`js.Node`, הוא שימוש שתראו המון בקטע' קוד וגם כתובו בעצמכם. נוסף על כך, יתרון גדול נוסף למודול `path` הוא סיווע שימושו בבדלים 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 ומעליהם את סקריפט `js.Node` שלכם לשרת מבוסס לינוקס. שימוש ב-`path` יכול לסייע את הבעיה.

פרק 16

# PACKAGE.JSON SCRIPTS



# package.json scripts

עד כה, כשרצינו להפעיל את התוכנה שלנו, נאלכנו להקליד באופן ידני `node ./app` או `node ./index`. כמו בדוגמה שבערך 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 להכיר אותו. במקומות שבהם אומרים את הראש אם להקליד `node ./app` או כל `entry point` אחר, הוא יכול להקליד פקודה אחידה וננו נגדיר בדיק מה היא עשו.

הפקודות האחידות נמצאות מתחת ל-`scripts` בקובץ `package.json`. בדוגמה הקודמת, הפעלנו את השרת שלנו באמצעות הקלדה של `node ./index`. אם נכניס את הקוד שלנו תחת `start` באופן זה:

```
{
  "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 `start` ומקה – הפקודה תריץ את כל מה ששמנו ב-`start`. במקרה זה:

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

למה זה שימושי? כי יש עוד פקודות וhoneks (Hooks) שאפשר לעשות איתם המונע דברים שימושיים ב- scripts npm. למשל ההוק prestart npm שרצה תמיד לפני שהוא מרים start npm. כמו, למשל, אני מבקש ממנו להציג הודעה:

```
{
  "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 npm, לפני שהתוכנה תרוץ, היא תכתוב הודעה של READY TO START בדצמ"ל. מובן שלא משתמשים ב-prestart כדי להציג הודעות. אם יש לי שרת, אני יכול לבצע בדיקות תקינות על הסביבה, כיווץ מסמכים ומשאים וניקוי כללי, ובעצם להריץ כל סקריפט שבא לי להריץ עם פקודה מסוימת.

## סקריפטים עם שמות

אנו לא חייבים להימנע לשמות של NPM אלא יכולים להשתמש בכל שם שהוא. כך למשל, אני יכול להחליט שיש לי סקריפט שנקרא `ahla` וכשאקרה לו הוא ידפס 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 `ch` לפני הפקודה. במקרה של `ahla`:

```
npm run ahla
```

שיםו לב שהיבים להשתמש ב-`ch` כאשר מרכיבים כל פקודה שהיא לא חלק מסט הפקודות המובנות ב-NPM. יש מעט פקודות מובנות שלא דורשות `ch` לפני הקידמת הפקודה, והבולטות שבהן הן:

הסבר על הפקודה	פקודה מובנית ב-NPM
מתקינה את כל ה- <code>dependencies</code> ב- <code>package.json</code>	<code>install \ i</code>
מרכיב את פקודה <code>test</code> המפורטת ב- <code>package.json</code>	<code>test \ t</code>
מרכיב את פקודה <code>start</code> המפורטת ב- <code>package.json</code>	<code>start</code>

## משתני סביבה

אחד הדברים שימושי לעשוט עם `ch` ומקו הוא לעבוד עם משתני סביבה ולהשתמש בהם בקוד. משתני סביבה הם קבועים נשיים — משתנים שמשמעותם מסביבת הרצה ולא מהקוד. אנו משתמשים במשתני סביבה בתנאים רגילים שאנו לא רוצים להכניס לקוד או בתנאים שעולים להשתנות מסביבה

לסבירה, למשל פורט. בסביבת הפיתוח, פעמים רבות אני רוצה להרים את השרת שלי בפורט 80 שהוא הפורט של ברירת המחדל. אך בסביבת הפיתוח, הסביבה האמיתית, אני רוצה להרים את השרת שלי בפורט ששרת הדיפולימנט מגידיר לו. איך עושים את זה? זהו הקלאס של שרת `http` שעבדנו עליו בפרק הקודם על `path`. בכל הדוגמאות של שרת `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;
```

air אני קובע את ממשתנה הסביבה?

## קביעת ממשתנה סביבה דרך הסקריפט

אפשר לקבוע ממשתנה סביבה דרך הסקריפט של `package.json` באמצעות פקודה `set` באופן פשוט במיוחד – הצבת

```
set PORT=3000
```

לפני הפקודה ושרשור 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 סקורייפטים: start ו-dev. כל אחד מהם רץ עם פורט אחר. אם אקליד npm start או npm run start היכנס לשרת ה-<http://localhost:3000> רק מ-localhost:3000 ואם אקליד npm run dev או npm run dev היכנס לשרת ה-<http://localhost:80> רק מ-localhost:80.

## קביעת משתנה סביבה דרך הגדרות מערכת הפעלה

הדרך הקומונת תעבור אך בהרבה מקרים היא בעייתית – במיוחד במקרים מסוימים ורגילים. לא מעת פעמים אנו צריכים טוקנים – סוג של סיסמאות המשמשות לחיבור למסדי נתונים ולשירותים שונים. אנו לא יכולים לאחסן אותם בקובץ מילוי סיבוב – אבטחת מידע ו咎 עניינים אחרים – לעיתים הטוקנים משתנים. בדיק בשביל זה משתמשים פעמים רבות על מערכת הפעלה.

משתני סביבה הם לא "יחודיים ל-`js.Node`" ויש לא מעט תוכנות שימושכיות עליהם. לכל מערכת הפעלה יש כמה וכמה דרכי להגדיר משתני סביבה.

### חלונות – CMD

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

### חלונות – הטרמינל של Visual Studio Code (ידעו גם כ-`cmd`)

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

### לינוקס/מcker

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

מודול `env-cross`, שלא אלמד עליו פה, מסייע לתוכנים ב-`js.Node` לנוהל משתני סביבה במנוי פלטפורמות. כדי להכיר אותו ואפשר למצאו עליו מידע ב-[npmjs](https://www.npmjs.com/package/cross-env):

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

## קביעת משתנה סביבה דרך קובץ

דרך נוספת לנוהל משתני סביבה היא באמצעות קובץ配置 (נקודה בתחילת שם הקובץ ולא סימנת קובץ). מדובר בקובץ שישוב 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 ולקרא לו באופן הבא בסקריפט שלכם:

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

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

אר מובן חשוב מאוד להקפיד שלא לשמר את זה. חלק מקובצי הפרויקט שלכם!  
בכל האפליקציות והאתרים מושנים את התנהוגות באמצעות משתני סביבה. משתנה הסביבה החשוב ביותר, וזה שמשתמשים בו הכי הרבה, הוא NODE\_ENV שיכל להיות development או production. תלוי בסביבה – בסביבת הפיתוח, המחשב שלנו, הוא יהיה development. בסביבה  
שבה האפליקציה שלנו עבדת הוא יהיה production.

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

## dev dependencies

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

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

בדוק בשיל זה קיימ **dev-dependencies**. מדובר במודולים שאנו משתמש בהם אך ורק בסביבת פיתוח. כאשר מתקין מודול שנדרש אך ורק לסביבת פיתוח, אני אתקן אותו באמצעות הקלהה של הפלג —`dev-dependencies`. הקלהה הזאת מכניסה את המודול שהותקן ב-package.json.

הבה נדגים. באפליקציית השרת שלנו, שעלה 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": {
  "eslint": "^5.16.0"
}
```

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

**תרגילים:**

התקינו את eslint בפרויקט שהוסבר עליו בפרק הקודם. הקפידו שהוא מותקן בסביבה חיה. צרו קובץ הגדרות של בדיקת תקינות באמצעותו-- eslint --fix. צרו משימה שתבצע בדיקה סטטית של הקוד באמצעות lint run מוקה.

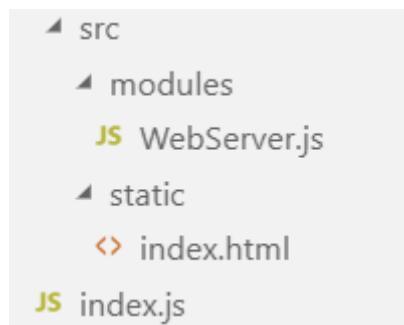
רמז: אפשר לבצע בדיקת קוד לאחר יצירת קובץ הגדרות באמצעות:

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

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

**פתרונות:**

מבנה התקינות של הפרויקט שלי נראה כך:



בתיקיית 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 שנותע:

```
<!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-`js.index` נמצאים הקריאה למודול וטעינתו:

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

תקן את eslint בפרויקט שיש לו `package.json` – ואם קראתם את הפרק הקודם וביצעתם את התרגילים, אמור להיות לכם `package.json` בפרויקט. במידה שלא, בום `init` ייצור יישור לכם `package.json` וaz תוכלו להתקן את eslint. אנו מתקנים 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 שנענה על כל השאלות, נראה שבתקייה הראשית של הפרויקט נוצר קובץ הגדרות של eslint. כדי לבדוק את העניין, נקליד בטרמינל:

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

אם הכל עובד, נראה את כל השגיאות הסגוניות שיש בקוד שלנו. כדי להכין את הפקודה לסקריפטים, ניצור ב-package.json תחת scripts את המשימה lint עם השורה שmaps להeslint את:

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

קובץ ה-package.json שלנו 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",
  }
}
```

## 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 מאוד, 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 בספרייה העשירה של המודולים שיש ב-NPM. אקספרס הוא בדיק מודול זהה שמאפשר לנו להרים שרת אינטרנט בקלות ויתר מכך – להוסיף לו פונקציונליות רובה ובקלה. בשורה אחת אני יכול להוסיף לאקספרס נתיבים, התנהלות, מודולי אבטחה, ניתוב, לוגים ועוד המון פיצרים שהיינו צריכים לכתוב המונן שורות כדי למש אתם בעצמינו.

בפרק זהה אני אסביר מהו אקספרס. המודול עצמו מורכב מאוד והוא יכול לאכלס ספר שלם, אך לא יוכל לסקור את כל הפונקציונליות, אבל כן אכסה את הדברים הבסיסיים מתוך הרצה שכשתרצו להרchieb – 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 ייצרת תיקייה 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? `express` או `npm` – נוכל לראות שאקספרס נוצר ב-`package.json` ב-`dependencies`. עכשו אפשר להשתמש בו ב-`app.js`. הבדיקה 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 `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 שאננו כבר מכירים. ראשית אנו קוראים למודול ואז אנו יוצרים את השירות באמצעות קרייה לפונקציה הבונה במודול `express` ויוצרים הפניה לשרת בקבוע `app`. נודיע לשרת להאזין לפורט 3000. מהרגע זהה אנו יכולים לבצע בעצם האזנה לتنועה שmagua מחדף במתודת GET לנטייבים שונים.



ה-`req` וה-`res` הם אובייקטים. ה-`req` שהוא קיצור ל-`request` – בקשה, הוא הקרייה המגיעה מrangle (לומר הדף) – הכוללת פרטים שונים (כמו למשל `headers` או `body`). ה-`res` שהוא קיצור ל-`response` – תגובה, הוא התגובה המגיעה מהשרת לrangle, הלא הוא הדף, וגם פה יש פרמטרים שונים.

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

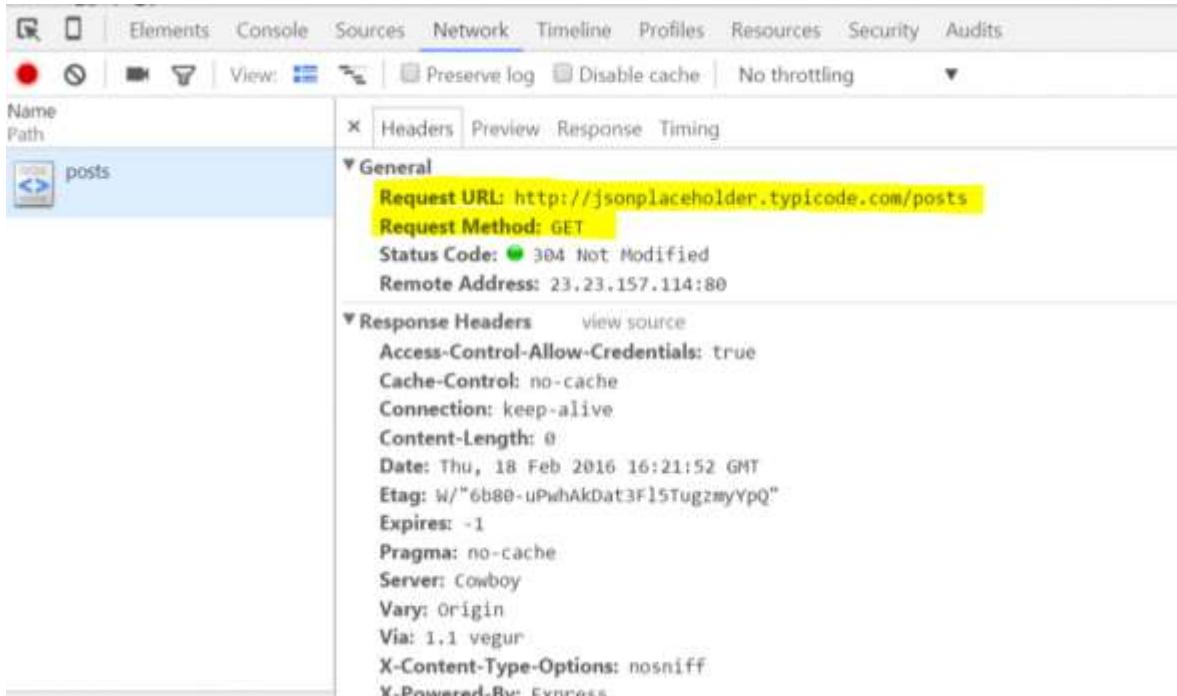
<https://expressjs.com/en/4x/api.html#res.send>

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

## טיפול במתודות של בקשות HTTP

עד עכשו דיברנו על בקשות GET. בפרוטוקול HTTP יש לנו כמה וכמה סוגי בקשות. אנו מכירים את בקשה GET שהדף מושג כאשר אתם גולשים ל-URL מסוים. כשאני נכנס לאתר, כמו 3000:localhost או google.com או localhost:3000 או localhost:3000/get. זה חלק מהפרוטוקול של HTTP. גם בדוגמה שלעיל אני מראה איך אני מאמין לתנועה של GET עם מתודת `get`.

אם תשתמשו בכל הפתחים בדף, תוכלו לראות את בקשה GET בקלות בתוך TAB ה-`:Network`



Request URL: <http://jsonplaceholder.typicode.com/posts>

Request Method: GET

Status Code: 304 Not Modified

Remote Address: 23.23.157.114:80

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 שהדפסן יכול לשלוח באמצעות ג'אווהסקריפט שנמצאות הצד הלוקה.

סוג בקשה	פירוט
<b>POST</b>	בקשה ליצירה – יכולה להכיל payload – כלומר מחרוזת טקסט, מספר או אובייקט שנשלח לשרת. השרת יכול ללקח את המידע זהה ולהשתמש בו ליצירת משאב כלשהו.
<b>PUT</b>	בקשה לעדכון – הבקשה יכולה להכיל payload בדיק כמו POST. השרת אמור ללקח את המידע זהה ולהשתמש בו לעדכון משאב כלשהו. במתען או ב-URL שאלוי הבקשה נשלחת יהיה ID של המשאב שאותו מעדכנים.
<b>DELETE</b>	בקשה למחיקה – בדומה ל-GET אין כאן payload. ב-URL שאלוי הבקשה נשלחת יהיה ID של המשאב שאותו מוחקם.
<b>PATCH</b>	בקשה לעדכון חלק – הבקשה דומה ל-PUT ומשתמשים בה לעדכון חלק של המשאב.

כאשר אני נמצא הצד הלוקה, אני יכול לשלוח בקשות במתודות שונות אל השרת. השרת, שעליו אני אחראים, אמור לדעת לטפל בבקשות האלו.

אנו יוצרים בקשות כאלה באמצעות פקודה `fetch` בג'אויסקייפ בצד הלקוח, ככלומר 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));
```

זהו פקודה שכמתכני ג'אויסקייפ אתם אמורים להכיר – היא מבצעת שליחת בקשה POST אל localhost:3000 עם המידעה:

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

אם תדביקו אותה בקונסולה של כל המפתחים, תקבלו שגיאה. מדוע? כי השרת שיצרנו יודע להתמודד רק עם בקשות GET ולא עם POST. אם נשתמש במתודת `post` הוא ידע לטפל בבקשתות כאלה.

היכנסו את הקוד הבא ל-`app.js`:

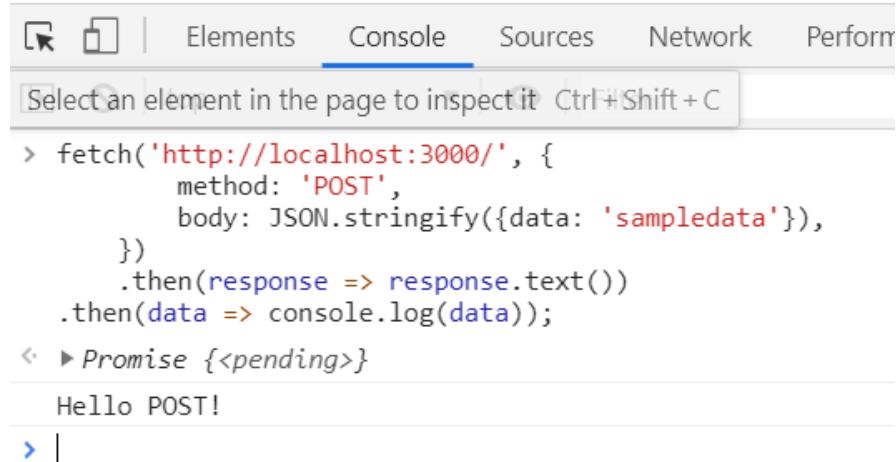
```
const express = require('express');
const app = express();

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 התחתונות 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 app.js). אדביך את הקוד ששולח שינויי הקוד לא יעדכנו ולא יריצו). אז אפתח את localhost:3000 בדף. אדביך את הקוד ששולח בקשה POST ואוכל לראות שבקונסולה של הדף אני מקבל `Hello POST!`.

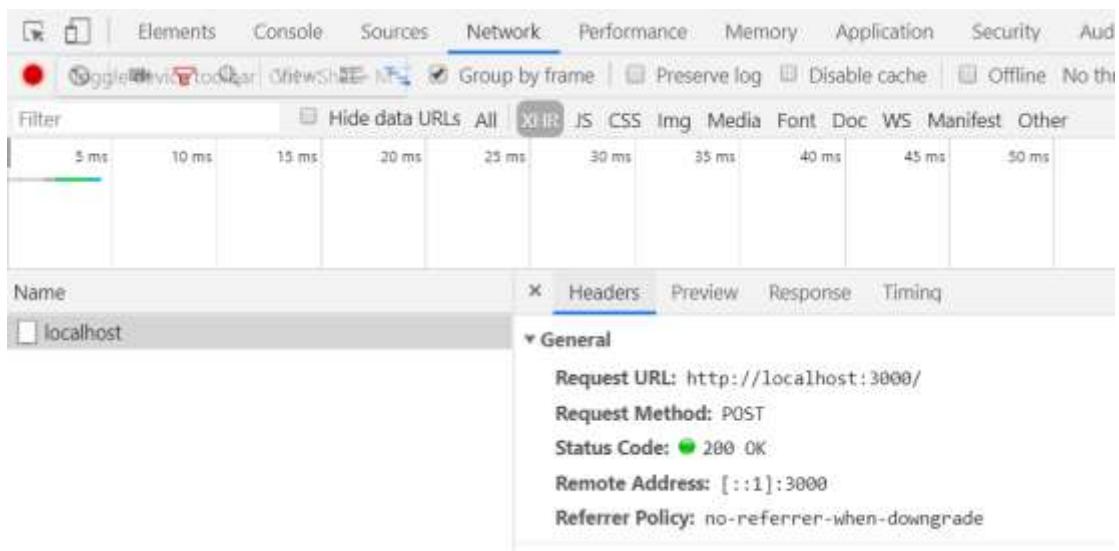


```

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.



air נתפסו את הבקשות האלה? כמו שיש לנו get כדי לטפל בבקשת GET, יש לנו post כדי לטפל בבקשת POST ובהתאםה patch, put, delete ו-all וגם all, כדי לטפל בכל הבקשות. שימו לב שכשמדובר במתודות של HTTP אנו כותבים באותיות גדולות: למשל, GET, POST, DELETE. כשמדבר על מתודות של express אנו כותבים באותיות קטנות. express נדרש לזכור של מרבית שמודול שמודול את הכל – עדין מדובר במודול שזהה מבחינה תפקודית ו מבחינה שימושית לשרת-HTTP הבסיסי שיצרנו. כך למשל, אם אנו רוצים להחזיר דף HTML בסיסי (ולא סתם טקסט של Hello World (Hello)) – אני עושה את זה בדיקן כמו בשרת רגיל. אם למשל ארצה לשחרר את הדוגמה שהראיתי בפרק על טעינת משאבי באמצעות path, ולאחר מכן המשמש דף HTML סטטי, שנמצא בתיקייה static. אני אעשה זאת באמצעות קרייה של קובץ והזרמתו למי שנכנס לאתר. קובץ ה-.js.kmp של 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 שאין קורא את התוצאה אל סטרים באמצעות מודול `fs` ומשגר אותה באמצעות `pipe` אל `res` בדיק כמו שרת HTTP בסיסי וטבוי של Node.js. בהנחה שהקובץ `index.html` באמת קיים בנתיב `src/static` – אני אראה את הקובץ אם אפעיל את השרת באמצעות `node ./app.js` וএসে עם הדף לכתובת `localhost:3000`. זה כמובן בסיסי מאוד ולא משוו שמיובל לעשות. באקספרס יש לנו מנועי רנדומ שעליהם נרחיב בהמשך הפרק.

## ראוטינג

בתת-הפרק הקודם טיפלנו במתודות, ועשינו נטפל בנתיבים. כewish ל' אתר קטן עם כמה נתיבים, למשל עמוד הבית ודף `help`, לא צריך לשבור את הראש. ה-`app` שלו יכול להכיל כמה נתיבים:

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

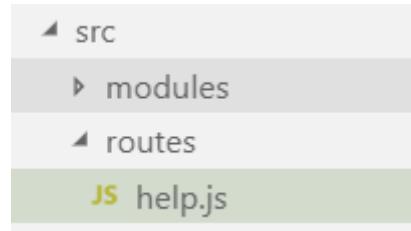
app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

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 כewish ל' כמה וכמה נתיבים סטטיים? 10? 100? 1,000? לשניכם את הכל לתוך רשימה ארוכה ב-`app` איננו רעיון טוב. מה עושים? בדיק בשביל זה יש לנו רואוטינג. רואוטינג, או ניתוב בעברית, זו פשוט דרך לנתח את הבקשות השונות לקבצים שונים. זה עובד בדיק כמו מודולים. אנו יוצרים מודול של רואוטינג ומיצאים אותו. מי שוצרר אותו הוא האפליקציה שלנו ב-`app`.

כדי להציג, ניצור רואוטר לדפי עזרה. מקובל לשים את כל המודולים של הרואוטרים בתיקייה שנקראת `:routes`



כאן יצרתי את `help.js`. זה מודול שבו יהיו נתיבים שונים שהיו מתחת ל-`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 כמו 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? במודולים של ריאוטינג אני משתמש ב:

```
const router = express.Router();
```

ואת כל הריאוטינג אני עושה באמצעות router.get ו�א app.get. בהמשך אני מיצא את המודול זהה  
כמו מודול רגיל.

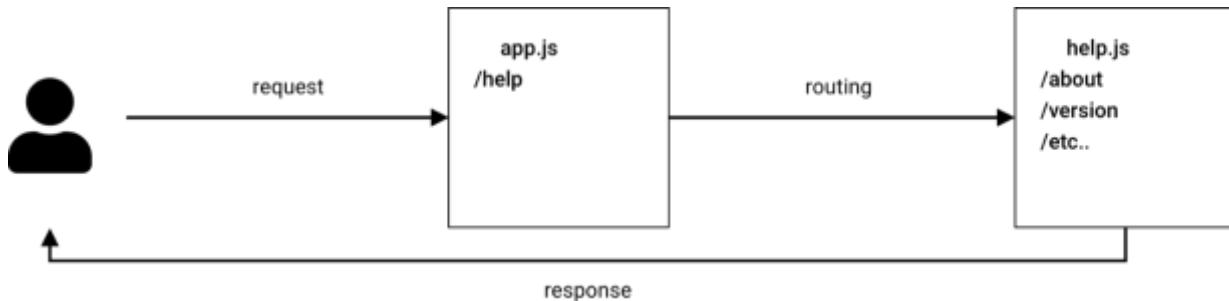
air אני צריך אותו? באמצעות `use`.app – זהו מתודה שנמצאת באופן טבעי ב-app של אקספרס ואפשר להשתמש בה ממש בקלות. היא בעצם מקשרת בין הקוד שרצ ב-app לבין הראטור הרלוונטי, במקרה שלנו `help`. ב-`use`.app אני מקשר בין כתובות הנתיב, שהיא טקסט פשוט, כמו, למשל, שהוא משתנה שמכיל רפרנס לבין הראטור:

```
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-`js`.app הגדרנו את הנתיב `/help`/ וצינו שמי שמתפל בו הוא הראטור. הראטור של `help` מתפל בשני נתיבים: `about-me` ו-`use-how-to`, והם יהיו זמינים מיד אחרי הנתיב שהראטור מתפל בו – במקרה זה `help`, ככלומר הכתובות `/about-me` ו-`/help/about`.



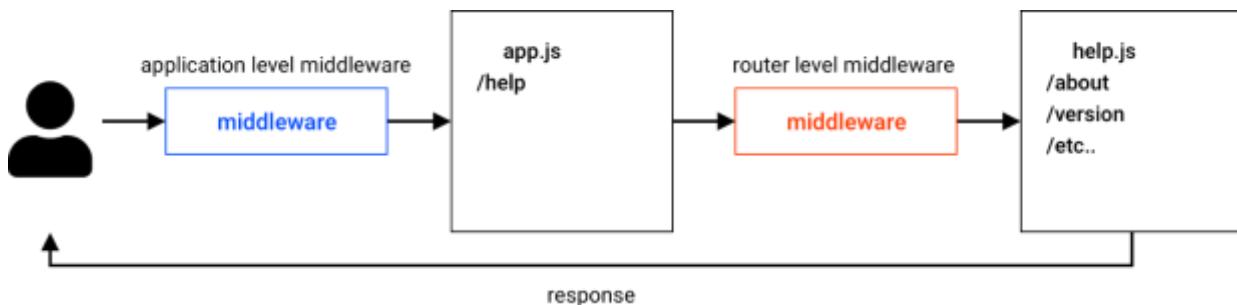
ה-`use`.app, שבו או משתמשים כדי לחבר את מודול הראטור שלנו אל הנתיב, הוא מתודה שבה אנו משתמשים כדי לחבר רואוטינג אבל גם `Middleware`.

## MiddleWare

את הקונספט של Middlewares ניתן ביצוע באמצעות akseptor. MiddleWare זה ייחידת תוכנה שהבקשות והtagיות עוברות דרךו ואנחנו יכולים לבצע פעולות שונות. למשל, אם יש לנו מערכת לוגים, אני משתמש ב-`Middleware` על מנת לבצע רישום של בקשות במערכת. אם למשל אני מקבל מהמשתמש תമונות ואני רוצה לכווץ אותן לפני שאני כותב אותן לדיסק הקשיח – אני אעשה את זה ב-`Middleware`. העיקרונות העומדים מאחורי `Middleware` הוא בעצם לקבל תנועה, לטפל בה ואז להעביר אותה להלאה להמשך טיפול.

מדובר בעצם בפונקציה שמקבלת שלושה ארגומנטים: (`next, res, req`). הארגומנט הראשון הוא הבקשה, `req`; השני היא התגובה – `res`. כבר הסבכנו ודנו באובייקטים הללו מוקדם יותר בפרק.

`next` הוא פונקציה שאנו קוראים לה בסיום העבודה שלנו כדי להעביר את התנועה להלאה. אנו יכולים להשתמש ב-`Middleware` בכל שלב באקספרט.



הבה נדגים באמצעות `Middleware` שמבצע לוגינג. בפרויקט שלנו ניצור תיקייה בשם `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;
```

שים לב שיש לנו כאן שלושה פרמטרים – `req, res, next`. אני יכול בשלב זהה לקרוא את הבקשה מהלוקה, ולכתוב לתגובה שהוא קיבל בעתיד. במקרה זה אני רק כותב ל-`log` ומיד אחר כך קורא ל-`next`. מה ש-`next` עושה הוא להעביר את התנועה להלאה. אם אני לא אקרא ל-`next`, התנועה לא תעבור והבקשה לא תושלם, אז זה חשוב. אם יש בקשות, קריואות, כתיבה לדיסק המקומי וכו' – ה-`next` יקרא מיד אחרי שהפרומים יושלמו.

איך אנו משתמשים ב-`Middleware`? פשוט יותר, מבעודים `require` למודול שלנו, במקרה זה – `logger.js` שבתיקית `middlewares`, ובמצעים `use` `app`:

```
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 אשתמש ב-`router`. אם אני אזכיר את ה-`router` הזה בראוטר של ה-`help` שלמדנו עליו בתחילת הפרק, הוא יראה כך:

```
const logger = require('../middleware/logger');

router.use(logger);
```

ב-`Middleware` אנחנו יכולים לעשות דברים בהתאם ל-`request` ולשנות את ה-`response`.  
ובן שהכוכב הגדול של אקספרס, בדומה ל-NPM, הוא שעת רוב ה-`Middleware` אנו לא צריכים לכתוב בעצמנו אלא אנו יכולים להשתמש ב-`Middleware` שנושאים כתבו ומפרסמים ב-NPM. בדיקן כמו כל מודול אחר. פשוט משתמשים במודול זהה אך ורק באקספרס.

למשל – לוגר. בדוגמה שלעיל כתבנו `Middleware` מאפס. אבל במצבים אם נרצה לוגר, סביר להניח שנחפש מודול ב-NPM שעושה את זה, למשל מודול כמו `morgan`.

<https://www.npmjs.com/package/morgan>

ה-`Middleware` זהה הוא מודול של NPM והוא מותקן בדיקן כמו כל מודול:

```
npm i morgan
```

איך משתמשים בו? בדיק כמו `Middleware` שכתבנו בעצמנו, כאשר במקרה זהה (ובכמעט בכל המקרים) יש הוראות מדויקות להפעלה בדף המודול. במקרה של `morgan`, מפעליים 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 הכנסיות עם מעט יותר מידע. אפשר ליצור בקלות לוגים מוחכמים, שאוגרים מידע נוסף או שכותבים לקובץ. איך? הכל בדוקומנטציה של `morgan`. **הדוקומנטציה לא טובה?** `the-logger` לא טוב מספיק? נחשך אחר או שנתרום לו קוד שיפור את המצב.

הכוח המרכזי של אקספרס הוא בספריה העצומה של המודולים שבאה אליו. לפני שאתם מתחילהם לכתוב קוד בעצמכם, אפשר ורצה לבדוק אם יש מודול שמתפל בזה. אקספרס הוא פופולרי ביותר, וסביר להניח שרוב הדברים שחתבתם עליהם כבר פותחו ונמצאים ב-`NPM`.

## URL דינמי

מדובר יותר בפרק הסברנו על רואוטינג וראינו איך אנו מגדירים כתובות שונות לאתר באמצעות אקספרס. כתובות אלו נקראות **כתובות סטטיות**. **למה סטטיות?** כי כתובנו, ממש בקוד, את הנתיבים ברואוטינג. אבל רוב האתרים מכילים כתובות דינמיות. אם יש לנו אתר אליו מתחברים וכל אדם יש פרופיל משתמש, לא נגידיר ידנית עבור כל משתמש, רואוטינג משלה. אם יש לנו אתר שיש בו מאמרם רבים, לא נגידיר עבור כל מאמר ומאמר כתובת ברואוטינג. בבדיקה שביב זה יש לנו כתובת דינמית, כתובת שבה יש פרמטר מסוים שמשתנה כל הזמן ואקספרס יכול לקרוא אותו ולהציג מיד תוצאה שונה לכל משתמש ומשתמש.

הבה נדגים באמצעות רואוטינג של `NAME`/`user`. `the-NAME` הוא בעצם משתנה דינמי, קלומר אם אני נכנס לאתר בכתובת: <http://localhost:3000/user/moshe> אני אzystה לראות התוצאות לפרט `moshe`.

הדבר הראשון שנצרך לעשות הוא להגדיר רואוטינג שיטפל בבקשת שמוונת לנתיב `user`. בתקיית `the-routes` בפרויקט האקספרס שלנו, ניצור קובץ שיקרא `user.js` ונותיר אותו ריק. נקרא לו בקובץ ה-`app.js` שמכיל 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 ש齡יל אמור להיות כמו מוכר מאוד כיון שעברנו עליו מוקדם יותר בפרק. עכשוו ניגש למודול הראוטינג של `user` ונגדיר את ה-URL הדינמי. זה הרבה יותר קל ממה שחושבים. כאשר אנו רוצים נתיב דינמי, אנחנו צריכים פשוט להוסיף נקודותים בנתיב, לפני שם הפרמטר. למשל, אם החלטתי שאני רוצה פרמטר ששמו יהיה `name`, כך אני אקבל 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 בראטור של `user`. קלומר ב-`src/routes/user/`. אני ניגש אל הפרמטר בכל מקום באמצעות `req.params.name`.

בנוסף הוא אובייקט הבקשה שאנו מקבלים ב-`params`. `Middleware`. אחת התכונות שלו היא `req`, `params`. `req` הוא אובייקט הבקשה שאנו מקבלים ב-`params`. `req` הוא אובייקט גנוי שלו כל הפרמטרים שיש בבקשת. קלומר אם יש פרמטרים של POST או GET – הם יהיו שם. את זה לא נראה במודול `http` קלאסי אלא זו פונקציונליות שיש באקספרס, אחת מרבות שעזרו לנו להפוך זהה להפוך לאחד הפופולריים ביותר.

אני יכול לקרוא לפרמטר של כל שם, למשל השם המופיע `ahla` גם הולך – השם עצמו לא משנה כלל, הוא רק קובע איך אני אקבל אותו ב-`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 אראה את הכתוב המתאים.

שיםו 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-router.post או בכל מתודה אחרת.

## מבנה

עד עכשיו החזרנו שני סוגים תגבות – דף HTML סטטי, שלו קראנו באמצעות `fs.read` או תגובה פשוטה באמצעות `res.send`. אבל בחיבים האמיתיים פעמים רבות אנו צריכים להשתמש בתבניות – קלומר שילוב של דף סטטי עם נתונים דינמיים. בדיק בשביל זה אנו צריכים להשתמש בתבניות. יש כמה סוגים תבניות שאקספרס תומך בהם ונו נלמד כאן על `ejs`, הסוג הפופולרי ביותר לאתרים סטטיים פשוטים. אנו נלמד עליו כיצד להציג איך מנוע רנדום עובד, למרות שבעולם האמיתי סביר להניח שתעבדו עם ריאקט, אנגולר או שוע.

`ej` לא בא ככלי מיחל עם אקספרס ויש להתקינו באמצעות:

```
npm install ejs
```

אחרי שהתקינו את המנווע הזה, יש צורך להוראות לאפליקציית האקספרס שלנו להריץ אותו. עושים את זה באמצעות הוראה מפורשת בקוד שמצויבים ב-`js`.app, הקובץ המרכזי של האפליקציה שלנו, באופן זהה:

```
app.set('view engine', 'ejs');
```

מהנקודה הזו אנו יכולים להשתמש במנוע התבניות. ניצור תיקייה מהנתיב הראשי שנקראת `views`. השם הוא ברירת מחדל והוא מכיל את כל התבניות. הבה ניצור תבנית בשם `index.ejs` ונכנס בה HTML רגיל וסטנדרטי:

```
<html>
  <h1>Hello world from template!</h1>
</html>
```

עכשיו אנו יכולים להשתמש בתבנית הזו בדיק כמו בקובץ סטטי. הדרך להשתמש בה היא פשוט להשתמש במתודת `render` שנמצאת ב-`result` ולהכנס את שם התבנית, במקרה שלנו `index`.index. אם למשל אני רוצה לקרוא למבנה הזה מהנתיב המרכזי והראשי, ב-`js`.app אני אكتب (עוד לפני הרוטינג):

```
app.get('/', (req, res) => {
  res.render('index');
```

ואם אני אכנס אל:

<http://localhost:3000/>

אני אוכל לראות את הדף. בדיק כמו דף סטטי! זה נחמד, אבל היתרון בתבניות הוא שאני יכול לדוחוף פנימה משתנים, אלו משתנים שבאל – בתור הארגומנט השני של מתודת `render`. למשל, בואו נכנס `subtitle`. אני אוצר אובייקט עם `subtitle` ופושט עבירות אותו כארגומנט השני, ממש ככה:

```
app.get('/', (req, res) => {
  res.render('index', {subtitle: 'This is subtitle'});
});
```

מעכשיו בתבנית שלי יש משתנה בשם subtitle. הוא יכול להיות כל דבר. במקרה זה הוא יהיה מחרוזת טקסט, אבל הוא יכול להיות הכל. על מנת להדפיס את המשתנה זהה, אני צריך להשתמש בתהביר מיוחד – חיצים עם אחוז. זה בעצם השימוש בתבנית:

```
<html>
  <h1>Hello world from template!</h1>
  <h2><%=subtitle%></h2>

</html>
```

אם אני אכו א'ל:

<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 משזה כזה:

```
<html>
  <h1>Hello <%=name%></h1>
</html>
```

אני גם יכול להכניס מערכיים — מה שיש בסעיף 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});
});
```

והתבנית תיראה כך:

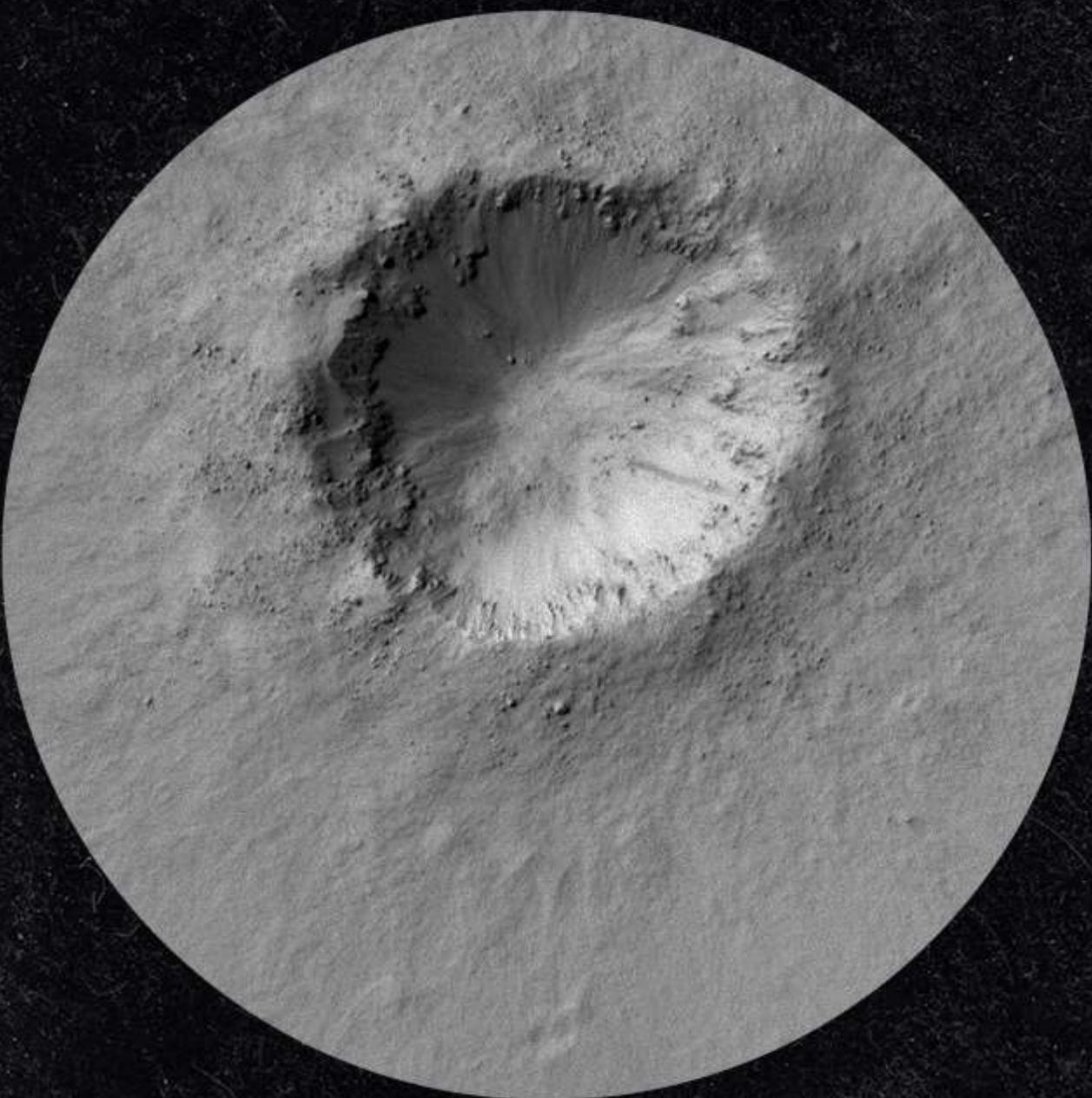
```
<html>
  <h1>Hello <%=params.name%></h1>
</html>
```

בתבניות סעיף יש המונ אפשרויות נוספות, כמו תות-תבניות, לולאות ותוספות נוספות, אבל הבסיס מספק ברוב האפליקציות. ברוב המקרים אנו משתמשים בספרייה צד שלישי כמו ריאקט על מנת לבנות את התבנית, בעוד הצד השרת אנו משתמש באקספרס כדי לבנות את ה-**API** – כלומר את הממשק **שה-AJAX** פונה אליו ואת התבנית הבסיסית שטוענת את ספריית הריאקט.

אפליקציה אמיתית באקספרס תכיל ראותרים שיגדרו את הנתיבים, או endpoint בرمאה מקצועית שקיימות **AJAX** יכולות פנומ אליה. באמצעות **the-Middlewares** יהיה בקרה אבטחתית, זיהוי ואוטנטיקציה. במודולים עצמים יהיה המידע שיועבר לראוטינג. מי שיקבל את המידע זה צד הלקוח, שמציג אותו באמצעות ריאקט, אנגולר או שיע, וביהם ספר זה אינו עוק.

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 מסוג MySQL הוא אחד מסדי הנתונים הנפוצים בעולם. ספר זה אינו מלמד מסדי נתונים והפרק זהה מנחית הבנה בשאלות בסיסיות ב-MySQL. במידה שאין לכם ידע בנושא, אפשר לדלג על הפרק זהה 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 שעובד. אני שוב יוצא מנקודת הנחה שאתם מכירים מספיק MySQL כדי להרים שרת זהה בעצמכם. אפשר להרים MySQL על כל מערכת הפעלה: חלונות, מkn או לינוקס. התקינה היא קלה והריצה עוד יותר. אם יש לכם מסד נתונים שרצ במחשב, סביר להניח שהוא נמצא ב-localhost בפורט 3306, אבל מסד נתונים יכול להיות בכל מקום, כמובן.

בפרק זהה אני משתמש בדוגמה במסד נתונים שנמצא על המחשב שלי, localhost ששמו הוא `test` שמשתכנים אליו עם שם משתמש `root` וסיסמה `123456`. הטבלה היא `clients` ויש לה שלושה שדות: `id` (המפתח הראשי), `name` ו-`city`. מובן שבאתר אמת לעולם אל תשתמשו בשם משתמש `root` ובסיסמה פשוטה. ונוסף על כן, שם המשתמש והסיסמה צריכים להיות במשתני סביבה בפרק הקודם.

## חיבור ראשוני

אנו מתחברים אל MySQL באמצעות מתודה אסינכרונית – כלומר היא תבלום את כל הפעולות שיבואו אחריה. זו אחת הסיבות שחשוב לבצע את החיבור פעם אחת ולשמור את הרפרנס זהה במקום שקל להגעה אליו. החיבור הוא פשוט ובו אנו צריכים לספק את השרת שעליו מסד הנתונים, שם מסד הנתונים והסימנה ושם המשמש לחיבור. כך זה נראה:

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

const connection = mysql.createConnection({
  host      : 'localhost',
  user      : 'root',
  password : '123456',
  database : 'test'
});

connection.connect();
```

מהרגע זהה יש לנו חיבור. אם נריץ את הקוד זהה (למשל נבדיק אותו ב-ז'קעפ ונריץ את הקוד באמצעות `node ./app`) נראה שהתחילה נותר פתוח. הרצת הקוד לא מסתייםת כיון שכל עוד החיבור פתוח, הקוד רץ.

חשוב מאד לציין שהקוד המובא כאן הוא רק לצורך הדוגמה. לעולם לא לנו להכניס סיסמאות בקוד המקורי של האפליקציה שלנו. זוכרים את משתני הסביבה שעליים למדנו בפרק מוקדם יותר? זהו מקום מציין להשתמש בהם. למשל, ליצור משתני סביבה בנוסח זהה:

```
HOST=localhost
USER=root
PASSWORD=123456
DATABASE=test
```

ולקרוא להם בקלות באופן 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 יסגר והקוד יפסיק לזרז.

בדרכן כל אנו לא יוזמים חיבור ישיר אלא מבצעים pooling, כלומר יוצרים כמה חיבורים, שומרים אותם בצד ומשתמשים בהם לפי הצורך. מודול MySQL מאפשר לנו לעשות את זה בקלות ובדרכן דומה מאוד לחבר ישיר באמצעות מتدות `pool.createPool`. מוגדרת זו זהה אחת לאחת למתודת `createConnection` אלא שבניגוד אליה, אנו צריכים לומר כמה חיבורים צריכים ליצור בצד. כך זה נראה:

```
const pool = mysql.createPool({
  connectionLimit: 10,
  host: 'localhost',
  user: 'root',
  password: '123456',
  database: 'test'
});
```

אנו לא צריכים לבצע `connection`. מהנוקודה זו, אנו יכולים להתחיל לעבוד. אנו כן צריכים לשמור את הקבוע `pool` בצד כי דרכו 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 מتدות `get` (באמצעות `pool` או `connection`) במודול נפרד. מוגדרת זו זהה אחת לאחת למתודת `get`.

הערה חשובה: אם אתם מקבלים את השגיאה זו:

`Client does not support authentication protocol requested by server; consider upgrading MySQL client`

הריצו את השאלה הבאה בשרת MySQL:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password
BY 'password';
כשה-password היא הסיסמה שלכם.
```

## שאילתת בסיסית

אחריו שיש לנו חיבור, אנו יכולים להתחיל לעבוד. נתחיל עם שאלה בסיסית של SELECT. אפשר לבצע שאלות ישרות מ-socket או מ-connect באמצעות query. השאלה זו מקבלת שני ארגומנטים. הארגומנט הראשון הוא השאלה עצמה, הארגומנט השני הוא קולבך שיש בו שלושה ארגומנטים: שגיאה, תוצאה (שגיאה אובייקט) 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 שאלה שהיא אל מתודת query ותמיד נקבל תוצאה או שגיאה. זה עד כדי כך קל. גם INSERT עובד או כל שאלה אחרת. כן, כולל MySQL מודול Node.js של MySQL הוא בסופו של يوم גשר – גשר בין הקוד שלנו למסד הנתונים. כל מה שצריך לעשות הוא להחליט איזו שאלה אנו שואלים.

## המרת הקוד לעבודה עם פרומיסים ולא עם קולבקים

אפשר לראות כמה זה קל. אבל יש עם זה בעיה: הקוד עובד עם קולבקים ולא עם פרומיסים. פתרון הבעיה הזה הוא קל יחסית, כיון שאפשר להמיר כל קולבק לפרקטיוס באופן זהה:

```
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);
});
```

זהוי טכניקה שמתכונתי ג'אווהסקריפט אמורים לשלוט בה. אנו יוצרים פונקציה ושם אנו מחזירים פרומיס. בתוך הפרומיס אנו מבצעים את הקריאה האсинכורונית עם הקולבק ובתוך הקולבק מבצעים `reject` או `resolve`. עכשו אפשר לקרוא לפונקציה שיצרנו ולקבל פרומיס כרגע, או להכניס אותה לתוך `:async/await`

```

    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) => {
                if (error) reject(error);
                resolve(results);
            });
        });
    }

    CreateQuery(pool).then((results) => {
        console.log(results);
    });
  
```

יצירת החיבור

פונקציה שמחזירה פרומיס

הקריאה מוגבלת - עם הקולבק, הקולבק שיבסביבת פרומיס

קריאה רגילה לפונקציה  
שמחזירה פרומיס

אנו חיברים להפעלה  
את החיבור

از בוגע לקולבקים, אין בעיה. פשוט צריך לזכור שבחיכים האמיתיים ולא בספר או בדוגמאות, נעתוף תמיד את הקוד של הקולבקים בקוד שמחזיר פרומיס.

## Prepared Statement

בעה גדולה יותר עם כתיבת Queries ישירות באופן זהה היא שمدובר 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 שמויחה 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 נשמע מוסף אבל המימוש של זה הוא פשוט. כתובים את ה-Query אבל במקומם הנתונים מציבים סימן שאלה [?] ומעבירים 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-id. אנו מחליפים אותו בסימן שאלה ומעבירים 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 באמת פשוט – בשאלתה שלכם אסור شيء נתוניים שmaguiim מבচוץ. אנו מציבים סימן שאלה בכל נתון זהה ומעבירים את הנתונים מבচוץ במערך לפי הסדר של סימני השאלה. זה נעשה כי ברוב המקרים הנתונים לשאלות מגיעים מקלט של המשתמש ואנו ממש לא רוצים לקבל מה 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 MySQL עם Node.js באמצעות מודול ייחודי, ומודולים כאלה קיימים לכל מודול נתונים שיש. זו הגדולה והכוה של Node.js.

פרק 19

# עליה לפורודהשו



# עליה לפרויקט

למדנו איך מרים שרת HTTP, ללא אקספרס ועם אקספרס, איך מרים שרת שתמך בסוקט, ונשאלת השאלה – איך מעלים הכל לרשת? אם צרתי אתר או API או שירות כלשהו מבוסס js.Node, איך אני מעלת אותו לרשת? איך אני מאפשר לאנשים אחרים להשתמש בו? איך אני עולת לסייע להפעלתה אמיטית? זה נקרא **עליה לפרויקט** – סביבת ייצור אמיטית.

יש כמה דרכים להעלות קוד של js.Node. הראונונה היא לשכור שירות אמיטי, **Private Server** או **mcnode** וירטואלית על שירות – **Virtual Private Server**, ולהפעיל את הקוד ממש בדיקון כמו שהפעלנו אותו מהמחשב בזמן הלימוד. צריך לזכור שבסוףו של דבר השירות 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 השירותים מבוסס -**js.Node** באופן דומה למחשב שלנו.

דרך שנייה היא באמצעות שירותי ענן. יש לא מעט שירותי ענן בעולם: אמזון, איז'ור והרוקו הם המובילים. שירות הענן דואג בעצמו לכך – הוא יפזר עוד שירותים וירטואליים אם הוא חש בעומס על השירות שלנו, למשל. יהיה בו גיבוי והפרישה תהיה אוטומטית. רוב החברות עובדות עם שירותי ענן וכל שירות ענן זהה יש מדריך משלו המסביר איך לפרש עליו קוד js.Node.

בפרק זהה נראה את שתי השיטות.

## עליה לפרויקט עם שרת

אם אתם מתכנים PHP או מתקנים בשפות אחרות, העלייה לפרויקט היא פשוטה – להעלות קבצים אל השירות, וה-Apache או ה-**X****Nginx** יודעים לטפל בזה. אבל ב-**js.Node** אנו לא יכולים לעשות את זה, אנחנו בעצם ממשמים את השירות. אנחנו צריכים לדאוג שהוא כל הזמן יהיה באוויר.

אם תרגלתם כמו שצרי, ראייתם שאם אתם לא מרכיבים את תהיליך **js.Node** שאחראי להרים את השירות שלכם – השירות שלכם לא יעבד. אם יש תקלה – התהיליך יקרוס ותצטרכו להרים אותו מחדש, או אם אתם עושים ריסט למחשב, תצטרכושוב להריץ אותו מחדש. זה בסדר כאשר לומדים ומתרגלים, אבל פחות כאשר מדובר באתר שמשרת לקוחות. אנחנו צריכים לדאוג לכך שהטהיליך יירוץ כל הזמן. זהו האתגר האמיטי כאשר אנו פורשים קוד של **js.Node** בשרת משלנו. גם כאשר השירות עבר ריסטרט וגם כאשר יש לנו תקלה מסוימת – התהיליך תמיד צריך להיות למעלה. תחשבו על גמד קטן שבכל פעם מריץ **js.app** כאשר האפליקציה נופלת.

לגמד הקטן הזה יש שם: **pm2**. נחשו מה? זה מודול של **js.Node** שדואג שהקוד שלנו יהיה כל הזמן במעלה.

מתקינים אותו על השירות באמצעות התקינה גלובלית של מודול.

```
npm i pm2 -g
```

אחרי ההתקינה, נקליד:

`pm2 start app.js`

האפליקציה תתרכז לאותו. אם תעשו ריסטרט למכונה או תסגורו את הטרמינל, השרת עדין יהיה באוויר. אפשר להריץ כמה וכמה שירותים וולראות אותם באמצעות:

`pm2 list`

2mk גם שימושי מאוד לסביבת פיתוח, ואפשר להתקין אותו על המחשב בכל מערכת הפעלה. אם תקלידו:

`pm2 start app.js -watch`

ותשנו את הקוד שלכם, האפליקציה תיתען מחדש.  
אם אתם רוצים לעצור סופית את האפליקציה פשוט תקלידו:

`pm2 stop app.js`

הדוקומנטציה העשירה של 2mk מסבירה היטב איך לעשות את כל הפעולות הללו. אם עברתם על כל פרקי הספר, לא צריכה להיות לכם בעיה להסתדר עם הדוקומנטציה זו. אפשר להגיד שם כמה סיבות לכך אפליקציה, להגדיר משתני סביבה שונים ועוד דברים מעניינים. אם אתם רוצים להרים את אפליקציית Node.js שלכם על שרת שלכם, 2mk זו הדרך.

## עליה לפרויקטן בענן

לכל סביבת ענן יש הדריכים שלה להעלות אפליקציית Node.js – בהתאם לצרכים. בחברות גדולות ואפיו קטנות יש אדם שתפקידו הוא להעביר את הקוד לסביבת הענן. התפקיד של האדם הזה נקרא DevOps וזהו הלוחם של שתי מיללים: Developer ו-Operations. תפקידו לקשר בין הפיתוח לפרויקטן והוא גם מופקד על סביבת הענן והתחזוקה שלה. לא פשוט לתחזק סביבת ענן ונדרש לכך ידע עמוק בכל פלטפורמה ופלטפורמה.

למרות זאת, יש סביבות ענן יידידותיות למפתחים בודדים. נלמד עליה בפרק זהה.

על מנת לעבוד עם Heroku, יש ליצור שם חשבון ולהתקין על המחשב שלכם את Heroku toolbelt. מדובר ב-CLI פשוט שמאפשר לכם להקים ליהיד בטרמינל את הפקודה `heroku` ובאמצעותה לעשות פעולות שונות. מיד לאחר ההתקינה נקליד `heroku login`. נקליד את הסיסמה וניצור מפתח התחברות. בעצם זה ללחוץ על `Yes`, `Yes`, `Yes` ולספק פרטיים בכל פעם שנדרש מאייתנו. אחרי שעשינו את זה. ניצור בתיקיית האב של הפרויקט שלנו קובץ שנקרא `Procfile` – כן, ללא סימת קובץ. הקובץ הזה מכיל את כל ההוראות לפרישה של הפרויקט שלנו. במקרה של כל הדוגמאות בספר זה יהיה משוחה בסגנון: `node app.js`. נפתח את קובץ `Procfile` ונכתבו בפנים:

`web: node app.js`

נשמר את הקובץ.  
עכשו להעלאת. כדי להעלות את הפרויקט שלנו ל-Heroku, אנו צריכים לעבוד עם מנהל גרסאות מסווג גיט. גיט וניהול גרסאות אינם נלמדים בספר זה אבל זה כל' חשוב מאוד למפתחים.  
נקlid Heroku create כשאנו בתקיה הראשית של הפרויקט שלנו. ואחרי כן:

```
git push Heroku master
```

וזאת בהנחה שאתה רוצה לבצע deployment ל-master. הקפידו שכל הקבצים שלכם אכן יהיו בגרסה זו. הפעילו את האפליקציה באמצעות:

```
heroku ps:scale web=1
```

והיכנו אליה באמצעות `open heroku`. שימו לב שכבר תועברו לנתיב `sh-heroku` יצירה עבורכם. שימו לב שגם יצרתם שרת ואתם רוצים לגרום לו לעבוד, הקפידו לא להכניס פורטים מסוימים בשרת שלכם. במקומם:

```
app.listen(3000);
```

או משהו בסגנון, הקלידו:

```
const port = process.env.PORT || 3000;
app.listen(port);
```

אחרת השרת שלכם לא יעבוד ב-heroku (או בכלל בסביבת ענן אחרת). כאמור, המדריך זהה 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 האתר אמיתי – בטח בשביל אתר מרובה משתמשים – תצטרכו לחקור יותר את הדוקומנטציה של Heroku ואולי גם להתייעץ עם איש DevOps.

פרק 20

# סינום



## ויכוח

אם אתם יודעים ג'אווהסקרייפט היטב, ועברתם על כל פרקי הספר זהה ותרגלו את כל התרגילים – אתם כבר אמורים להבין היטב איך js.Node עובדת ואייר התוכניות שלה רצות. עכשו, כל מה שצריך הוא ניסיון. לא ידע תיאורטי. שפת תכונות ופלטפורמות שונות לומדים תמיד דרך הידיים – ניסיון מעשי. אפשר לצבור את הניסיון המעשית הזה בקלות גם בלי עבודה בחברה.

השאלה ש תמיד שואלים היא – איך? איך צוברים ניסיון? למידה תיאורטיבית זה טוב ויפה, אבל יש צורך גם בניסיון פרקטטי. מהניסיון של, הקהילה בארץ מודח חמה ותומכת ואפשר לפגוש אותה בכם וכמה מקומות. האנשים שם יسمחו לסייע לכם להשתלב בתחום בהתאם ליכולות ולשאיפות שלכם.

## ミיטאפים

הדרך הטובה ביותר לפגוש אנשים שעובדים עם js.Node ביוםיום היא להגיע לミיטאפים. יש בישראל לא מעט מיטאפים למתכנתים js.Node, שרובם מרכזים באתר [meetup.com](https://www.meetup.com). הגיעו לミיטאפים, שמעו הרצאות מעניינות ודברו עם אנשים שם. הם יכולים לסייע לכם בנוגע להתקדמות הלאה.

נוסף על מיטאפים יש האקטונים. זו הזרמת הנדרת להצטרף לצוות של מתכנתים מנוסים יותר, ללימוד מהם ולבוד לצדם בפרויקטים אמיתיים.

## קבוצות דיוון

בפייסבוק יש כמה וכמה קבוצות בעברית המוקדשות לג'אווהסקרייפט ול-javascript. הקהילה שם היא חמה ומקצועית. הטרפו לקבוצות, קראו את הדינונים והשתתפו בהם. זה יועיל מאוד לידע המקצועי שלכם ויכoon אתכם לעבודות ולפרויקטים.

## גיטהאב

בגיטהאב יש לא מעט פרויקטים של קוד פתוח מבוסס js.Node. בחלק מהם כבר השתמשתם בספר זהה. נסו לראות אילו פרויקטים מבוסס js.Node יש שם ונסו לתרום להם או לבחון את הקוד שלהם.

## התנדבות בעמותות ובמיזמים

לא מעט עמותות, מיזמים כמו כניסה פתוחה, למשל, וארגוני מחפשים מתכנתים בתחילת דרכם שיעוכלו 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 מאד לתרגם, לתרגם. שפה ופלטפורמה הן בדיקן כמו שפה בעולם האמיתי. אם לא תשתמשו ותתרגםו – תשכחו. ב-`js`-`Node` אפשר לעשות המון דברים. אפשר ליצור תוכנות לשרתים, אבל יש גם פלטפורמות המאפשרות להשתמש ב-`js`-`Node` על מנת ליצור אפליקציות לנידים, תוכנות למחשב ביתי ואפילו תוכנות ל-`IoT` – האינטרנט של הדברים. בחרו את אחד התחומים שימושכם ונסו ליצור פרויקט משלכם. יהיו קשיים, יהיו בעיות – לא הכל אפשר ללמוד מספר או מקורס, טובים ככל שיהיו. נתקלתם בעיה? חפשו את השגיאה בגוגל ונסו לפתור אותה עצמאית, חפשו באתר StackOverflow או נסו להיעזר באחת מקובוצות המתכנתים בפייסבוק. מה שחשוב הוא לא להתיאש. `Node.js` באמת נמצאת בכל מקום, קל להפעיל אותה והיא יופי של פלטפורמה לכל מתכנת ג'אווהסקייפט. בהצלחה!

# נספח: בדיקות אוטומטיות ב-Node.js

מאט דניאל כץ חברת Elementor

## מה זה בדיקות אוטומטיות?

תתארו לעצמכם מגדל ג'נגה גבוהה, ואתם יושבים עליו בזמן שבאופן אקראי נשלפים ממנה לבנים. מרגישים את הפקד? אז זהה השגירה בארץ ה-Node.js. הקוד שלכם יושב מעל מגדל זהה הבני ממודולים משתמשים כל הזמן, ובו בזמן אתם וחברי הצוות שלכם גם מכניםים קוד חדש שיכל להחיל באגים חדשים או רgresions בדברים שעבדו קודם.

כל אפליקציה Node.js תלויה לכל הפחות בגרסה של Node.js עליה היא רצה, רוב האפליקציות משתמשות גם בספריות וקחו הרבה, וכך הן תלויות גם בגרסה של וקוח עצמו. וכל אחת מהחבריות בהן האפליקציה משתמשת עלולה להיות תלויה בחבילות נוספות עצמה. רק כדי לסביר את האוזן, אפליקציית express היכי מינימלית תלויה בכ-400 חבילות תוכנה שונות.

אז איך מייצבים את המגדל? איך בכל זאת בונים כלים ושירותים יכיבים דוגמת Netflix ו-ebay, מהשירותים היuibים והסקילבילים בשוק.

כדי לענות על השאלה הזאת, בואו נסתכל על תוכנה לא קוד, אלא zusätzlich של חזדים אשר ממומשים באמצעות קוד. כמובן, כמודול חושף פונקציונליות כלשהי, הוא בעצם יוצר חזזה שהפונקציונליות הזאת לא תשתנה בעתיד.

לדוגמה, נניח שהקייםים 2 מודולים של מחשבון: `pythagoras` ו- `euclid`:

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 לכולנו שני המודולים אינם זהים, שכן בעוד שמודול pythagoras יכול רק לחסוך מספרים, המודול euclid יכול הן לחסוך והן לחבר. בעצם, נניח שמיימשתי ב-`pythagoras` גם יכול חיבור, ואז באמצעותה מימושי מחדש גם את יכול החישור כך:

### pythagoras.js

```
function add(a, b) {
  return a + b;
}

function subtract(a, b) {
  return add(a, -b);
}

module.exports = { add, subtract }
```

האם הרגע הפכתי את `pythagoras` ל-`euclid`? ברור שלא. למרות שני המודוליםicut בעלי יכולות זהות, pythagoras לא נהפרק ל-`euclid` בגלל שיש הבדל בקלט בין מה ש-`pythagoras` לבלין מה ש-`euclid` מקבל; בעוד ש-`pythagoras` מקבל מצפה ל-`euclid` אובייקט עם 2 ארגומנטים (`a` ו-`b`), המודול `euclid` מקבל אובייקט עם 1x ו-`2x`); - יש להם חזזים שונים!

שאלה יותר טובה היא: האם המודול `pythagoras` עדין נשאר `pythagoras` או שעתכשוו הוא כבר משהו שלישי - חדש?

אם נגידיר את תוכנה zusätzlich של חזזים ממומשים, נוכל להוכיח שהיות והגרסה החדשה של `pythagoras` עדין מימושת את החזזה שהוגדר בגרסה המקורית שלו (`(b) subtract(a)`) - זה עדין אותו המודול.

אבל רגע! האם אכן הגרסה החדשה של `pythagoras` עדין מימושת את החזזה הקודם? כדי שנוכל לקבע בודאות שהגרסה החדשה מכבדת את החזזה הישן, אנחנו צריכים דרך להוכיח שעבור כל קלט, הגרסה החדשה תחזיר את אותו הפלט כמו הגרסה הישנה.

בואו נבדוק כמה דוגמאות של סוג 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 ערכיהם מסוג `number` או מסוג `string` ומחזירה תוצאה נכונה - זהו החוצה של המודול.

כעת, נזין את אותו הקלט לגרסת החדשה ונווידא שהחוצה עדין מתקין:

### 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')); // Outputs: "3-2"

// 3. it should return the result of addition of two numbers
console.log(add(1, 2)); // Outputs: 3
```

רגע, מה? למה החישוב  $2 - 3$  מחזיר  $"2-3"$ ? ציפינו לקבל 1, החוצה נשבר!

בגרסת הראשתונה המתודה `subtract` תמכה בארגומנטים מהסוגים `number` ו-`string`. ובגרסת השניה זיהינו רגסティיה - חוצה שהיה מכובד בגרסת הקודמת, הופר בגרסת החדשה. מה שגרם לרגסティיה הוא שמיישנו מחדש את `subtract` באמצעות `add` ולמעשה, שינוינו את האופרטור בו השתמשנו, מאופרטור חיסור (-) שאוטומטית מנסה להמיר מחרוזות למספרים, לאופרטור חיבור (+) שבמקרה זה מבצע שרשור מחרוזות, וכן איבדנו את התמייה במחרוזות בפעולת החיסור.

למעשה, הבדיקות שהרגע הרצינו, היו בדיקות ייחודית, ובעזרתן גילינו שהמימוש החדש לא עונה יותר על החוצה שהגדרנו, או במליל'ם אחרות - מצאנו באג!

## בדיקות אוטומטיות

כל מה שחרס לנו על מנת להבטיח את המשך קיום החוצה של המודול הוא לוודא שאנו מרכיבים את סט הבדיקות הזה שוב אחרי כל עדכון של המודול. וכך ייש לנו עדין שני אתגרים:

1. אמנים כתבנו בקוד את הבדיקות, אך הווידוא של תוצאות הבדיקה הוא עדין ידני.
2. אין לנו דרך נוחה להריץ את סט הבדיקות.

על מנת להתמודד על האתגר הראשון נגדיר את התוצאה הרצויה, ובמידה ומתיקבלת תוצאה שונה מරוק 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 '${expected}' but actual is '${actual}'` +
  );
}

// ...

```

וכדי שנוכל להריץ את הבדיקות בקלות, נוסיף סקורייפט `test` בקובץ `package.json` כר' :

### package.json

```
{
  "name": "pythagoras",
  "version": "2.0.0",
  // ...
  "scripts": {
    "test": "node pythagoras.err.js"
  },
  // ...
}
```

עכשו נוכל הריץ `npm run test` וולדעת 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 כעת אתם חושבים לעצמכם, רגע... שבסבב לבודק 3 שורות קוד, הרגע כתבתי 8 שורות, אתם לgambar צודקים. ישנו כלים שמקצרים ומייעלים בהרבה את כתיבת והרצת הבדיקות האוטומטיות. הם מתחולקים עיקר לשתי קבוצות:

1. מרייצ'י בדיקות - Test Runners דוגמת Mocha ו-Jest, אשר אחראים על הרצת הבדיקות והפקת דוחות תוצאה.

2. פרימוורק בדיקות - Assertion libraries דוגמת Chai ו-assert שמאפשרים לבטא בקלות את תנאי הבדיקה, ומיצרים תיאור שגיאה אינפורטטיבי במקרה של כשלון.

## Mocha

על מנת שכתיבת והרצה של בדיקות תהיה נוחה ועקבית, עם הזמן נוצרו "MRIICI בדיקות" שונים, המוכרים ביניהם הם Mocha, Jest ו-Jasmine. אנחנו נתמקד ב-Mocha, אבל אין הבדלים משמעותיים באופן כתיבת הבדיקות למספריות האחרות.

התפקיד של MRIICI הבדיקות הוא לאפשר קבצי בדיקות בפרויקט, להריץ אותם, להחליט אילו בדיקות עברו ואילו נכשלו, ולדוח את התוצאות במגוון דרכים - לדוגמה הצגה של התוצאות בקונסול או שמיירתם לקובץ.

לצורך הוספה של Mocha לפרויקט, נרץ בשורת הפקודה:

npm install --save-dev mocha

ונשנה את הסקריפט `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 להרצה מכל מקום במחשב ע"י הparameter `-g`:

npm install -g mocha

היחסון בדרכו, הוא - להיות ומודובר בהתקנה גלובלית, כל הפרויקטים על המחשב המודובר יישתמשו באותה הגרסה של הספירה, ולפעמים זו לא התוצאה הרצוייה.

כעת שהוספנו את mocha לפרויקט, בואו נחזיר 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 הטען:

```
$ 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 mocha מודעת לקוד הבדיקה הראשון שכתבנו, רק שעכשיו mocha מאפשרת לנו לארגן את הבדיקות באמצעות מethodות `describe` ו-`it` (עליהן נרחיב בהמשך), ומדפסה אוטומטית את ההפרש בין התוצאה הרצויה לתוצאה בפועל. והחבילת `assert` מאפשרת לנו לבטא בקלות את התנאים להצלחת הבדיקה, בלי הצורך בכתיבת משפטי `if` ארוכים.

עכשו שראינו איך זה נראה מלמעלה,בואו נתייחס לחלקים השונים המרכיבים את הבדיקה יותר לעומק.

## describe

כפי שראינו בדוגמה מלמעלה, בлок `describe` מגדיר נושא שאנו אונחנו הולכים לבדוק, אבל לא מכיל את קוד הבדיקות עצמו. ניתן לקוון כמה בлокי `describe` אחד בתוך השני. לדוגמה, כשאנו חונכו רוצים לכתוב בבדיקות על מетодה `add` של מודול `pythagoras` אנחנו יכולים לבטא את זה כבלוק בשם `'add'` מוקון בתוך בлок 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 אונונימית.

ניתן להשתמש בבלוק `describe` לא רק כדי לקובץ בבדיקות על חלק טכני של הקוד (כמו מודול או מетодה), אלא גם לפיה כל נושא העולה על הדעת. mocha תשתמש בהיררכיה של בלוקי `describe` לארגון התוצאות, דבר שמקל מאוד על הבנת הדוחות.

## it

כל קובץ בבדיקות 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 אונונימית שהיא הבדיקה עצמה.

החוקיות לפיה נקבעת הצלחת הבדיקה פשוטה מאוד - אם הקוד זורק שגיאה, הטסט נחשב ככשל אחרית - הצלחה. לדוגמה, הבדיקה הבאה תחשב להצלחה למורות שהיא ריקה מקוד בכלל שהמוגדרת האוניבריאט 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 בפרימוורק בדיקות דוגמת assert או chai לצורך כתיבת הבדיקה, ואלו מארורי הקלעים יזרקו שגיאה מסוג AssertionError במקרה שבדיקה נכשלה.

בינתיים 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' תעבור, כי הבדיקה  $1+1=2$  נכונה, מה שקרה לומר על  $1+1=42$  ולכן הבדיקה 'false assertion' תכשל.

## מחזור חיים

לפעמים בדיקה תדריש פעולות הכנה לפניה ופעולות ניקוי לאחריה. לדוגמה נניח שאנו בודקים מודול שמבצע שאלות על מסד נתונים. יתכן שאנו נרצה לבדוק את מסד הנתונים במצב ידוע לפני הבדיקה, ולהציגו במצבו ההתחלתי אחריו.

בשביל זה יש ב-mocha את המתודות: `before`, `after`, `beforeEach`, `afterEach`. המתודות האלה נקראות `hooks` - ווים כי הם מקושרים לאיירועים בזמן ריצת הבדיקות.

כל המתודות האלה מקבלות רק ארגומנט אחד - `fn` שזה מתודה אוניברסלית 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 בлокים אחד בתוך השני, כל hook חל על הבלוק שבו הוא מופיע ועל כל הבלוקים שבו.

המתודות `before` ו-`after` רצות לפני תחילת ביצוע הבדיקה הראשונה בבלוק, ואחרי סיום ביצוע הבדיקה האחרונה בבלוק עליו הון חלות - בהתאם.

המתודות `beforeEach` ו-`afterEach` רצות לפני ואחרי ביצוע של כל בדיקה ובדיקה בבלוק עליו הון חלות - בהתאם.

המתודה `beforeEach` רצתה לפני המתודה `beforeEach` והמתודה `afterEach` רצתה אחרי המתודה `beforeEach`.

במקרה שיש `hook`-ים בבלוקים מיקוניים, המתוודות `beforeEach`-ו `before` של בלוק חיצוני יותר רצות לפניו המתוודות הנ"ל של בלוק פנימי, ומתוודות `after`-ו `afterEach` של בלוק חיצוני יותר רצות אחריו המתוודות הנ"ל של בלוק פנימי.

#### הערה:

ניתן לכתוב `hook` גם בגין קובץ הבדיקות, מוחוץ לכל `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:

1. החלק הראשון - **Arrange**. בחלק זהה אנחנו נכין את כל נדרש לצורך ביצוע הבדיקה ובביא את המודול שבבדיקה למצב הרצוי.
2. החלק השני - **Act**. זה החל שבו בפועל קוראים לפונקציה שבבדיקה עם כל הארגומנטים שהכינו בשלב הראשון.
3. 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 AAA** או **Triple 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);
});
```

## פרימוורק בדיקות

כדי שהטסטים יהיו יעילים הם צריכים לוודא שלאחר ביצוע הפעולה הנבדקת, התוצאה או המצב זהה למה שנחננו מצפים לקבל. כמו כן שטכנית אפשר לכתוב את תנאי הבדיקה במבנה כפי שעשינו בתחילת הפרק:

```
if (!<condition>) throw new Error ('<error message>')
```

ישנה דרך נוחה ואלגנטית יותר לוודא תוצאות הבדיקה באמצעות **פרימוורק בדיקות**, במקרה שלנו המודול `assert` של `Node.js`.

המודול `assert` מכיל רשימה של מתודות כשל אחת מהן מיועדת לבדיקת תסרים נפוץ מסוים, ובמקרה של כישלון היא יודעת להפיק הודעה שגיאה מפורטת שיעזרת להבין מה בדיק השתקש. לדוגמה: `strictEqual` משתמש במתודה

```
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.

כמובן שגם דוגמה בסיסית ביותר, `assert` מכיל מתודות נוספות לבדיקות מורכבות בהרבה. להלן כמה מהן:

**assert.ok(value)**  
eo שימושי כדי לבדוק אמינות (truthiness) של ערך כלשהו והטסט יעבור רק אם הערך הוא `truthy` ויכשל אם הוא `falsy`. 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`, שיכשל במקרה שהערך לא זהים. מנגד, `notStrictEqual` יכשל דווקא במקרה שהערךים 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 המרות. וכך הבדיקה הבאה תיכשל:

```
assert.strictEqual(1, '1'); // FAIL
```

ולכן גם השוואה של 2 אובייקטים זהים תיכשל, בגלל שאופרטור `==` משווה אובייקטים לפי המצביע. השגיאה גם תכיל הודעה על כך שההשוואה נכשלה בגלל השוני במבנה ולא בגלל שוני בין האובייקטים עצם:

```
assert.strictEqual({ name: 'foo' }, { name: 'foo' }); // FAIL
// (Values have same structure but are not reference-equal)
```

### **assert.notDeepStrictEqual(actual, expected)**

במקרים בהם נדרש לוודא שהוא דואג את שוויון הערבים בשני אובייקטים יש את המתודה `deepStrictEqual` המבוצעת השוואה של ערכים בין שני אובייקטים בצורה רקורסיבית. ההשוואה בין זוג של ערכים שאינם אובייקט או מערך עדין מתבצעת באמצעות האופרטור `==`. הפעם עם השוואה של 2 אובייקטים זהים תעבירו:

```
assert.deepStrictEqual({ name: 'foo' }, { name: 'foo' }); // PASS
```

במקרה של אובייקטים שונים השגיאה תכלול פרטים על ההבדל בין האובייקטים:

```
assert.deepStrictEqual({ name: 'foo' }, { name: 'bar' }); // FAIL
```

+ actual - expected

```
{
+ name: 'foo'
- name: 'bar'
```

ה

### notDeepStrictEqual

 עובד בצורה זהה רק שנכשל דואג במקרה שהאובייקטים זהים.

### **assert.throws(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', () => {
  assert.throws(() => greet(null)); // PASS
});
```

זה בדיק מה שהמתודה `throws` עשו. היא מקבלת כารוגמנט פונקציה שאמורה לזרוק שגיאה, והבדיקה עוברת במקרה שאכן נזרקת שגיאה.

## ספריות נוספת

כמו בכל נושא אחר ב-`js`, גם בתחום פריימוורק בדיקות יש מבחן של אפשרויות שונות. נראה הראויו ביוטר לציין היא ספריית `chai`. ב-`chai` יש מבחן עשיר יותר של סוג בדיקות כמו כן הספרייה תומכת בשלוש סוגניות של כתיבת קוד, יש עבורה עולם שלם 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 שתלוות 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, ולכן צריכה גם קובץ פיזי כדי לעבוד. איך נכתב בדיקה במקרה?

התשובה היא mocking, או חיקוי. ספרית mock מאפשר ליצור חיקויים למתודות אמיתיות לצורך בדיקות. באקויסיטם של Node.js הספרייה המוכרת ביותר בתחום היא `sinon`.

נתקן את `sinon` משותה 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 "לדרס" את המתודה `readFile` של `fs` בעזרת מימוש דמה שלנו שתמיד מחזיר את הערך `'2line1\nline2'`.

המתודה `stub` מקבלת 2 ארגומנטים, הראשון הוא האובייקט המכיל את המתודה אותה אנחנו רוצים להחליף בדמה (`stub`), והารוגומנט השני מכיל את שם המתודה. המתודה `stub` מחזירה אובייקט `stub` שלו קיימת (בין היתר) המתודה `resolves` שבבזרתנה ניתן להגדיר מה `stub` יחזיר כשופעל. שימו לב, אנחנו קוראים ל-`resolves` ולא `returns` כי המתודה `readFile` אמורה להחזיר `Promise`.

בהמשך אנחנו מרים את הפונקציה `getLineCount` עם `null` במקום נתיב הקובץ, ובודקים שההתוצאה היא אן 2 שורות, בדיק כפוי שטיינו.

בראש הבדיקה יוצרים `sandbox` (עליו נרחב בהמשך) וב-`each` אנחנו מאמסים את ה-`sandbox` כדי שהבדיקות מבדיקה אחת לא שפיעו על הבדיקות הבאות אחריה. קרייה `restore` על `sandbox` מבטלת את כל ה"דרישות" שעשינו באותו ה-`sandbox` ע"י `mock` או `stub`.

עכשו נניח שאנו רוצה לוודא שהמתודה `getLineCount` אכן משתמש בקידוד `utf-8` בזמן קריית הקובץ `string`. כאן באה לעזרתינו היכולת השנייה של `stub`, והיא 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` שיצרנו במשתנה `stub` על מנת שנוכל לתחקר אותו בסוף הבדיקה ע"י `calledOnceWithExactly` שמדוודא שה-`stub` אכן נקרא בדיק פעם אחת, עם בדיק הפרמטרים `null` ו-`'utf-8'`.

המתודה `calledOnceWithExactly` זמינה עקב הפעלה של `stub` כל `stub` הוא גם `spy`. כתת נפרט קצת יותר על היכולות השונות של `spies`, ועל השימושים שלהם.

האפשרות "לרגל" אחרי קריאות למתודות. בעזרה ע"י נוכל לדעת כמה פעמים נקרה מתודה מסוימת, ואילו ערכים הועברו לה כארגומנט.

ע"י מחזיר פונקציה שאפשר לקרוא לה כמו כל פונקציה אחרת, אך בנוסף לה, יש לה מבחן מתודות שמאפשרת לחשאל 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 מתודת "דמיה" בעלת התנהלות מוגדרת, ולקבוע מה תחזר מתודת הדמיה עם אחת מהfonקציות `rejects`, `returns`, `resolves`, `throws`. בנוסף, כל `fake` גם כן `spy`.

דוגמה:

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 הפעמים 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 לדרhos מתודה בתוך אובייקט, דבר שמאוד שימושי בדרישת של תלויות מודולים 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 מועד יותר לכתיבת ידוא, ולא רק לדרישת אובייקטים. כשורטים או עותפים אובייקט ב-`mock` ניתן להגדיר את הציפיות ממנו באמצעות `expects`, ולאחר מכן לבודד 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 ו-`chonos` דורס את המימוש המקורי של פונקציות ואובייקטים, צריכה להיות דרך להציג את האובייקט במצב המקורי בסיום הבדיקה כדי ששינויים מבדיקה אחת לא ישפיעו על הבדיקות האחרות. מסיבה זאת, מומלץ תמיד להשתמש ב-`chonos` לשירות אלא קודם ליצור `sandbox` (ארגז חול) ע"י קרייה ל-`createSandbox`, ובסיום כל בדיקה (למשל ב-`afterEach`) לקרוא ל-`restore` כדי להציג הכל קדומו.

דוגמה:

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 חזו רצימ לבודק פונה לרשת לקבלת מידע ולביצוע פעולות? כמובן שניתן לדריש מתודות במודול `http` לצורך הבדיקות, הבעה היא שמאוד קשה "לזיף" שרת `http` אמיתי בגל המורכבות של ה프וטוקול.

בשביל זה קיימת ספרייה בשם `nock`. עם `nock` ניתן לתרוף שאלות `http` לפני שהן יוצאות לרשת, ולהציג תשובה שנראית כאילו היא באה משרת אמיתי.

נתקן את `nock`:  
`npm install --save-dev nock`

עכשו, נניח שיש לנו מתודה שמחזירה את מספר הכוכבים שיש לリפו 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 רצימ לבדוק 2 תסיטיטים; בהינתן sh-github מגיב עם קוד 200, המתודה `getElementorRepoStarCount` תחזיר את הערך של `stargazers_count` שהוחזר, ובמידה שמתקיים סטטוס שגיאה - המתודה 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 `http` בדוגמה השתמשנו בספריה `node-fetch`. חשוב לציין ש-`nock` יעבוד עם כל ספרייה `http` אחרת, כי בפועל הוא מתלבש על מודול `http` הסטנדרטי של `Node.js`.

כמו ב-`nocks`, גם ב-`nock` חשוב להזכיר את המצב לקדמונו בסיום כל בדיקה ע"י `All``nock.clearAll`. אך בኒגוד ל-`nocks`, ב-`nock` כל זיוף מחייב רק לкриאה אחת. לאחר קריאה אחת לכתובות שעלה עשוינו `nock`, הזיוף נטבל והкриאה הבאה תבצע לכתובות הרשות האמיתית. את הקריאה `All``nock.clear` כדאי לעשות במקרה שבדיקה תיכשף לפני שהיא תספיק לבצע את השאלתא, ורק 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 לחלוקת, ובודקים כל "יחידה" צאת בבידוד משאר התוכנה כדי לוודא שהיא מתפקדת כמצופה. הערך שבדיקות ייחודה נוטנות הוא כפול: מעבר לעצם העובדה שבדיקות ייחודה מאפשרות שכלי ייחודה ייחידה פועלת כמו שצריך, הם גם גורמות לנו לתקן ולכתוב את הקוד בצורה יותר מודולרית כדי שנוכל לכתוב בדיקה לכל חלק בלי תלות בחלקים האחרים.

למעשה קיים סגנון פיתוח שנקרא Test Driven Development (פיתוח מכון טסטיים) או בקצרה TDD שבו כותבים את הבדיקה מראש ולאחר מכן כתבים את המימוש על מנת לගרום בדיקה לעבור. TDD מחייב את המפתח לכתוב את הקוד בצורה מאוד מודולרית עם תלויות מוגדרות היטב. חשוב לציין שגם רק דרך אחת, ולא חיבורים לעשות TDD על מנת לכתוב בדיקות ייחודה אינטואיטיביות.

עם זאת, לבדיקות ייחודית יש כמובן גם חסרונות. הן אמנים מודדות תקינות של כל יחידה ייחודית, אך לא מבטיחות דבר לגבי התוכנה בשילומתה, בעיקר ככל שהמורכבות של התוכנה גדלה. בנוסף, זה לא טריויאלי למצוא איזון טוב בין כמות ואיזורי הטסטיים לבין איות הcisio וצימוד בין הטסטיים ליחידות אותן הם בודקים.

בכתיבת בדיקות ייחודית חשוב מאוד לוודא שבודקים את התוצאה של היחידה ולא את הדרך שבה התוצאה הושגה, אחרת קל להגיע למסוב של שינוי בקוד ידרשו גם שינוי בבדיקה.

### בדיקות קומפוננטה

סוג נוסף של בדיקות הוא בדיקות קומפוננטה. בኒגוד לבדיקות ייחודית שבודקים את התוכנה מבפנים החוצה, בבדיקות קומפוננטה אנחנו מתייחסים לכל המודול כאלו יחידה אחת, ומתמקדים בבדיקה החוצה שהמודול חשוב, כמו למשל מבחוץ פנימה.

אחד מהבעיות העיקריות בבדיקות קומפוננטה היא עמידות לשינויים. ככלMORE ששינויים בIMPLEMENTATION התוכנה לא אמרויים "לשבור" טסטיים קיימים, ואם טסט נשרב קרוב לוודאי שגם רגרסיה יש לתקן את המימוש.

בדיקות קומפוננטה מתאימות במיוחד לפיתוח זמש (אג'לי). הן בתסריט שבו אנחנו מפתחים backend לאפליקציית client, ובין אם אנו מפתחים שירות בארכיטקטורת מיקרו-סרביסים, תמיד יש ערך גדול ביכולת להגדיר ממשקים ברורים בין הרכיבים השונים בשלב מוקדם בתהילר, ולאפשר למפתחים של הרכיבים האחרים להסתמך עליהם. כיסוי של אותם המשמקים בבדיקות קומפוננטה מאפשר בהשכמה לא גדולה לוודא שבגרסאות הבאות של השירות אותן המשמקים ימשכו לתפקיד צפוי.

בנוסף, בבדיקות קומפוננטה הן נקודת התwnהה טובה לכתיבת בדיקות, שכן בדרך כלל ניתן לכסות את ההתנהגות הצפוייה של שירות js.js במספר לא גדול של בדיקות.

ספריית *supertest*

ב-*js.js* ניתן לבצע בדיקות קומפוננטה בצורה מאוד אלגנטית באמצעות ספרייה בשם *supertest*. הספרייה *supertest* מאפשרת "לזיף" שאילתת *http* לשירות בדומה ל-*client* או שירות אמיתי אחר שפונה לשירות שבבדיקה, ולודא שתשובות עוננות על הדרישות שהצבנו.

לדוגמה נניח שיש לנו שירות זהה:

### 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 שהעברנו ב-*name*.  
עכשו בואו נכתוב 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 צריכים את האפליקציה שלנו בשלמותה, ובאמצעות המетодה `get` מז"פים שאלותת GET לנútב/`greet` עם ערך `name` בשדה `name` ו-`header` שמצוין שאנוינו ממעוניינים לקבל JSON חזקה, ובעזרת המетодה `expect` אנחנו מודאים שאכן קיבלנו JSON, עם סטוס 200 ושהערך של `hello` הוא אכן `Elementor` כפי שלחנו בשאלותא.

## סוגים נוספים של בדיקות

עד כה התמקדנו בבדיקות וילדיות - בדיקות אשר מוגדאות שהתוכנה פועלת בהתאם לציפיות. אבל גם תוכנה שעובדת להפליא לא שווה הרבה אם קשה להבין ולחזק אותה.

## eslint

עד ראשון לקוד קרייא הוא היצמדות ל"תקן כתיבה" (Coding Standard) כלשהו. בדיקות כאלה נקראות `linting`, ותוכנה שמבצעת את הבדיקה הזאת נקראת `linter`. הלינטර המוכר ביותר ל-`JavaScript` הוא `eslint`. הוא לא ספציפי ל-`Node.js` ומתאים ל-`JavaScript` של כל קוד `JavaScript` באשר הוא.

כדי להתחיל להשתמש ב-`eslint` יש לבצע 2 צעדים:

1. להתקין אותו ע"י `npm install-dev eslint --save-dev`
2. להיות ותקן כתיבה הוא דבר די סובייקטיבי, צריך לספר ל-`eslint` מהו התקן שאנוינו מעדיפים ע"י הרצה 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**

למעשה סיפרתי לאשף שאני כותב קוד JavaScript על Node.js ואני אוהב את התקן הכתיבה של Airbnb. האשף שומר את ההגדרות שלו בקובץ `eslintrc.js`, ובזה מסתיימות ההגדרות.

עכשו כדי לבדוק מה eslint מוצן 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);
```

ונריץ את eslint:

\$ ./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.

ואללה! eslint מזזה את הביעות הבאות:

- השתמשנו ב-`var` כשנדי! אפשר להשתמש ב-`let` או `const`แทน.
- 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 קרייה 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}`);
```

ביצעו את השינויים הבאים:

- שינוי את שם המשתנה ל-`theAnswer` ב-`camelCase`.
  - הכרינו עליו עם `const` כי אנחנו לא מתכוונים לשנות אותו.
  - השתמשנו ב-`Template literal` כדי להזrik את הערך של המשתנה לגוף הודעה.
  - והוספנו הערה `eslint-disable-next-line` מעל הקריאה ל-`log` ב-`console` כדי להסביר לנו ש-`eslint` שאנו משתמשים ב-`console` במודע וזה אינה טעות.

## נרייצ' את eslint

```
$ ./node_modules/.bin/eslint **/*.js
```

מצוין! עכשו הקוד שלנו קרייא יותר לנו, ולחברי הצוות האחרים.

eslint הוא כל' מאד פתוח לקונפיגורציה, שכן יש אינספור סגנונות כתיבה שונים. וכן הוא תומך בהמון אינטגרציות, בעיקר עם וורכית טקסט. לדוגמה אם אתם משתמשים ב-vscode לכתיבת הקוד, קיימ תוסף eslint שמציג את הבעיות של eslint בתוך העורך - תוך כדי הכתיבה, ומאפשר תיקון שגיאות אוטומטי.

## npm audit

כפי שהזכרנו בתחילת הפרק, חבילות ב-`js`-Node נוטות להיות תלויות בהרבה חבילות אחרות כשהן בתורן תלויות בחבילות נוספות. מטבע הדברים, כל הזמן נמצאות חולשות אבטחה חדשות בחבילות השונות, ובעיקיר בחבילות שנמצאות בשימוש נפוץ - מכיוון שהן נחקרו יותר. חולשות אלו בדרך כלל מתוקנות תוך זמן קצר, אבל אין ניתן לדעת אם אחת מהחבילות הרבות שהשירות של依 תלוי בה סובלת חולשת אבטחה?

בשביל זה קיימת ב-מקח פקודת audit.

כשMRIcrit audit מוקם מול חבילת `js.Node`, הוא סורק את כל הפעולות של אותה החבילת ומדועו אם מילויים מוגבלים סובלים מעבית אבטחה.

למשל כשריצים audit rpm מול חבילת המכילה גרסה בעיתית של הספרייה `lodash` קיבל 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 שמצוין בסיום הפלט כל מה שצריך לעשות כדי לפטור 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
```

npm audit -i אוטומטית מעדכן את החבילת הבעיה לגרסה בטוחה.

## אז מה יוצא לי מזה?

בדיקות אוטומטיות פותחות את הדלת לפיתוח גמיש (אג'יל). על מנת לאפשר פיתוח מהיר, יכולת לבצע שינויים בקוד ולהביא אותם ללקוח בזמן קצר מבליל לאבד מאיכות המוצר, חיבת להיות דרך "להקפיא" את המשק החיצוני של הקוד - החזזה שלו מול הליקוחות, בזרה צאת שעדיין יהיה ניתן לבצע שינויים במימוש ולהוסיף יכולות חדשות בלי' פחד מתמיד שהשינוי ישבור משהו שעבד עד כה. בבדיקות אוטומטיות בשילוב עם תהיליך CI/CD (תהליך אוטומטי שMRIIZ בבדיקות לפני פיסת קוד נכנסת למערכת ניהול הגרסאות, וכן לפני שהיא נפרשת בשרת את נשלחת ללקוח) נונצנות בדיקות יכולת הزادת.

כדי למקסם את היתרונות של הבדיקות האוטומטיות בפיתוח גמיש, חשוב לחת את הדעת על האיזורים אותם חשוב לכסות בבדיקות מול איזורים שנמצאים פחות בשימוש או שהם פחות מורכבים, ולהתחליל את כתיבת הבדיקות דזוקא מהמקומות המורכבים והנמצאים בשימוש תדיר. בנוסף, כאשר שלמרות הכל נמצא באג בתוכנה, לאחר תיקונו, כדאי לכתוב טסט שמכסה את הבאג הזה. כך נוכל לוודא שהbag הספציפי הזה כבר לא יחזור על עצמו וACITY המוצר תשתרף.

ניתן לבדוק את ה-coverage (coverage), כלומר את אחוזי הקוד (לפי כמות שורות) שמקסוט ע"י טסטים באמצעות הריצה של mocha עם המפתח coverage--. חוכמת השבט גורסת שה-coverage (coverage) האופטימלי של בדיקות הוא עד 85% מהקוד, משלב זה והילך הוספה של בדיקות חדשות לא מחזירה ערך שווה להשקעה.

באמצעות CISI בדיקתי של קטעים קרייטיים לפעולות המערכת, ניתן להציגו שחרור קיצרים עם אחוזי רגרסיה נמוכים מאוד, מפתחים רגילים ולקוחות מאושרים.

# נספח: שינויים מהמהדורה הקודמת (מהדורה 1.1.0)

תיקון דוגמה בחלק של `sockets`

תיקוני שגיאות קלים

תודה רבה על הרכישה של הספר!

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

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

העתיק זהה נמכר ל:

beninson@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלי - כלומר בתוך דפי הספר נחברים פרטיה הרוכש באופן שקוֹף למשתמש. כדאי מאוד להמנע מהעתיקה של הספר לאלו שלא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למשהו אחר במתנה - העבירו לו את הפרטיהם שלכם באתר ומיחקנו את העותק שנמצא ברשותכם.

תודה וקריאה נעימה!