

תודה רבה על הרכישה של הספר!

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

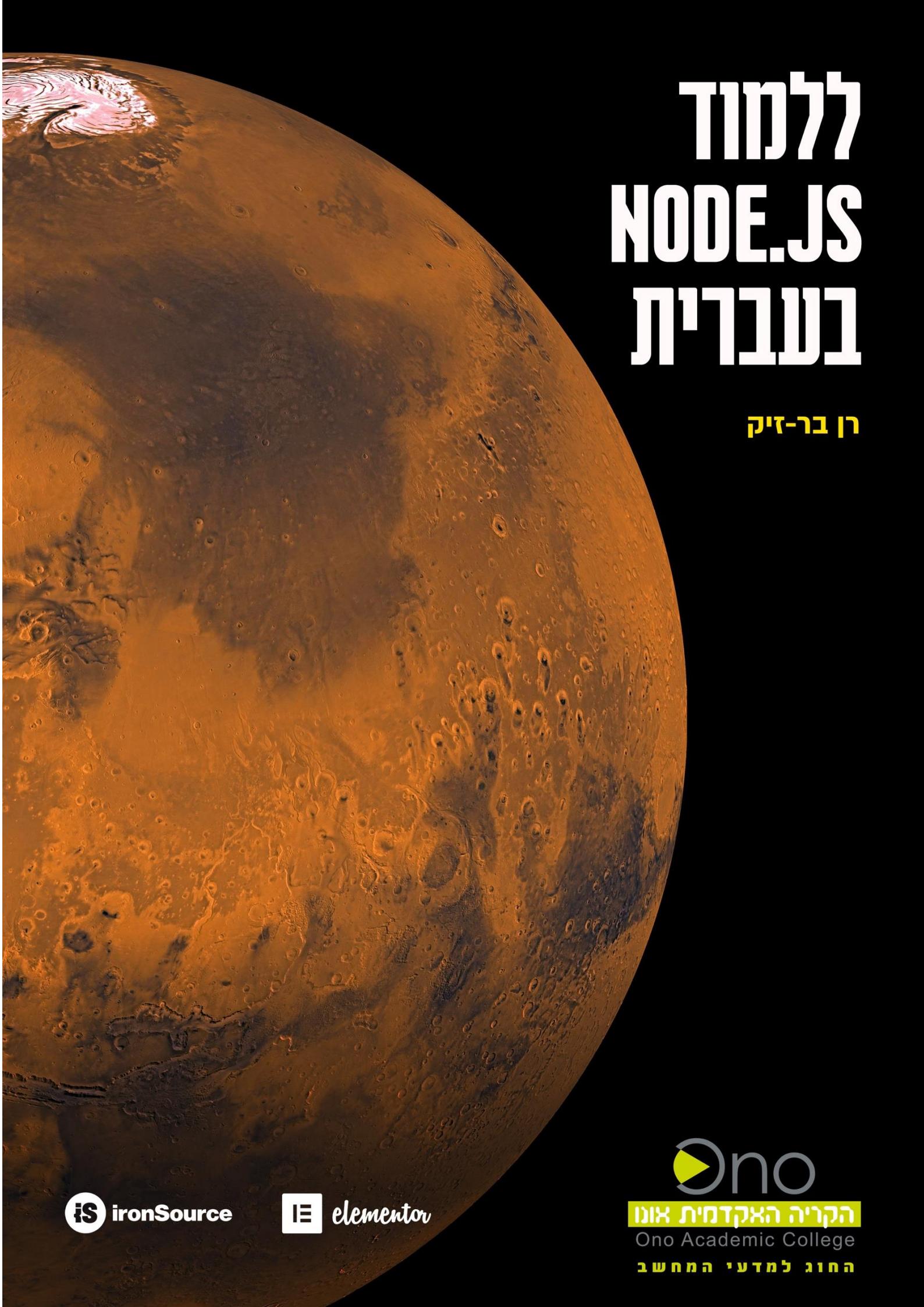
הספר אינו מוגן במערכת ניהול זכויות. כלומר ניתן לקרוא אותו בכל התקן ללא הגבלה. אם מתחשך לקרוא גם מהקינדל, גם מהמחשב וגם מהטלפון הנייד אין בעיה להעתיק את הקובץ שלוש פעמים ולשים אותו בתוך כל התקן. מתוך תקווה שהרוכש והותמן לא ינצל את האמון שנתתי בו להעתקה סיטונאית של הספר לאנשים אחרים והפיצה שלו. אני מאמין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

העתק זהה נמכר ל:

anguru@gmail.com

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

תודה וקריאה נעימה!



ללמוד NODE.JS בנברית

ר' בר-זיק

ללמוד Node.js בעברית

רן בר-זיק

מהדורה: 1.0.0



כל הזכויות שמורות © רן בר-זיק, 2019.

ספר זה הוא יצירה המוגנת בזכויות יוצרים. אתה קובלת רשיון לא-בלודי, לא-ייחודי, אישי, בלתי ניתן להעברה (למעט על פי דין), ובבלתי ניתן להסבה לעשות שימוש אישי בספר זה לצרכים לימודים בלבד.

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

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

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

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

הפקה: כריכה – סוכנות לסופרים
www.kricha.co.il



תוכן עניינים

10.	על הסוף
10	על המונחים בעברית
12	על המחבר
13.	על העורכים הטכניים
13	בנג'מין גריינבאום.....
13	גיל פינק.....
14.	הקדמה – מה זה ואיך זה התחיל
16.	דרכן הלמידה
17.	התקנת סביבת עבודה ועבודה עם טרמינל
20	התקנה על חלונות.....
23	התקנה על מק
23	התקנה על לינוקס.....
24	תקלות נפוצות.....
24	 כתיבת התוכנה הראשונה
30.	וחודולים <i>Require</i>
35.	היכרות עם הדוקומנטציה של <i>Node.js</i>
43.	גרסאות סינכרוניות למתחוזות אסינכרוניות
50.	<i>NPM</i> – הקרה ראשונית והפעלה של <i>package.json</i>
51	יצירת <i>nodejs</i>.<i>package</i> לפרויקט שלנו
53	התקנת המודול הראשון
55	שימוש במודול חיצוני
60.	עבודה אסינכרונית ומעבר מוקלבים לפורmisים ול-<i>async/await</i>
64	מודולים ב-<i>Node.js</i> שתומכים בפורמייסים באופן טבעי
67.	איירושים

כיבוי מאzin	70
הפעלת יותר מאירוע אחד	71
הצמדת כמה פונקציות מאזינoot לאיירוע אחד	73
העברת נתונים באירועים	75
יצירת שרת HTTP בסיסי	79
ה-<i>Node.js</i> של Event Loop	88
מתודות הטימרים	88
תורץ כשאני אומר לך – <code>setTimeout</code>	88
תורץ מייד עם קולבק – <code>setImmediate</code>	89
הלולה הקבועה <code>setInterval</code>	89
טור הקריאה	90
Streams	99
סוגי הסטרוימים השונים	101
סטרוים טרנספורמציה	102
אירועים בסטרוימים	103
אריזת הקוד שלנו כמודול	110
קביעת גרסאות	116
גרסאות סמנטיות	117
קביעת גרסאות סמנטיות ב-<code>package.json</code>	120
התקנה גלובלית ו-CLI	124
כתיבת <code>bin</code> והתחממות עם ה-CLI	128
Sockets	138
קריאה משאבים באמצעות מודול <code>path</code>	147
package.json scripts	152
סקרייפטים עם שמות	154
משתני סביבה	155
קביעת משתנה סביבה דרך הסקריפט	156

156.....	קביעת משתנה סביבה דרך環境 מערך ההפעלה
158.....	קביעת משתנה סביבה דרך קובץ
159	dev dependencies
164.....	אקספרס
166	טיפול במתודות של בקשות HTTP
171	ראוטינג
174	MiddleWare
176	URL דינמי
179	tabniot
183.....	חיבור ל-MySQL
184	חיבור ראשוני
186	שאילתת בסיסית
187	המרת הקוד לעבודה עם פרומיסים ולא עם קולבקים
189	Prepared Statement
192.....	עליה לפורודקשן
192	עליה לפורודקשן עם שרת
194	עליה לפורודקשן בענן
197.....	סינון
197	מיטאפים
197	קבוצות דיוון
198	גיטהאב
198	התנדבות בעמותות ובמיזמים
199.....	נספח: בדיקות אוטומטיות ב-Node.js
199	מה זה בדיקות אוטומטיות?
202	בדיקות אוטומטיות
203	Mocha
206.....	describe

207.....	it
208.....	מחזור חיים
211.....	מבנה בדיקה
212	פרויומוורק בבדיקות
213.....	assert.ok(value)
214.....	assert.notStrictEqual(actual, expected) assert.strictEqual(actual, expected)
214.....	assert.notDeepStrictEqual(actual, expected) assert.deepStrictEqual(actual, expected)
215.....	assert.throws(fn)
215	ספריות נוספת
216.....	ספריות mock
219	1-spies.js
220.....	mock(obj)
221.....	בדיקות עם קריאות http
224	סוגי בדיקות
224.....	בדיקות יחידה
224.....	בדיקות קומפוננטה
226	סוגים נוספים של בדיקות
226.....	eslint
229.....	npm audit
230	از מה יוצא לי מזה?

על הספר

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

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

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

על המונחים בעברית

אני כותב בעברית על טכנולוגיה ותוכנות כבר יותר מעשור והדילמה "באילו מונחים בעברית להשתמש" מלווה אותי תמיד. מצד אחד, האקדמיה ללשון העברית מספקת לנו מונחים רבים בעברית. מצד שני, בתעשייה ההייטק, שמננה אני מגיע, איש לא משתמש ברבים מהמונחים האלה. אם תגינו לראיון עבודה ותגידו: "במפגש המתכנתים האחרון שמעתי על דרך חדשה לבצע הידור שבודק הוצאות במנשך מבוססת הבטחות", סביר להניח שלא תקבלו את העבודה. אבל אם תגידו "במיוחד הפעולות שבודק אינדנטציה-ב-API מבוססות

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

חשוב לציין שאיני בכלל לאקדמיה ללשון וshall ממה מונחים שלא אכן נכנסו לשפה המדוברת במרכז הטכנולוגיה השונים (למשל: קובץ או מסד נתונים), אבל בכלל מקום שהיא הייתה לי ברירה בין להיות מובן לבין לעמוד ב כללי הלשון, העדפתה 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-HPE ועד Verizon, שם הוא מפתח בטכניקות מתקדמות הן מצד הלקוח הן מצד השרת, ושם דגש על בניית תשתיות פיתוח נכונה, על שימוש ב-CD\AI וכמו כן על אבטחת מידע.

נוסף על עבודתו כ מפתח במשרה 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), שהוא אתר טכני המכיל מדריכים, מאמרים והסבירים על תכונות בעברית, ומתעדכן לפחות פעם בשבוע.

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

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

על העורכים הטכניים

בנג'מין גריינבאום

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

gil fink

gil fink הוא מומחה לפיתוח מערכות ווב, Web Technologies Google Developer Expert MVP Microsoft Developer Technologies .sparXsys והוא ייסד של חברת sparcy. כיום הוא מייעץ לחברות ולארגונים שונים, שם הוא מסייע לפיתוח פתרונות מבוססי אינטרנט SPA. הוא עורך הרצאות וסדנאות לייחדים ולחברים המעוניינים להתחמות בתשתיות, בארכיטקטורה ובפיתוח מערכות ווב. הוא גם מחבר של כמה קורסים רשמיים של מיקרוסופט Pro Single Page Application (Microsoft Official Course MOC) .AngularUP (Apress) "Development לשוטף בARGINON הכנס הבינלאומי Gil Fink".

[לפרטים נוספים על גיל:](http://www.gilfink.net) <http://www.gilfink.net>

הקדמה – מה זה ואיך זה התחיל

Node.js הופיע בשנת 2009. מדובר בסביבת הריצה של ג'אוوهסקרייפט בסביבת שרת. סביבת הריצה זו בנויה כולה על מנוע הריצה של כרום 78. מדובר במנוע חזק ומהיר מאוד המשמשים בו בכרום. ב-node.js ההרצה היא מחוץ לדפדפן, אך מנוע 78 מאפשר לג'אוوهסקרייפט לroz מהר מאוד ויעיל מאוד. מרכיב נוסף של node.js היא ספריית `uv.lib`, הכתובה ב-C ומאפשרת הריצה של פעולות קלט ופלט במהירות רבה.

מתכנת בשם ריאן דאל רצה לבנות סמן התקדמות של תעינות קובץ. הוא ניסה לעשות זאת בשרתים הקודמים, ובראשם Apache, אך לא הצליח לעשות כן בגלל בעיות ביצועים. הוא החליט לבנות שרת מבוסס על 78 המהיר, עם דרכי פשוטות לבצע קלט ופלט למערכת הפעלה ועם ג'אווהסקרייפט.

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

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

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

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

כיום מי שМОבייל את Node.js הוא מוסד ללא כוונת רוח שנקרא OpenJS Foundation – מוסד שבנו על פי עקרון "הממשל הפתוח" וכל אחד שיש לו מספיק רצון יכול להשתתף בדיונים ולהשפייע על התפתחות העתידית של סביבת הריצה.

Node.js היא לא שפה, השפה היא ג'אוوهסקרייפט. Node.js היא סביבת הריצה. כל לכל מתכנת או מתכנתת ג'אוوهסקרייפט לעבוד היטב עם Node.js. ספר זה אינו מלמד ג'אוوهסקרייפט ואני יוצאת מנקודת הנחה שהקוראים מכיריהם היטב ג'אוوهסקרייפט ובדגש על ג'אוوهסקרייפט מודרני ואסינכורי. אם איןכם מכיריהם היטב את השפה זו, אני ממליץ לכם לקרוא את ספרי הקודם "למידה ג'אוوهסקרייפט בעברית", שייצא בהוצאה הקרידית האקדמית אוננו. לימוד של הספר הקודם יביא אתכם למצב שתוכלו להבין את הספר הזה היטב.

בספר נלמד Node.js משלב ההתקנה ועד השלב שבו נדע לשЛОט בה באופן מושלם. הדבר החשוב ביותר שנדאי לזכור ב-node.js הוא שעוצר הספריות העצום שלו בעצם חוסך המון מזמן הכתיבה. אנו נלמד פה למשל איך מקימים שרת HTTP, אך הסיכון שתctroco לעשות את זה בחיים האמתיים הוא אפסי, כיון שהמודול הפופולרי Express משמש את רוב המתכנתים ליצור שרת HTTP. כמו כן תוכלו להשתמש בידע שתלמידו בספר הזה כדי להוסיף למודולים קיימים או לנכתב אפליקציות של ממש או שירותים של ממש שימושים במודולים של Node.js. ברגע שתבינו איך עובדים עם ג'אווהסקרייפט על סביבת הריצה זו – השמיים הם הגבול. כאמור, משתמשים(Node.js) בכל מקום. גם במקומות שבהם כתבים בעיקר בשפות תכנות אחרות, כיון שהכח של Node.js הוא ביכולת שלו לפעול בכל מקום, גם במשימות תחזקה וגם במשימות של אבטחת מידע.

דרך הלמידה

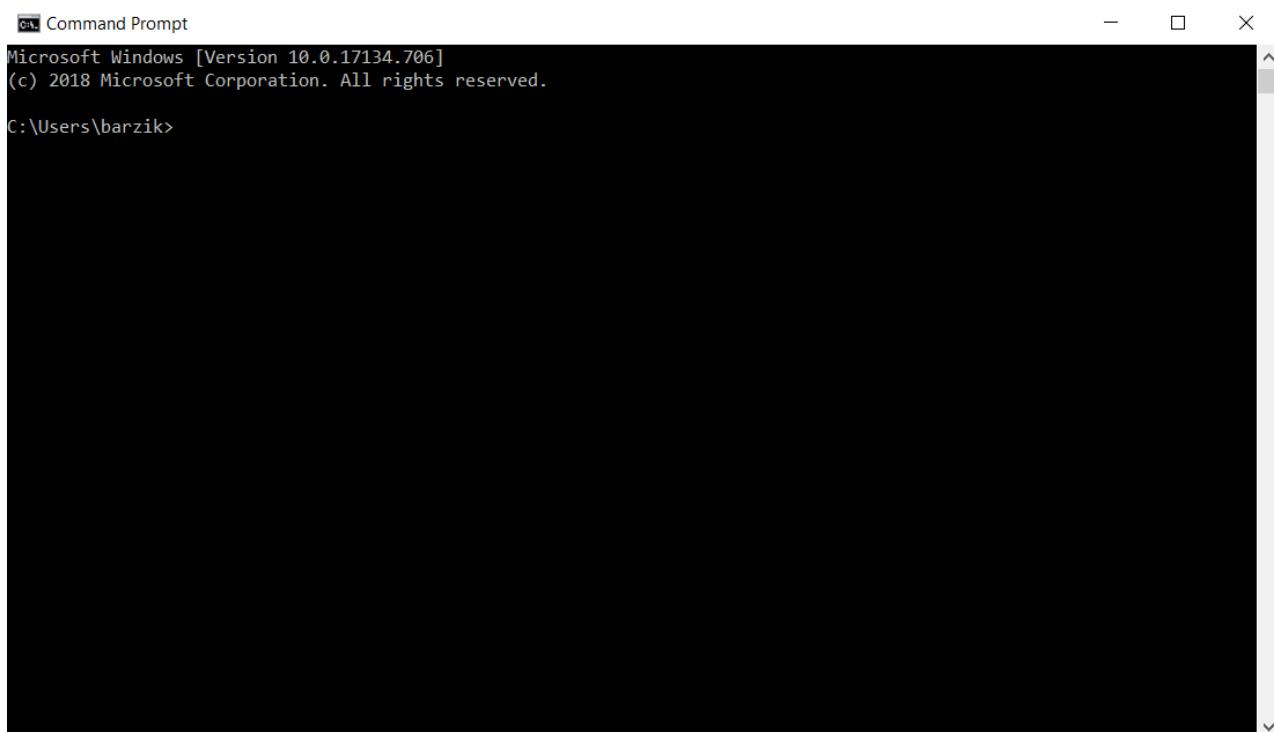
דרך הלמידה היא פשוטה ביותר – קריאת הפרק ותרגול של התרגילים שנמצאים בסופו. התרגול הוא קריטי, בגלל זה חשוב מאוד להשיקע זמן בפרק הראשון ולבנות את סביבת העבודה שלכם. ללא בניה של סביבת העבודה ותרגול – הקריאה לא תהיה אפקטיבית ממש. ראשית יש להבין את החומר, לקרוא פעם, או פעמיים או שלוש, וכך לבצע את התרגילים.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

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

לא תמיד כל הסבר המופיע בספר הוא כולל או מתאים. אם קראתם את הפרק פעם ופעמיים ושלוש פעמים ועדין לא הבנتم – הבעיה לא בכם אלא בהסביר. לא להתייחס – פה כדאי להתייעץ בקהילות של ג'אווהסקריפט ויש לא מעט כאלה בפייסבוק ובמקומות אחרים. גם חיפוש בגוגל לפעמים יכול להוציא אתכם מבוז אמית. נתקעתם? אל תהייאשו – הבעיה לא בכם. Node.js היא קלה אבל יש בה כמה חלקים קשים. נתקעתם? לא לדאוג – בקשו חילוץ. חפשו בגוגל, שחקו שוב ושוב עם הדוגמאות ובסוף זה ישב. אם אני הצלחתי – כל אחד יכול.

התקנת סביבת עבודה ועבודה עם טרמינל

כאמור, jsNode היא סביבת הריצה, וכדי שהיא תוכל לרוץ צריכה אותה על המחשב, ממש כמו כל תוכנה אחרת. מה שההתקנה זו עשוה הוא פשוט למדי – היא מאפשרת לנו להפעיל את jsNode כמו כל תוכנה אחרת. זה הכל. אנו יכולים 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 מגיעים לטרמינל באמצעות לחיצה על הזוכנית המוגדלת (בחילונית 10) והקלדה של cmd – ראשית תיבות של command. נגיע לחלון שנראה כך:

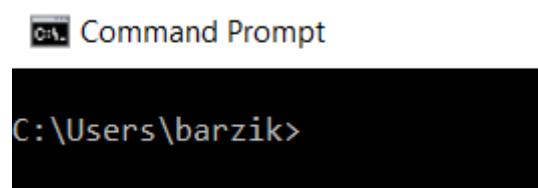


פתחו את החלון הזה, הקלידו notepad ולחצו על enter. ופתחו ה-notepad של חלונות. ברכותי! הפעלתם תוכנה באמצעות שורת הפקודה. הקלידו calc ולחזו על enter, תוכנת המחשבון Tipcal. זו עוד תוכנה שהפעלתם באמצעות משקל הפקודה. הממשק זהה, או הטרמינל בשפת העם, והוא סביבת העבודה של jsNode. מפעילים את jsNode באמצעות הטרמינל. זה בדוק מה שקרה בשורת "אמתוי". זכרו שררת בסופה של דבר הוא מחשב – ייתכן שמחשב ללא מערכת הפעלה גרפית אלא רק עם טרמינל – אבל מחשב שבוסס על חלונות או על Linux. ייתכן שהשרת חזק בהרבה מהמחשב הביתי שלכם – אבל עדין מדובר במחשב לכל דבר. כאמור, jsNode רצה היטב

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

צורך להכיר מעט את משק הפקודה של הטרמינל ולהתמצא בו. כיוון שהטרמינלים שונים בין חלונות לlinpus, יש שוני קטן בין הפקודות. כיוון שחלונות היא מערכת הפעלה הנפוצה, ומשתמשי linpus בדרך כלל מ信נים יותר בטרמינל, אני מסביר פה על הפקודות בחלונות. בסוף הפרק יש טבלה קטנה שבה מובאות הפקודות בלינוקס ובחלונות.

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



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

אפשר לראות שאנו נמצא בדיסק C, בתיקיית Users ובתת התיקייה barzik, זהה 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 שאנו יכול לעשות במשק הגרפי אני יכול לעשות בטרמינל.

כדי לראות את רשימת הקבצים בתיקייה, אני צריך להקליד dir. הקלדה של dir ואז enter תראה לי את רשימת הקבצים שיש בתיקייה שבה אני נמצא. רשימת הקבצים זו תהיה זהה לחלווטין לרשימת הקבצים שאנו רואה בסיר הקבצים כשהוא 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 רווח בשם התיקייה, אני צריך להקליד אותו במירכאות. אם אני משתמש ב-TAB הוא יעשה זאת זה עבורו.

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

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

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

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

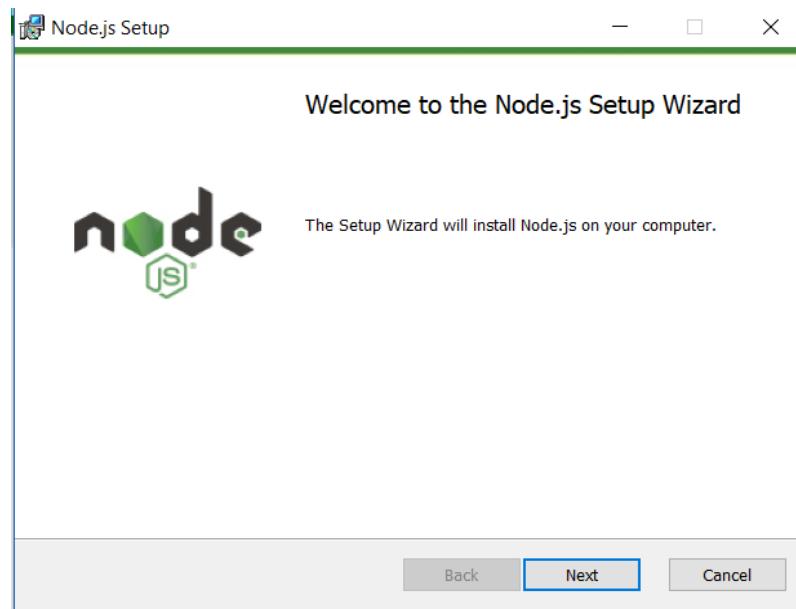
פקודה בLinux/מック	פקודה בחלונות	פקודה
<code>ls -al</code>	<code>dir</code>	הציג רשימת הקבצים והתיקיות בתיקייה
<code>cd</code>	<code>cd</code>	עברו לתיקייה אחרת
<code>exit</code>	<code>exit</code>	יציאה מהטרמינל
<code>clear</code>	<code>cls</code>	ניקוי המסך

לאחר שננו יודעים איך לעבוד עם הטרמינל, נתקן את `Node.js`. ההתקנה שונה במערכות הפעלה שונות אבל ביכולן היא קלה למדי. בחרו את מערכת הפעלה שלכם ותתקינו את `Node.js` לפי ההוראות.

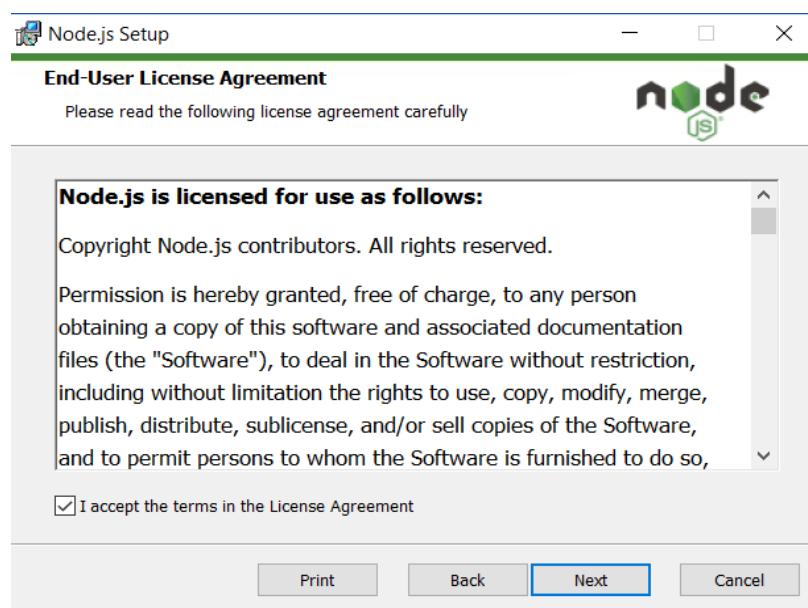
התקנה על חלונות

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

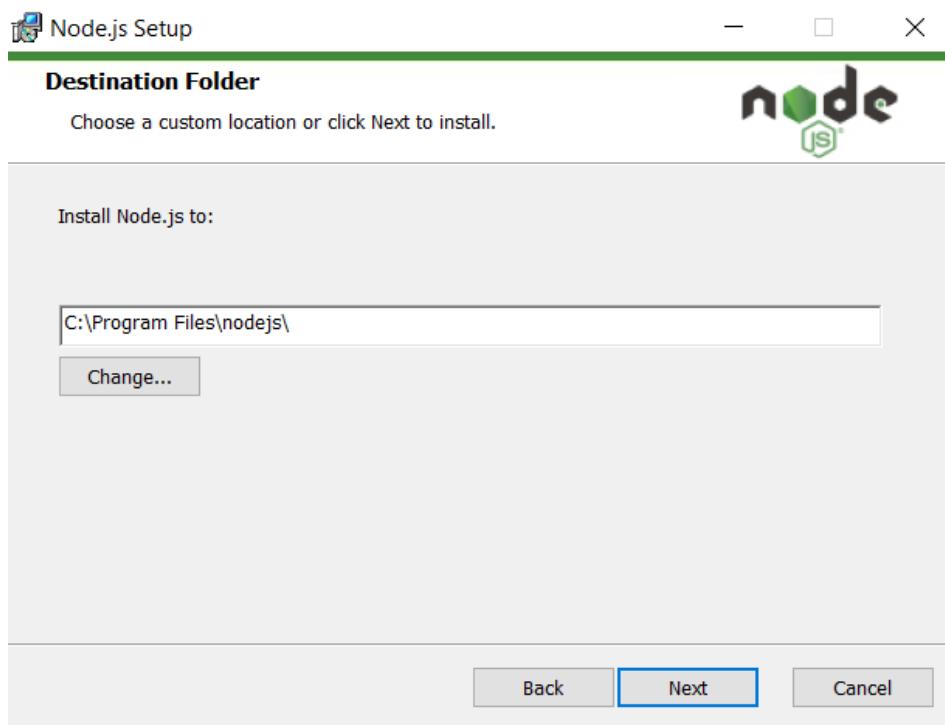
אנו נבחר בגרסת LTS – ראש תיבות של "גראף לטוח אורך", ונבחר במערכת הפעלה שלנו – אם מדובר בחלונות, יש לנו installer נוח. מוריידים, לוחצים על התוכנה שיורדת:



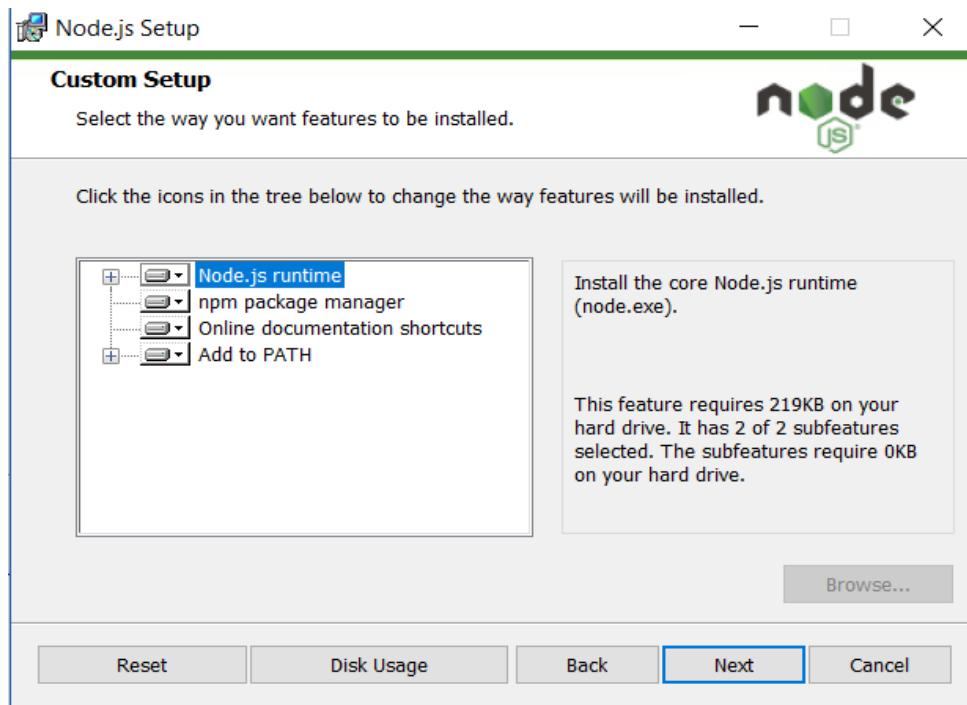
מקבלים את התנאים:



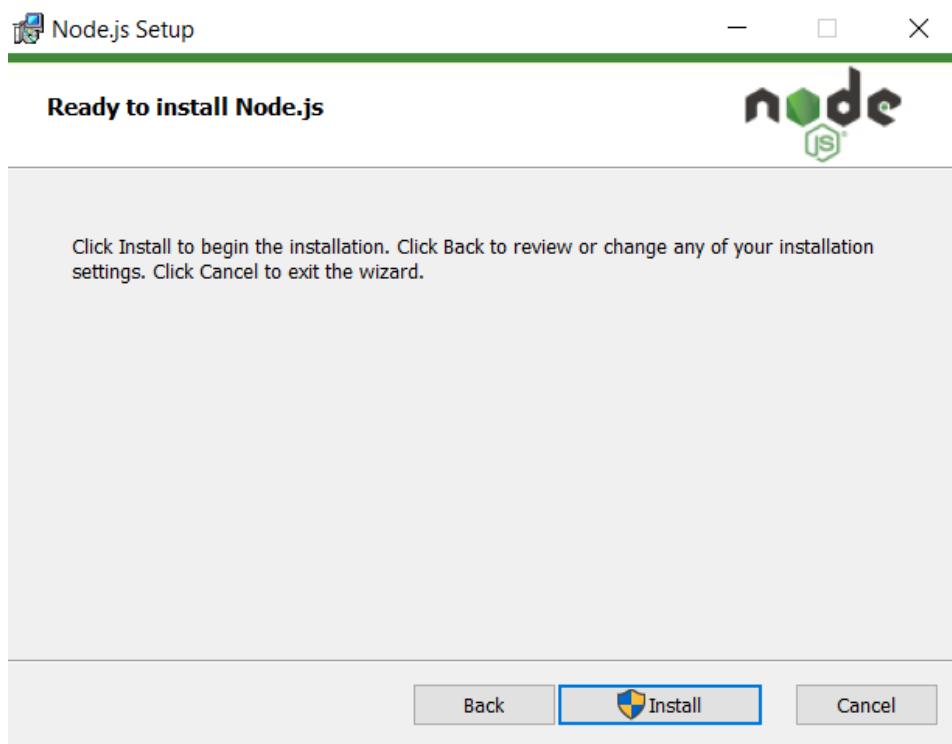
לוחצים על next:



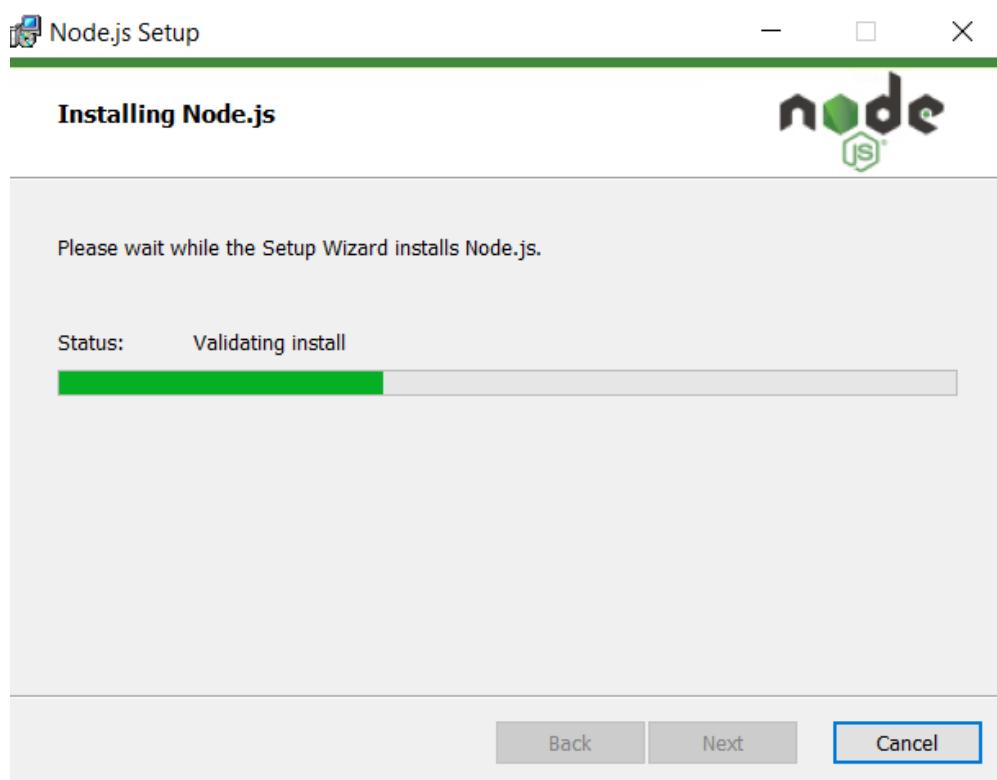
לשוב על next:



לחיצה על **Install** תתקין לבסוף את התוכנה:



כל מה שנouter הוא לחכות לסיום ההתקינה:



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

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

C:\Users\barzik>
```

התקנה על מק

ההתקנה של `Node.js` על מק היא פשוטה מאוד. נקליד בוגול `Node.js Download` או ניכנס אל:

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

אנו נבחר בגרסה LTS – ראשית תיבות של "גרסה לטוח ארוך", ונבחר במק – ייד קובץ `dmg` שהוא אפשר להתקין כמו כל תוכנה אחרת בהנחה שהמחשב שלכם הוא לא מחשב ארגוני שמונע התקנות מהאינטרנט. ההתקנה היא פשוטה ביותר.

אם אתם משתמשים ב-`Zsh` או ב-`Oh My Zsh` אז אני ממליץ להתקין את `Node.js` בעזרת

`homebrew`, באמצעות הפקודה (אם `homebrew` מותקנת אצלכם, וכך שיהיא מותקנת):

`brew install node`

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

התקנה על לינוקס

אם אתם משתמשים בדبيان, אז בדרך כלל ברוב ההפצאות `sudo apt-get install node` יטפל בהתקנה, אך אם עלולים להתקין גרסה ישנה של `Node.js` זה עשוי להיות בעיה. למרות הפיתוי, הייכנסו 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 ממשתמשי חלונות ויודעים להתקין חבילת תוכנה ללא הסברים נוספים. כך או אחרת – לאחר ההתקנה, כניסה לטרמינל והקלדה של -node תראה לכם את מספר הגרסה בדיקן כמו בחלונות או במק.

תקלות נפוצות

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

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

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

כתיבת התוכנה הראשונה

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

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

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

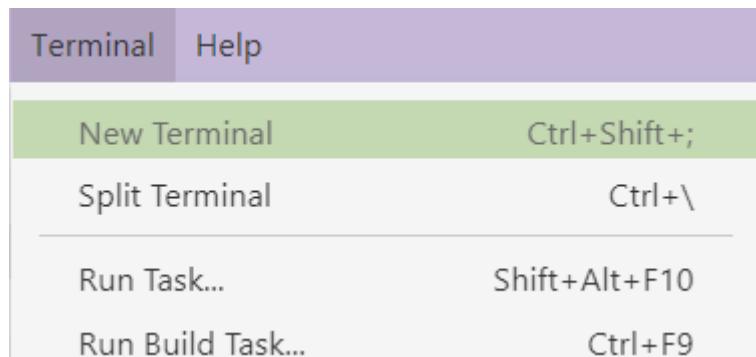
```
console.log('Hello World!');
```

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

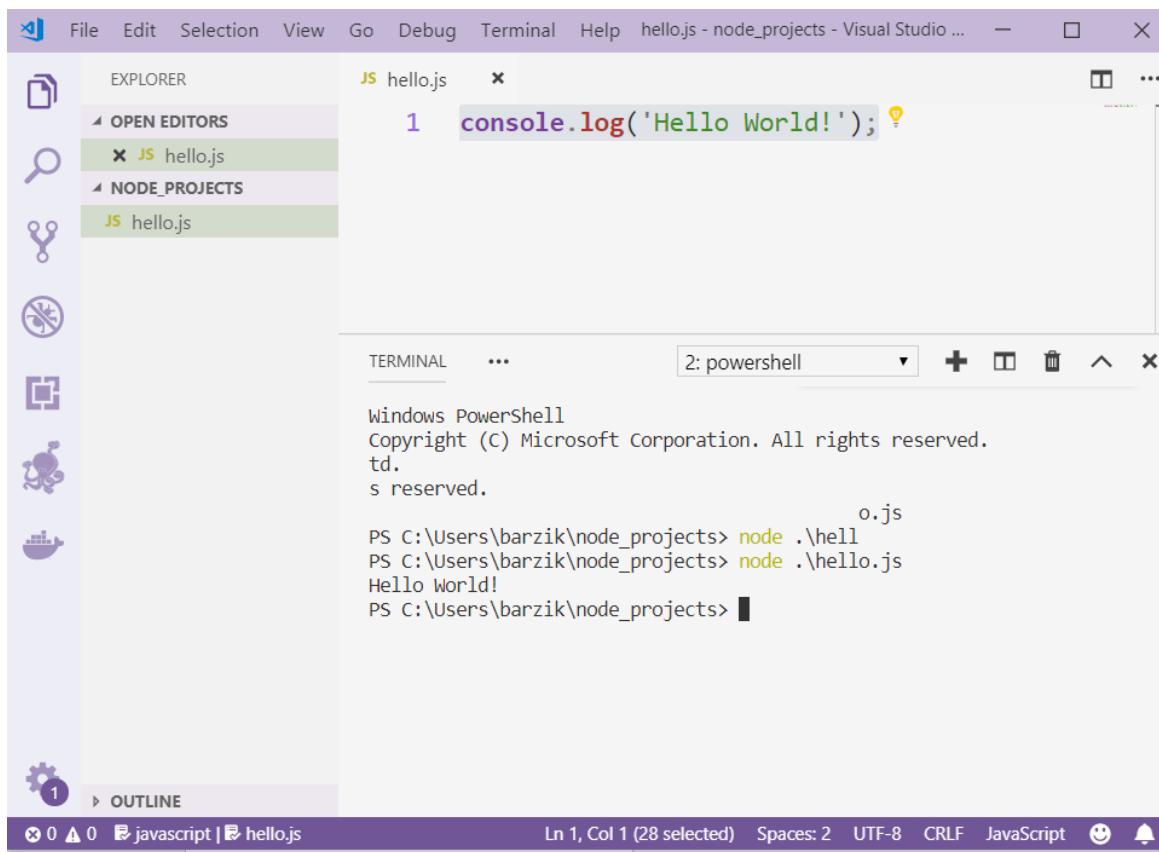
```
C:\Users\barzik\node_projects>node hello.js
Hello World!
```

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

הערה חשובה נוספת: ב-`Visual Studio Code` וגם בסביבות עבודה אחרות טרמינל מובנה ב-IDE. אפשר בטעות העלין `Terminal` ולהציצו עליו. בחרו ב-`New Terminal`. ייפתח לכם בתחום המסך טרמינל במיקום של הקבצים שלכם.



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

**תרגיל:**

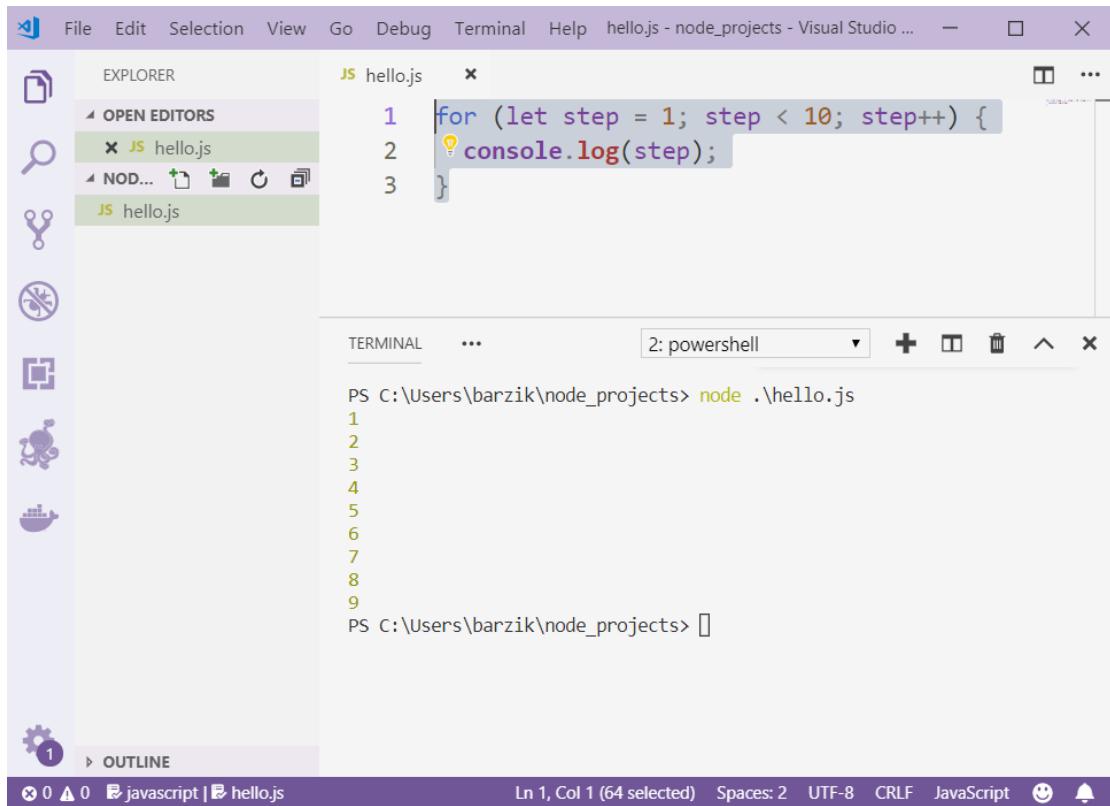
כתבו לולאה שרצה מ-1 עד 10 ומדפיסה את המספר של הלולאה. הריצו אותה ב-node.js.

פתרון:

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

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

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



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

תרגיל:

כתבו קוד שזרוק הערת שגיאה. הריצו את הקוד.

פתרונות:

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

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

אני נכנס למקום התיקייה, באמצעות הטרמינל במערכת הפעלה שלי או באמצעות הטרמינל ב-IDE שלי. אני מupilד לוודא שאני בתיקייה שבה נמצא הקובץ וכותב node hello.js. אני רואה שגיאה וגם את ה-stack trace – השרשרת של פקודות שהובילה לשגיאת. בראשיתה אני בעצם רואה את הסיבה לבעה – השורה הראשונה בתרגיל:

```

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

PROBLEMS TERMINAL ...
2: powershell + - X ^

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

^

Error: This is an Error!
    at Object.<anonymous> (C:\Users\barzik\node_projects\hello.js:1:69)
    at Module._compile (internal/modules/cjs/loader.js:701:30)
    at Object.Module._extensions..js (internal/modules/cjs/loader.js:722:10)
    at Module.load (internal/modules/cjs/loader.js:600:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:539:12)
    at Function.Module._load (internal/modules/cjs/loader.js:531:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:754:12)
    at startup (internal/bootstrap/node.js:283:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:622:3)

```

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

פרק 1

REQUESTS וМОודולים REQUIRE



Require ומודולים

הכוח הגדול של Node.js הוא חבילות התוכנה שלו. ל-node.js יש יותר ממאה אלף חבילות תוכנה שכל אחד יכול להשתמש בהן. איך משתמשים בהן? באמצעות **require**. אחד ההבדלים הגדולים בין node.js לבין ג'אווהסקריפט בסביבת DFDFN הוא require. הוא כמעט ייחודי ל-node.js (יש ספריות נוספות שימושísticas בו, אך ללא ספק הוא סימן היכר משמעותי של node.js) וממש שיבוא ולשימוש בחבילות תוכנה. ל-node.js יש חבילות תוכנה שבאות איתה כבירה מחדל ואנו נשתמש בהן בהתחלה.

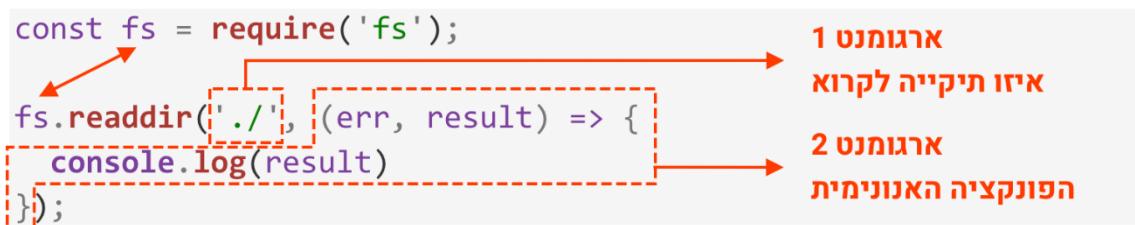
חבילת התוכנה הראשונה שאנו נשתמש בה היא `fs`, שידועה גם כ-`file system` שבאה תמיד עם node.js (אי-אפשר להתקין את node.js בלבד) ומסייעת לנו לטפל במערכות הקבצים של המחשב. יש לה מתודות רבות שמתפעלות את מערכת הקבצים. מתודה אחת שאבחן בה היא `readdir` – מתודה שמקבלת שני פרמטרים. הראשון הוא הנתיב של התקייה שאני רוצה לבדוק והשני הוא פונקציית קולבק. פונקציית הקולבק נקראת על ידי `readdir` בגמר הפעולה ומחזירה שני ארגומנטים – אובייקט שגיאה (אם היא מתקימת) או אובייקט תוצאה שמציג את הקבצים והתקיות שיש לנו בתיק.

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

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

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

מה הקוד הזה בעצם אומר? הדבר שלא אמרו להיות ברור לתוכנת ג'אווהסקריפט מן השורה הוא `require`. כאן אני בעצם קורא לחבילת התוכנה או למודול שנקרא `fs`. מזכיר בכך עצם באובייקט כמו כל אובייקט שאנו מכירים, שיש לו מתודה שנקראת `readdir`. זו מודה שמקבלת אסינכרונית שמקבלת ארגומנט ראשון של התקייה שבה מחפשים וארוגמנט שני של קולבק. בקולבק יש שני ארגומנטים – אובייקט שגיאה ואובייקט תוצאה. זה פורמט סטנדרטי של קולבקים ב-node.js.
ואני ארכיב על כך בפרק על פורמייסטים ב-node.js.



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

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

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

```

const fs = require('fs');

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

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

אפשר לבצע `require` לכמה מודולים בו-זמנית. מודול נוסף שבא יחד עם `Node.js` הוא מודול `os`, שנוטן מידע על מערכת הפעלה. מתוודה אחת מתוך `os` היא `homedir` – המתוודה הזו לא מקבלת ארגומנטים, ומחזירה את תיקיית ה"בית" של מערכת הפעלה. אם אני למשל בחלונות, תיקיית הבית שלי היא `C:\Users\barzik`, תיקיית הבית של רן היא `/home/barzik`.

אם אני כותב סקרייפט של `js`Node, אני רוצה שהוא יעבד בלי קשר למערכת הפעלה ואני לא מעוניין לדעת מה היא. שימוש ב-`os` הוא הדרך.

כך נכתב את הקוד:

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

const homeDirectory = os.homedir();

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

אפשר לראות שפשוט עשיתי `require` ל-`os` והשתמשתי במתודה `homeDir`. מדובר במתודה סינכרונית שלא מקבלת קולבק, אז אין בעיה מהותית להשתמש בה. `require` הוא לא קסם או וודו אפל. מדובר בקבלה של מודול, וברגע שקיבلت אותו אני יכול להשתמש בו בדיקות כמו שאני משתמש בכל מודול אחר בג'אווהסקרייפט. כך למשל, אם אני רוצה ליעיל את הקוד הנוכחי ולהוסיף שורה, אני יכול לבצע `require` ל-`os` ומיד 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 חושב שמדובר במקרה שליל, אבל היא אמורה להבהיר לכם שלא מדובר במקרה. באחד מהפרקים הבאים נראה מקרה איך ה-`require` עובד כאשר נכתב מודול מסוינו.

תרגילים:

צרו תוכנת `js`Node שתיצורקובץ בתיקייה ומידי אחרי כן תציג את הקבצים בתיקייה (אחד מהם אמור להיות הקובץ).

פתרונות:

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

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

הדבר הראשון שאנו עושה הוא `require('fs')`. אני יוצר את הקובץ עם מethodת `writeFile` ואני מעביר לה שלושה ארגומנטים. ארגומנט ראשון הוא שם הקובץ שאותו אני יוצר, הארגומנט השני הוא ה-`Hello world` והשלישי הוא קולבק. פונקציית הקולbak נקראת אחרי שהקובץ סימן להיווצר. בתוכה אני אבצע קריאה נוספת ל-`fs`, לקרוא התיקייה ולהדפסת התוצאות.

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

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

קולבק: פונקציה חוץ

**הfonktsia נמצאת
בתוך הקולbak**

פרק 2

היכרות עם הדוקומנטציה של NODE.JS



היכרות עם הדוקומנטציה של Node.js

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

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

בואו נסתכל על הדוקומנטציה של readdir על מנת לנסות להבין:

סוגרים מרובעים - אפשרי ולא חובה

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



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

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

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

השלישי, הוא הקולבק שלנו. זה פונקציה (מקובל להעיבר פונקציית Hz) שמופעלת לאחר שהפעולה מסתיימת. ככלומר ברגע ש-`fs.readdir` מסיימת לקרוא את תוכן התיקייה, היא לוקחת את הפונקציה שהעבכנו כארוגומנט שלישי ומפעילה אותה. כשהיא מפעילה אותה היא מאכלסת את שני הארגומנטים בקולבק. אני יכול לטפל בקולבק בארגומנטים אלו או לא. הינה מה שיצרתי בעקבות הקריאה בדוקומנטציה:

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

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

כדי לשים לב שביקשתי 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 תראו מהهو 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` שיכול להיות מחרוזת טקסט (ויכול להיות גם סוג אחר), השני הוא אובייקט אפשרויות שהוא לא חובה, והשלישי הוא הקולבק. הקולבק הזה-Amor להיות מופעל כשהמתודה `readFile` מסיימת את תפקידה. היא תפעיל את הפונקציה הזו ותעביר אליה שני ארגומנטים, שגיאה (אם קיימת) ואת המידע.

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

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

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

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

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

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

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

The callback is passed two arguments (`err, data`), where `data` is the contents of the file.

If no encoding is specified, then the raw buffer is returned.

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

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

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

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

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

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

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

תרגיל:

אתרו לדוקומנטציה את המתודה `fs.mkdir` והשתמשו בה בסкриיפט של Node.js על מנת ליצור תיקייה במיקום כלשהו במחשב שלכם.

פתרון:

המתודה `fs.mkdir` נמצאת לדוקומנטציה תחת File System פה:

https://nodejs.org/api/fs.html#fs_fs_mkdir_path_options_callback

אם נסתכל עלייה נראה שהיא למתודות של `readFile` ו-`readdir`. גם היא מקבלת שלושה ארגומנטים:

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

► History **Argument Shallow** **Argument First** **[No Chobah]** **Callback**

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

Argument First הוא המיקום שבו רוצים ליצור את התיקייה החדשה, Argument Second הוא אפשרויות, והArgument Shallow הוא הקולבק שנקרא לאחר השלמת הפעולה. הקולבק הזה מקבל Ark וرك אובייקט שגיאת אם הפעולה נכשלה. כדאי לשימוש בו שהוא תמיד מופעל לאחר הצלחת הפעולה ואם אין שם אובייקט שגיאת, אין יכול להיות שהפעולה הצליחה.

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

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

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

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

תרגיל:

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

פתרונות:

גם כאן קל להשתמש בדוקומנטציה של `Node.js` על מנת לאתר את המתודה 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 ורק אובייקט שגיאת אם הפעולה נסלת.



הקוד שלי ייראה כך:

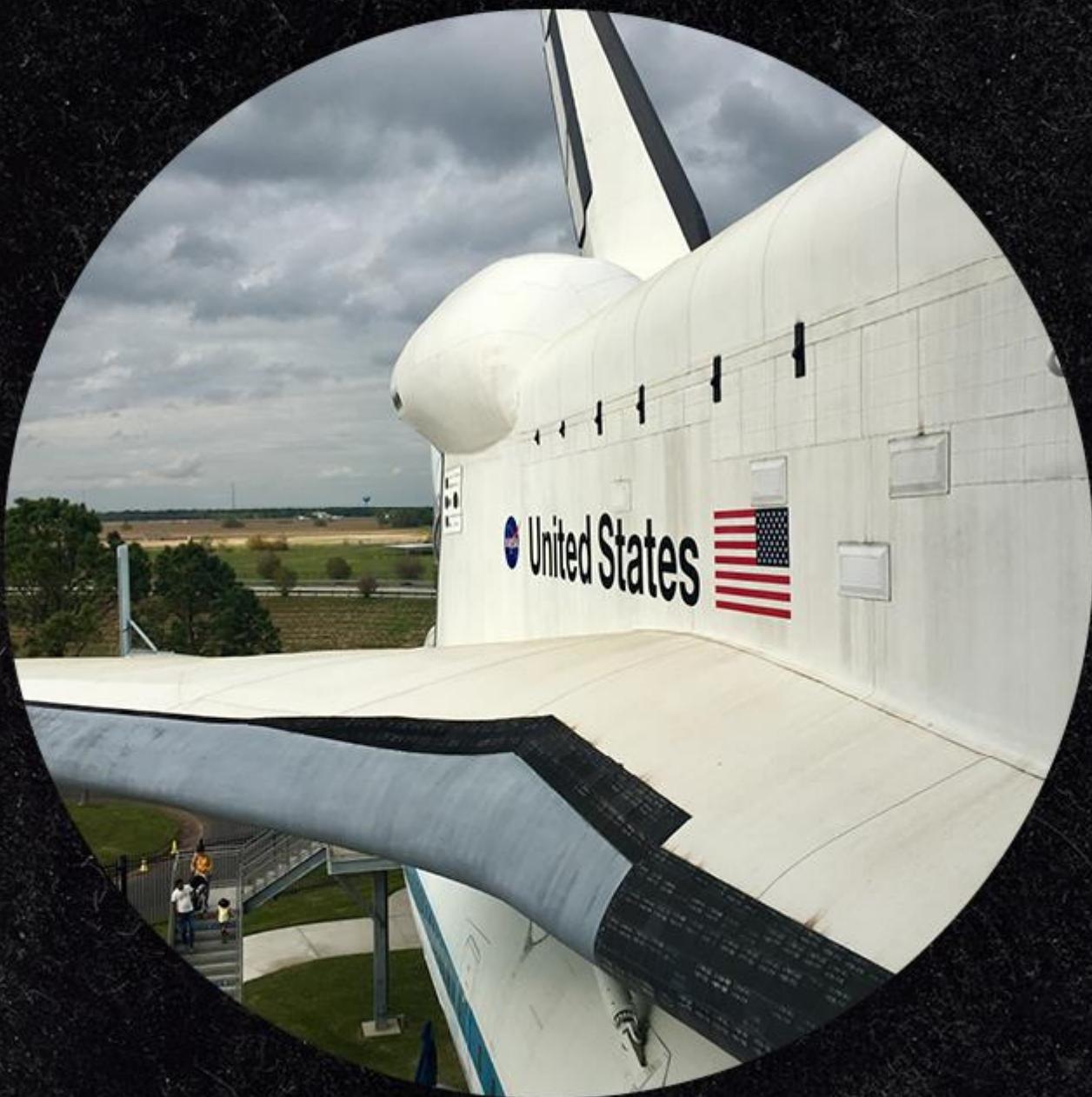
```
const fs = require('fs');

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

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

פרק 3

גורסאות סינכרונית למוחוזות אסינכרוניות



grossאות סינכרוניות למתודות אסינכרוניות

כתבתי בתחילת הספר ש-`fs` הינה אסינכרונית ושהה כוח שלה. `fs` רצה על הליך אחד בmund ואם אנו נדרשים לפעולת מסוימת כמו קוריאת קובץ, התוכנה לא עומדת וממתינה לקובץ זהה, אלא ממשיכת הלהה. אם אני כותב משהו כזה למשל:

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

console.log('Before readdir');

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

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

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

מה שאני רואה בטרמינל הוא זה:

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

למה? כי בהתחלה אנו מדפיסים את ה-`Before`, קוראים ל-`fs.readdir`. בזמן ש-`fs` רצה לтиקיה, אפשר להמשיך ואכן השורה הבאה, שהיא הרצה של ה-`After`, רצה. רק כשה-`readdir` סיימה את העבודה, הקולבק מופעל ו מביא את התוצאות.

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

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

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

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

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

זה אומר שהקוד ממש יעזור ויחכה להשלמת הפעולה. אם אני אשיט `log` לפני ואחרי, אני אראה שהקוד רץ לפי הסדר. אין קולבקים, אין אסינכרוניות:

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

console.log('Before readdir');

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

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

זה מה שאני קיבל בהרצה:

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

ואיך אני תופס שגיאות? במקרה הזה אין לי קולבק שמעביר אובייקט שגיאה (אם יש שגיאה). אז פה אני משתמש ב-`try-catch` רגיל לחלוטין שייפעל אם יש שגיאה. בקוד הבא למשל אני מנסה לקרוא תיקיה שלא קיימת באמצעות `readdirSync` – הפקצייה הסינכרונית תעיף לי שגיאה שאוותה אני יכול לתפוס עם `try-catch` ולטפל בה כראג'ל:

```

const fs = require('fs');

console.log('Before readdir');

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

```

התוצאה של הריצת הקוד זהה תהיה:

```

Before readdir
Error has occurred!
After readdir

```

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

כמעט לכל מתודה שמתפלת בקבצים יש גרסה הסינכרונית שלה.

הינה רשימת המתודות שלמדנו עד כה:

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

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

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

תרגיל:

צרו מחרוזת טקסט רנדומלית עם:

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

בעזרת פונקציה סינכרונית, צרו תיקייה עם שם רנדומלי, דוחחו על הייצירה שלה ואז מחקו אותה.

פתרון:

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

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

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

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

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

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

תרגיל:

בצעו את התרגיל הקודם בעזרת פונקציות אסינכרוניות.

פתרון:

מציאת הגרסאות האסינכרוניות אמורה להיות פשוטה – פשוט להסיר את ה-Sync משם הפונקציה ולהפץ בדוקומנטציה או להזכיר בדוגמאות של הפרקים הקודמים. במקרה זה אנו משתמשים ב콜בקים כי מדובר בפונקציות אסינכרוניות.

כיוון שאנו חיבים לוודא שהתקייה קיימת לפני שنمחק אותה, נבצע את המחיקה ב콜בק של הייצרה. ככלומר ברגע שהתקייה נוצרה, הקולבק של הייצרה מופעל ורק בו אנו יכולים למחוק אותה שנותר.

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

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

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

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

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

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

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

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

הראשוں
מ剔ת התיקיה
נעשית בתוך הקולבך

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

קולבק ראשון
מופעל לאחר יצירת התיקיה

קולבק שני
מופעל לאחר מ剔ת התיקיה

פרק 4

PACKAGE.JSON הנדסה ראשונית והפעלה של NPM



הכרה ראשונית והפעלה – package.json של NPM

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

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

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

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

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

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

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

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

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

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

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

יצירת `package.json` לפרויקט שלנו

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

```

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

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

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

Is this OK? (yes)

```

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

כך הקובץ נראה他自己:

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

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

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

התקנת המודול הראשון

אנו נבחר במודול הפופולרי ביותר ב-NPM, הלא הוא `lodash`. הוא נמצא בכתבota הזו: <https://www.npmjs.com/package/lodash> אבל הוראות התקנה של מודול הん פשוטות. מקלידים `npm install` ואז את שם המודול. במקרה שלנו:

```
npm install lodash
```

אחד הקיצורים המקבילים של `install` הוא `i` ואפשר להקליד:

```
npm i lodash
```

אם תקלידו את השורה הזו בטרמינל במקום שבו יצרתם את `package.json`, שזויה התקינה הראשית שלuproיקט, NPM תוריד את המודול הזה.

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

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

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

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

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

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

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

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

`npm i`

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

שימוש במודול חיצוני

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

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

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

let array = [1, 2, 3];

_.reverse(array);

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

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

בואו נדגים עם מודול אחר. הפעם בחרתי במודול `chalk` שנמצא בכתובת – <https://www.npmjs.com/package/chalk> אנו נראה שיש הוראות התקנה שם, אבל אנו לא צריכים אותן. לפי כתובות המודול אני רואה שהוא `chalk`. אנווט לתקינה של הפרויקט שלי באמצעות הטרמינל (או באמצעות הטרמינל של `Visual studio code`) ואכתב:

```
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
```

אחריו שאלחץ אנטר ואמתין שהפעולה תושלם, אני אראה שני דברים. ראשית, שבקובץ package.json מתווסף המודול chalk ומספר הגרסה שלו. באחד הפרקם הבאים אסביר על הגרסאות, אבל כרגע, קבלו את הנתון הזה כפי שהוא – מספר הגרסה עם הסימן ^.

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

שנית, בתקיית node_modules נוספה תיקייה של chalk שבה נמצא הקוד של המודול. אני יכול כעת להשתמש בchalk.

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

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

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

הקוד הזה נלקח ישירות מהdockumentציה של chalk. על מנת לראות מה הוא עושה אחזור לטרמינל וএকটো `node app.js` ו אז אראה Hello World אבל בכחול.

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

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

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

let array = [1, 2, 3];

_.reverse(array);

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

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

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

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

ככה עובדים עם מודולים חיצוניים וזה הכוח של NPM ושל האkosיסטם של Node.js – שפע של מודולים חיצוניים שמאפשרים המון דברים ויכולת מדיהימה 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.

תרגיל:

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

פתרונות:

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

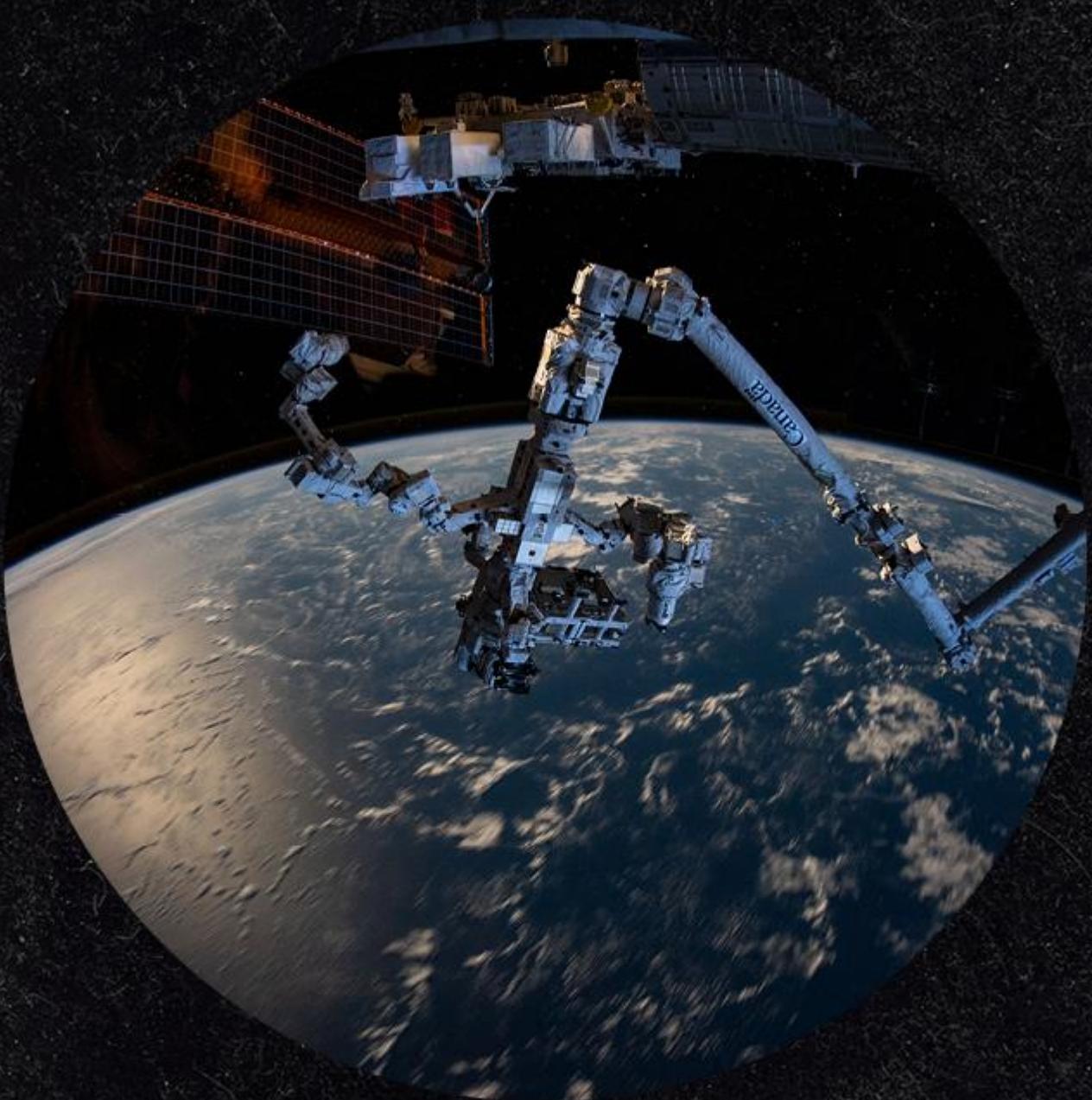
```
const request = require('request');

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

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

פרק 5

עובדת אסינכרונית ומטריך מילולבים לפומיסים ול-`ASYNC/AWAIT`



עבודה אסינכרונית ומעבר מ콜בקים לפרומיסים ול-`async/await`

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

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

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

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

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

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

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

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

זה יעבוד רק על מethodות של מודולים שמקבלים קולבק בהתאם הארגומנט האחרון ובוקולבק יש `error` ו-`value` של חזקה, כמו ב-`readdir`. להזכירם, `readdir` נראה בדיק כהה:

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

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

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

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

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

init();
```

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

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

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

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

init();
```

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

באו נראה איך הכל מתחבר יחד. אני אקח את מודול `fs`, את מודול `dns`, ואצור תוכנה אסינכרונית שעובדת בלי קולבקים. התוכנה שלי תעשה משהו פשוט – היא תיניח את ה-IP של `nodejs.org`, תיצור תיקייה המכتوבת ה-IP'ה זו וAz תדפיס את כל תוכן תיקיית האב.

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

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

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

init();
```

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

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

מודולים ב-`js`Node שתומכים בפרומייסים באופן טבעי

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

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

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

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

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

init();
```

בשעת כתיבת שורות אלו עדין מדובר בפייצ'ר ניסיוני.

תרגיל:

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
 המתודה זו מקבלת כתובת IP ומחזירה את שם המתחם הקשור אליה. יש ליצור פונקציה אסינכרונית שמקבלת IP כפרמטר ומחזירה את כל שם המתחם. כדי לבדוק, נסו את 8.8.8.8.

פתרון:

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

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

reverseIPLookup('8.8.8.8');
```

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

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

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

פרק 9

איזוניים



AIRNODE

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

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

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

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

class MyClass extends EventEmitter { };

const myClass = new MyClass();

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

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

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

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

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

בהמשך הקוד אני פשוט מפעיל את האירוע באמצעות הפונקציה `emit`. הפעלת האירוע גורמת>kolbak להתקיים:

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

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

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

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

```

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.emit. גם פה אני משתמש ב-`this` כי אני מפעיל את emit מתוך הקלאס ולא מבחוץ. אבל הדוגמה צריכה להיות ברורה:

```

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

const myClass = new MyClass();

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

```

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

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

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

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

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

const myClass = new MyClass();
```

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

כיבוי מאזין

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

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

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

```

}

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

};

const myClass = new MyClass();

```

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

הפעלת יותר מאירוע אחד

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

```

const EventEmitter = require('events');

class MyClass extends EventEmitter {

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

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

const myClass = new MyClass();

```

אבל אני יכול גם לבצע האזנה פעם אחת בלבד ואז להרוג את המאזין באמצעות הפקודה `.once`. אם אני מצמיד את ההאזנה באמצעות `once` במקום 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 ומאזינים שונים רוצים לאירוע אחד. זה כל העניין באירועים. מתרחש `emit` אחד – אבל לאירוע הזה שמתקיים יכולות להיות אלף מאזינים שיביצעו אלפי פעולות.

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

class MyClass extends EventEmitter {

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

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

const myClass = new MyClass();
```

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

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

class MyClass extends EventEmitter {

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

```

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

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

};

const myClass = new MyClass();

```

נראה את הפלט הבא:

```

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

```

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

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

```

const EventEmitter = require('events');

class MyClass extends EventEmitter {

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

```

```

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

const myClass = new MyClass();

```

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

העברת נתונים באירועים

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

```

const EventEmitter = require('events');

class MyClass extends EventEmitter {

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

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

const myClass = new MyClass();

```

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

ה-oooooooon Hellllllooooן כМОון הנגע מהAIROU עצמו.

תרגיל:

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

פתרונות:

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

class Dog extends EventEmitter {

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

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

const dog = new Dog();

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

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

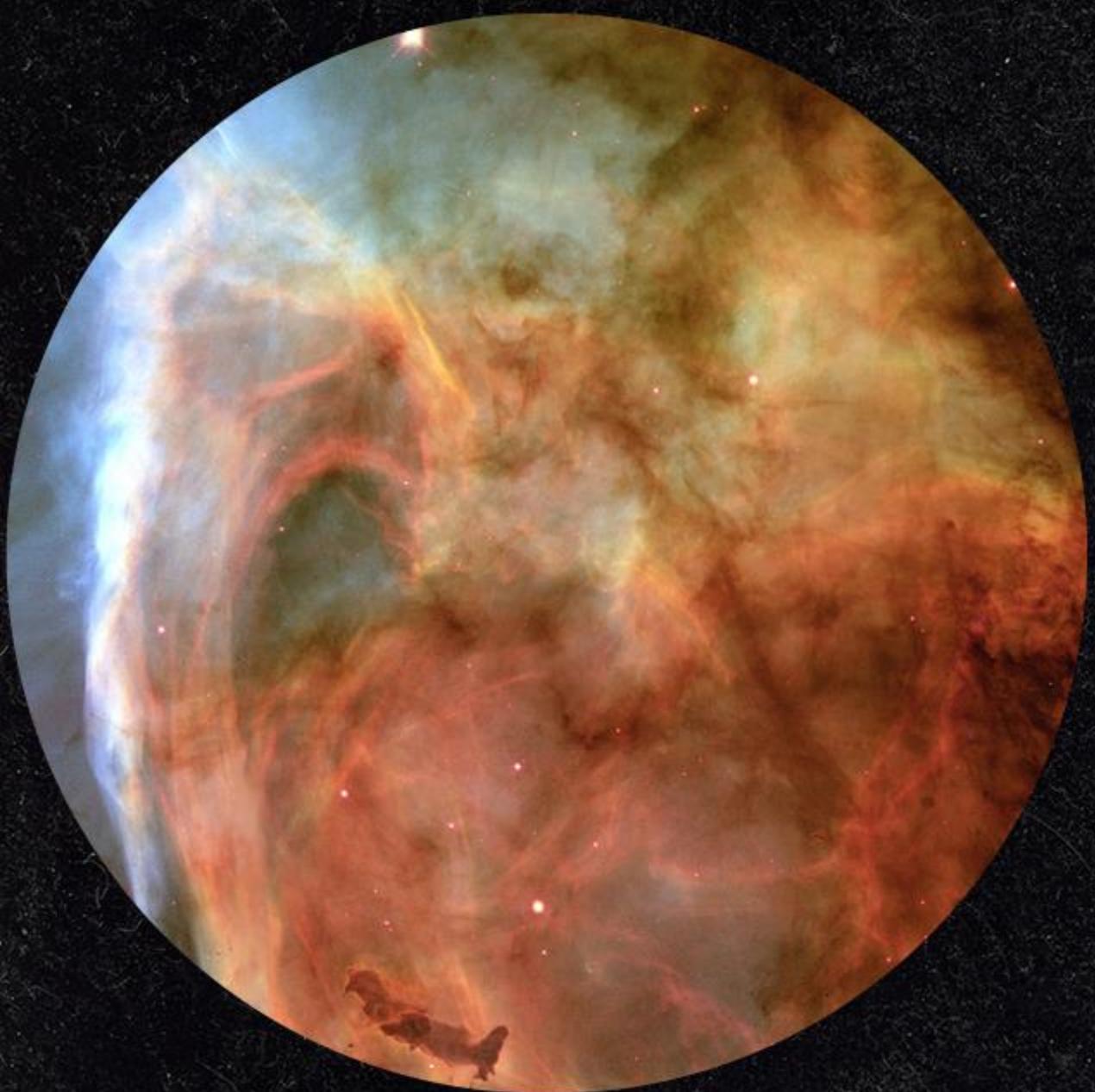
יצרתי `constructor` והכנסתי בו `super` כיוון שהוא יורש (זו קונבנצייה של ג'אווה סקריפט). יצרתי גם פונקציה בשם `foodHandler` שנובחת אל תוך הקונסולה.

השלב הבא הוא להציג את הפונקציה המאזינה, `foodHandler`, אל אירוע `food` באמצעות `on`.

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

פרק 7

יעידת שורת הטרנסיסי



יצירת שרת HTTP בסיסי

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

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

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

Class: `http.Server`

Added in: v0.1.17

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

אם נלחץ על הקישור נראה שבעצם הקלאס המרכזי שמןנו `https.server` יורש הוא `.EventEmitter`.

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

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

- `Class: http.Server`

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

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



כאמור, לא להיבהל. חמושים בידע זה, ניגש למלאכת בניהת השרת שלנו. תראו שזה יגמר בתוך
כמה דקות.

ראשית נוצר `http.Server`. איך אנו יודעים שדווקא אותו? כיון שהוא יורש מ-`net.Server` וכותב במפורש שהוא מיועד לייצרת שירותים שיודעים לקבל תנועת TCP, שזו התנועה הרגילה. איך אנו מבצעים `require`? כמו כל מודול אחר, אבל כיון שב-`http` יש כמה מודולים, אנו ניאלץ לבצע `require` מ-`http`. לא משחו מלחץ במיוחד, זה מציין במפורש בדוקומנטציה: `Class: http.Server`

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

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

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

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

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

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

const myServer = new http();

myServer.listen(3000);

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

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

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

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

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

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

const myServer = new MyServer();
```

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

טוב, האמת שזה לא כל כך פשוט – הקוד הזה הוא בן 14 שורות, אבל בשוביל ליצור אותו היינו צריכים לצלול לדוקומנטציות לא מעטות: לדוקומנטציה של `http.Server`, `ティアור.net`, `ティאורה`, `Node.js` זה שאפשר להעיר קריירה שלמה מבלוי לקרוא את הדוקומנטציה או לכתוב שורות שימושות במודולים הבסיסיים של `Node.js`, כיוון שם תctrco למש שרת לא כתבו את השורות הללו אלא תשתמשו במודול קיים `express` – מודול לבניית שרת אינטרנט שנלמד בהרחבה בהמשך הספר. מבלוי לשבור את הראש ולהיכנס לעומקם של דברים. אנו לומדים את זה כאן כדי להבין יותר לעומק איך אירועים עובדים ואיך לקרוא לדוקומנטציה.

תרגיל:

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

פתרונות:

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

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

```

        }
    }

const myServer = new MyServer();

```

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

```

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

```

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

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

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

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

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

תרגיל:

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

פתרון:

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

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

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

const myServer = new MyServer();
```

בדוק כמו בפתרון הקודם, אני יוצר קלאס שיורש את כל המתודות של httpServer. ב-constructor אני שם super, כי ג'אווהסקריפט דורשת זאת ממני. אחרי כן אני יוצר את השרת באמצעות методת listen וווצר מאיזין לאיורע request כפי שיש בדוקומנטציה – בהזנה, במקום להחזר מחרוזת טקסט פשוטה, אני מכניס קריאה ל-fs.readFile כפי שלמדנו. התוכן שהוחזר ממשם הוא התוכן שמוזרם למתודת end.

אשמור את הקוד זהה ב-app.js ואפעיל אותו באמצעות נווט בטרמינל אל מיקום הקובץ, הקלהה של node app.js וaz הקשה על אנטר. אני אוכל לראות את תוכן הקובץ.

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

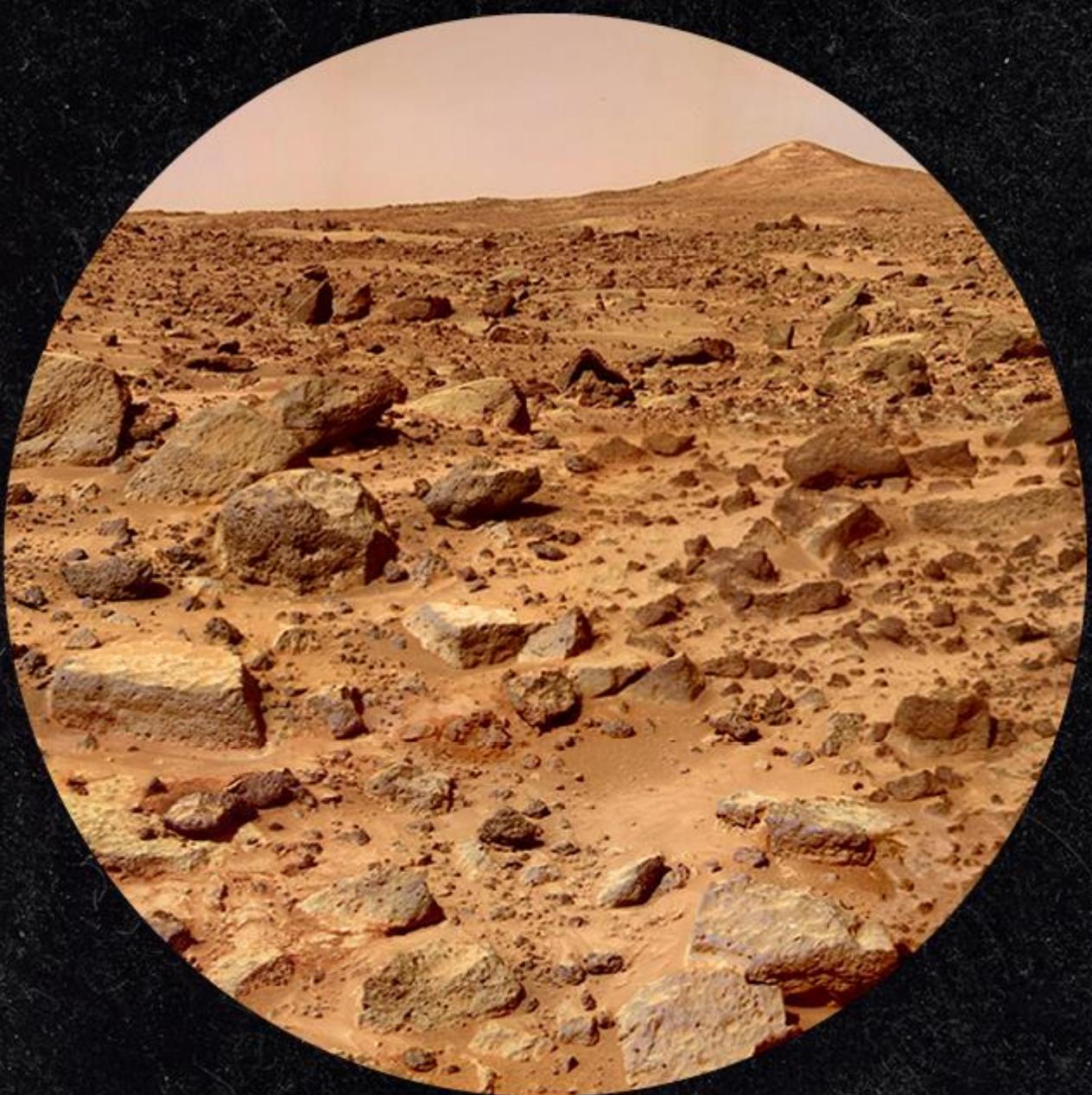
```
const HttpServer = require('http').Server;
const fs = require('fs');

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

ה-node.js של EVENT LOOP



ה-Node.js של Event Loop

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

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

מתודות הטיימרים

ב-Node.js יש לנו כמה טיימרים – פונקציות שימושיות לנו בתזמון הקוד שלנו. משתמשים בהם לכל מיני שימושים ש廣告ים בהמשך, אבל בינתיים אסביר עליהם כאן.

תrox כשאני אומר לך – `setTimeout`

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

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

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

Callback with arg 1

ת្រוץ מייד עם קולבק – **setImmediate**

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

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

למה צריך פונקציה כזו? בהמשך נבין.

הlolאה קבועה **setInterval**

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

אם תרצו את הקוד הזה, תוכלו לגלוות שהוא לא עוצר עד שתאתם לא הורגים את התהילה באמצעות **קונטROL+C**.

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

טור הקריאה

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

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

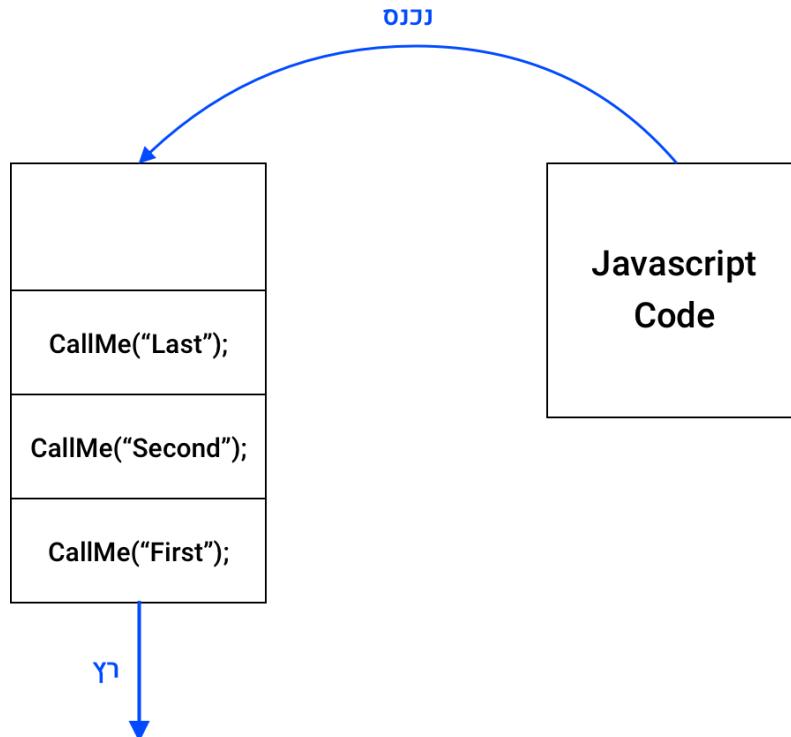
בפלט אני רואה:

```
First
Second
Last
```

מה קורה מאחורי הקלעים? איך ג'אווהסקריפט יודעת להרים את הקוד? מדובר בקוד סינכרוני, קוד שרצ לפי הסדר מלמעלה למטה והוא בולם את המשך. ג'אווהסקריפט לא תמשיך לשורה השנייה
לפניהם שהשורה הראשונה מסתירה לroz ולא תמשיך לשורה השלישית עד שהשנייה תושלם.
התור הזה נקרא "טור הקריאה" והוא יעבד גם עם פונקציות כמו:

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

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



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

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

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

כדי להמחיש את זה נבחן את הקוד הבא:

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

```
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');
```

הריצו את הקוד. הפלט שיופיע הוא:

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

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

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

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

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

יש שלב במעגל האירועים שנראה 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');

```

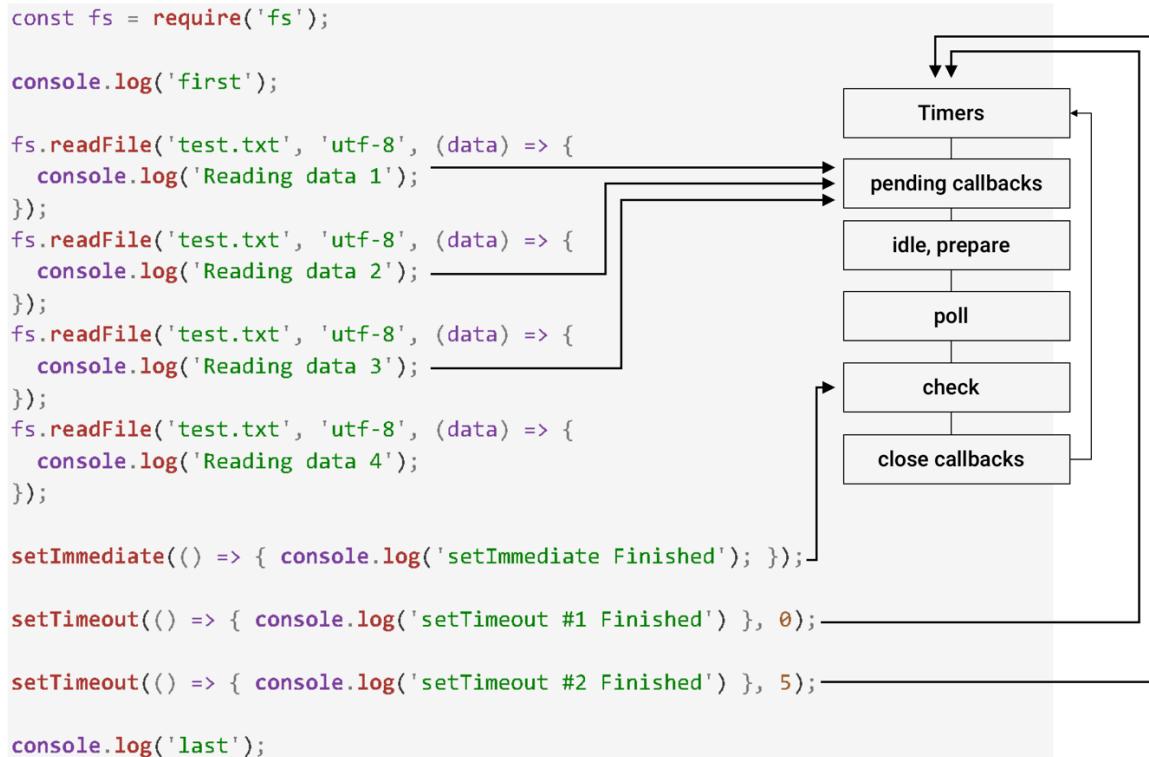
ראשית יירוץ הפלט הסינכרוני:

First
Last

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

`setTimeout #1 Finished`

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



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

`setImmediate Finished`

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

`setTimeout #2 Finished`

אם לא, הקוד יירוץ מיד אל Pending callbacks ויריץ את מה שוחר ממערכת הקבצים לפי סדר החזרה. קלומר נוכן לראותות שם:

Reading data 3
Reading data 1

אחריו שהתוර התרוקן, הלולאה ממשיכה לפעול, check ריק אז חזרים לטויימרים. אם הא-setTimeout השני עדין לא רץ, הוא יירוץ עכשו כי בטח עברו 5 מילישניות. ואז הלאה אל ה-Pending callbacks יש שם מהهو? מעולה. Node.js תריץ את הכל. אין? מצוין. חזרה לlolאה עד שכך אין שם קולבק שמחכה לחזור ואז הסקריפט ימות. העבודה הסתיימה והlolאה רצתה במלואה.

תרגיל:

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

```
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 למדי – ראשית הקוד הסינכרוני רץ, ורק אחריו כל מה שנכנס לlolאת האירועים של Node.js. במקרה זה רק setTimeout.

תרגיל:

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

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

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

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

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

פתרונות:

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

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

פרק 9

STREAMS



Streams

בפרק על ייצירת שרת http בסיסי הרأיתי את הקוד הבא:

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

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

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

const myServer = new MyServer();
```

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

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

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

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

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

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

const myServer = new MyServer();
```

שתי השורות היחידות שונות מהדוגמה ללא סטרים הן:

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

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

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

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

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

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

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

readStream.pipe(writeStream);
```

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

סוגי הסטרים השונים

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

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

נוסף על הסטריםים שכבר רأינו יש סטריםים נוספים: סטרום דו-כיווני (Duplex) לנכיתה ולקראיה כמו סוקט, שלוו נרחב בפרק על סוקטים, וסטרים שמבצע טרנספורמציה, קלומר הוא מקבל מידע, משנה אותו וublisher אותו להלה.

סטרים טרנספורמציה

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

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

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

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

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

אירועים בסטרים

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

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

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

```

const fs = require('fs');

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

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

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

readStream.pipe(writeStream);

```

הינה דוגמה קצרה יותר מעניינת:

```

const fs = require('fs');

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

let dataLength = 0;

readStream.on('data', (chunk) => {
  dataLength += chunk.length;
})

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

readStream.pipe(writeStream);

```

הכוח של סטרימים גדול מאוד, כי הוא מאפשר לי לטפל במידע בינארי או לא בינארי ממש בקלות ובלוי להתאמץ יותר מדי, מבלתי לחשב באפרים (Buffer). באפר זה בעצם "אזור הרמתנה" של הסטרים. כשהאנו יוצרים סטרים, הנתונים שלא מספיקים להיכנס אליו "מחכים" בעצם בבאפר. אבל כאמור אנו לא נדרשים להבין לעומק את עניין הבאפרים. יוצרים סטרימים ומחברים אותם עם פיפויים, ובאמצעות 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 `4-7.out` במאזעות סטרימים והכנסו אותו לקובץ בשם `app.js`. הפעילו אותו באמצעות `node ./app.js`.

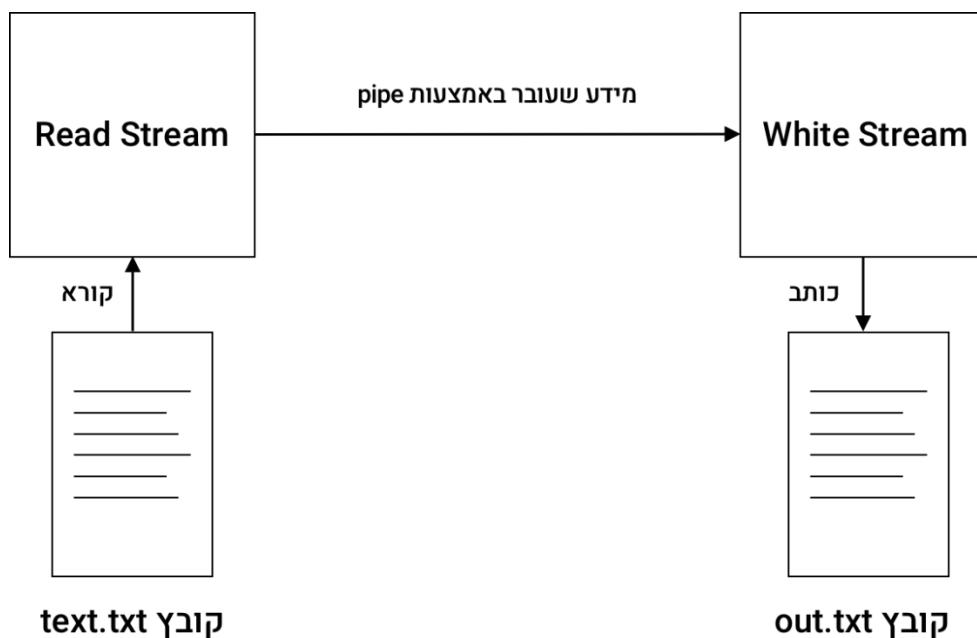
פתרונות:

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

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

readStream.pipe(writeStream);
```

הסבר: ראיית קראתי למודול `fs`. `fs`, כפי שŁmdנו בפרק ואנו רואים בדוקומנטציה, יש שתי מתודות: `createReadStream` ו-`createWriteStream`. המתודה הראשוֹנה `createReadStream` מקבלת את שם הקובץ שטמנו אנו קוראים. המתודה השנייה `createWriteStream` מקבלת את שם הקובץ שאליו אנו כתבים. כל מה שנוצר לעשות הוא לחבר את המתודות השונות באמצעות `pipe`. הסטרים הקורא מתחבר לстрים הכותב.



תרגיל:

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

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

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

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

פתרונות:

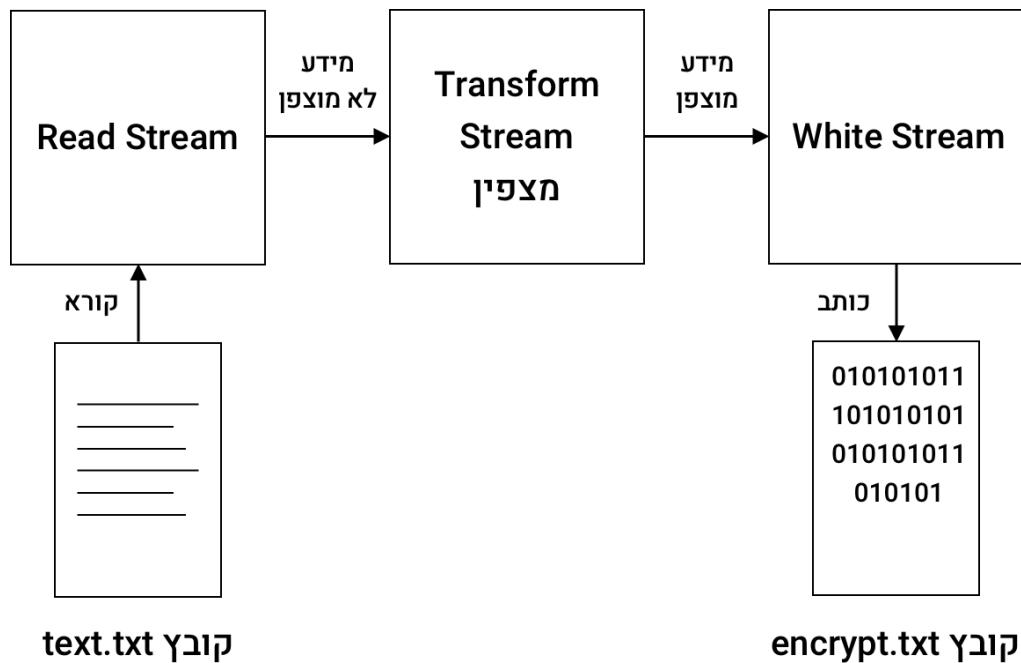
```
const fs = require('fs');
const crypto = require('crypto');

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

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

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

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



כדי לשימוש לב ש-`scryptSync` היא פונקציה סינכרונית חוסמת. אם נzieב פונקציה כזו בחיקום האמיתיים, נסתכן בבעיית מהירות ויציבות.

תרגיל:

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

פתרונות:

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

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

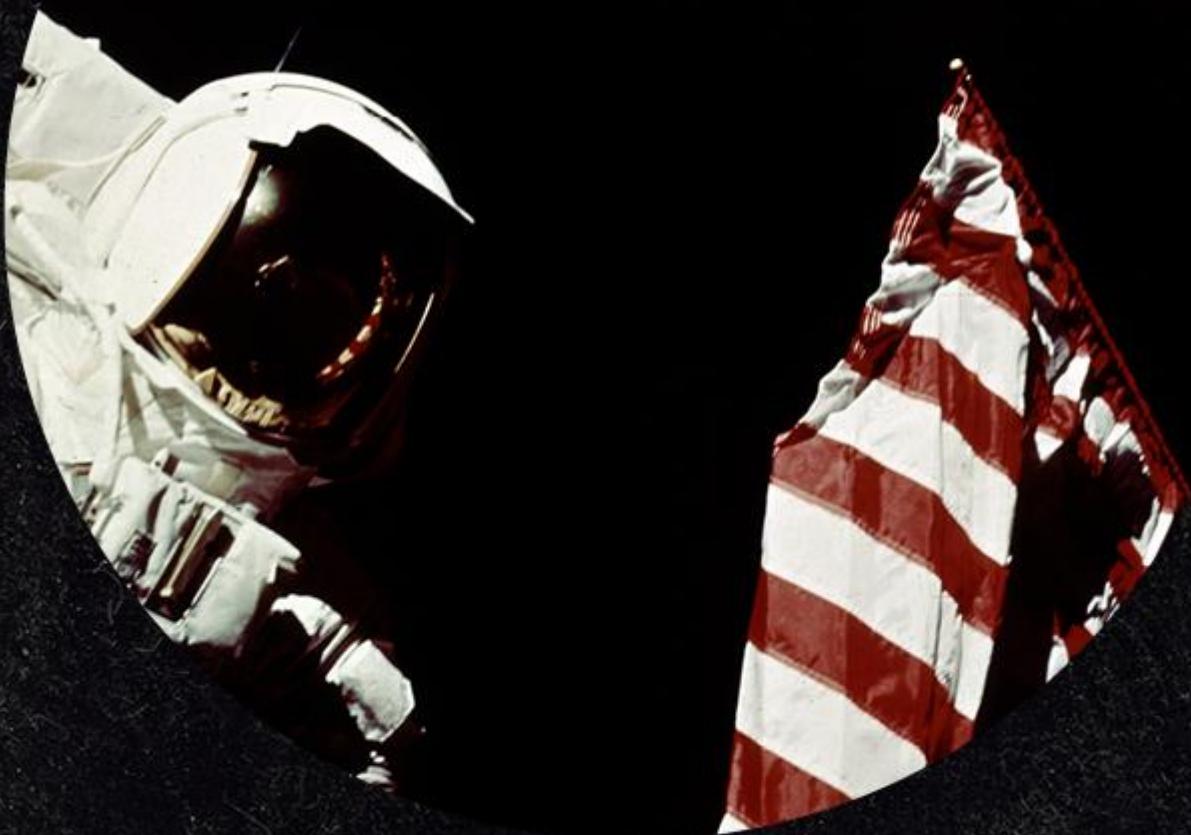
const readStream = fs.createReadStream('./encrypted.txt');
const decryptStream = crypto.createDecipheriv(algorithm, key, iv);
const writeStream = fs.createWriteStream('./decrypted.txt');

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

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

פרק 10

אריזת הבוד שלנו נמושול



אריזת הקוד שלנו כמודול

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

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

```
const myNumber = 42;
```

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

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

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

אני מיצא דברים באמצעות אובייקט מיוחד ל-`Node.js` שנקרא `module.exports`. הוא נמצא ב-`module.js` כאובייקט גלובלי ומה שנכנס אליו יוצא ב-`require`.

- אם אני אזכיר את המשתנה שלי – זה שנמצא בקובץ המודול שלי `module.js`, אני אקבל אותו בכל פעם שאני אעשה `require`. כך זה נראה:

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

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

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

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

תראו שב-`myModule` או בכל משתנה אחר שמקבל את ה-`require` מקבלו 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 `node myModule.js`, אני אוכל לראות את האובייקט הזה. אני יכול לעשות export לקלאס שלם, כמו למשל הקלאס שיצרת שרת אינטרנט, שהראיתי את הקוד שלו בפרק הקודם. אם אני אשים אותו ב-myModule.js, אני אוכל ליזא אותו בקלות כך:

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

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

    response.end('Hello World')

  }
}

module.exports = MyServer;
```

ואיך אצורך אותו? בדיק כמה משתנה, ככה:

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

myServer = new myModule();
```

אני יכול לבצע את היזירה של 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');
```

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

תרגיל:

קחו את הפתרון לתרגיל בפרק על בקשת http והפכו אותו למודול שיושב 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.
});
```

פתרונות:

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

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

וההפעלה מתבצעת 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 ויצאתי החוצה באמצעות module.exports ומInSection, כל מי שיעשה require לקובץ זהה, יקבל בחזרה את:

```
new GoogleCaller();
```

כאילו כתבתי את הכל באותם קובץ.

פרק 11

קביעות גרסאות



קביעת גרסאות

בפרק על NPM ומודולים חיצוניים הסבירתי שמדובר באחת החוזקות הגדולות של package.json. היכולת שלנו ליצור פרויקט עם "הוראות התקנה" למודולים חיצוניים היא נהדרת. כשאנו מעבירים או משתמשים בפרויקט, אנו לא צריכים להעביר גם את המודולים החיצוניים, אלו שמואחסנים ב-node_modules, אלא רק את הקובץ 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 שמדוברblkoch, לחבר נוסף בצוות או בשרת – אני לא צריך להעתיק את כל קובצי המודולים, אני רק צריך לוודא שלפרויקט שלי יש package.json בתיקיית האב שלו.blkoch יוכל לעשות npm install ותוכנת NPM תסתכם על package.json ותדע לקרוא ממנו את גרסאות המודולים והשמות שלהם ולבצע התקנה נוספת. על מנת לבדוק זאת, העתיקו את הקובץ package.json זהה אל תיקייה נטוה במחשב שלכם.

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

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

גרסאות סמנטיות

לכל תוכנה שהיא יש גרסאות – בין שמדובר במודולים של Node.js או במודול שאנו כותבים בעצמנו, או בכלל בתוכנות שלא קשורות לג'אווהסקרייפט. באקויסיטם של Node.js אנו עובדים לפי גרסאות סמנטיות. מדובר בסטנדרט של ממש שיש לו גם תרגום לעברית באתר הרשמי. אתם מוזמנים להיכנס 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, במקרה זה **4**, הוא הינו למייג'ור (Major) – גרסה ראשית. כמשמעותם גרסה זו – שוברים API. ככל מרבית מושגים מתודות או התנהלות. כמשמעותם מודול בגרסה

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

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

החלק השלישי, במקרה זהה 11, הוא הכנוי לפאטץ' (Patch) – תיקון באגים. כشمקדמים גרסה זו מתקנים באגים/בעיות אבטחה או משפצים פונקציונליות – קלומר קידום של הגרסה זו לא יעשה לנו בעיות.

בואו נדגים. נניח שיש לי מודול זהה:

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

class GoogleCaller {

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

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

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

1.0.0

אם החלטתי לתקן הערת כלשהו בקוד, למשל במקרה:

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

לכתוב:

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

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

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

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

class GoogleCaller {

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

  callGoogle() {
```

```

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

module.exports = new GoogleCaller();

```

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

```

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

```

הקוד שלו יעבד, אבל הוא יקבל התראה שהמתודה עומדת להשתנות. אני שינייתי בעצם קוד מהותי באפליקציה אבל עם泰安ות לאחרו. זהו שינוי מיינור. אני אשנה את הגרסה שלי ל-1.1.0.0. במקומם 1.0.0.

כשאני אחלייט להוריד את `callGoogle` לחלווטין, מי שנסמך על המתודה זו במודול שלי יצטרך לשנות אותו. במידה שלא, הקוד לא יעבד. ככלומר ביצעת ישינוי מהותי בלי泰安ות לאחרו. פה אני אהיה חייב לקדם את גרסת המיג'ור שלי ל-2.0.0.

אם אתם מייצרים מודול LNPM, ככה אתם-Amorim לעבד וככה המודולים האחרים עובדים. ככלומר – אם אתם משתמשים על `request` – אין לכם בעיה לעדכן פאצ'ים, אין לכם בעיה לעדכן מיינוריים – אבל כן כדאי שתבדקו היטב בדוקומנטציה של המודול ובמועד אם אתם רוצחים לעדכן מיג'ור.

קביעת גרסאות סמנטיות ב-`package.json`

וחזרה ל-`package.json`. אם ב-`package.json` יש גרסה מדוקית של המודול, ככלומר מספר לא תוספת של כובע או טilda או כל סימן אחר – כאשר אני או כל משתמש אחר עושה `npm link` למועדם, אז NPM תתקין את הגרסאות המדוקיקות ביתור.

אם יש כובע (הסימן ^ או caret באנגלית) זה אומר LNPM תתקין את כל הגרסאות שתואמות לגרסה המצוינת. ככלומר אם יהיו 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 `npm i lodash` תראה שהגרסה של lodash היא 4.17.13, היא תתקין את הגרסה الأخيرة. גם גרסה X.4.17. (X אומר כל מספר), כיון שהוא לא שובר פונקציונליות. אבל אם הגרסה الأخيرة היא X.5.X אז npm לא תתקין אותה אלא את הגרסה الأخيرة של X.4. אם יש טilda (הסימן ~ או tilde באנגלית) זה אומר ש-NPM תתקין את הגרסאות של הfatcats'ים בלבד, ולא של המינוריים. כאמור, אם יש לי פירוט גרסה כזו:

`"lodash": "~4.17.11",`

אם אני מבצע התקנה מאפס של המוצר עם `npm i lodash` היא גרסה 4.17.12 ומעלה, היא תתקין את الأخيرة, אבל היא לא תתקין את X.4.18 ובודאי לא את X.X.最新: אפשר ל לנכט על הקצה ולצין בפניהם最新 להתקין את הגרסה الأخيرة באמצעות הטקסט

`"lodash": "latest"`

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

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

תרגיל:

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

פתרונות:

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

אני מקבע את מספר הגרסה ב-`package.json` של הפרויקט שלי. אחרי שאני שומר את הקובץ אני מקליד בטרמינל, בעודי מכפיד להיות במיקום של הפרויקט שלי, את הפקודה `npm install`, ואז אנטר. NPM תבודוק 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 CLI

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

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

```
npm i eslint
```

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

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

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

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

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

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

בדוק בשביל זה יש לנו התקנה גלובלית. בהתקנה הגלובלית אנו בעצם הופכים מודול `js` שעובד ב-CLI לגלובי. ככלומר אני יכול להפעיל אותו מכל מקום בטרמינל. מכל מקום! איך עושים את זה? כתבים (בכל ספריה שהוא) בטרמינל:

```
npm i eslint -g
```

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

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

```
npm list -g
```

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

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

מצד שני זה עלול להיות בעיתי – למלא את המחשב שלנו במידע שאין לו צורך. נוסף על כך המודולים האלה לא נכנסים ל-`package.json`. לפיכך התקנה גלובלית אינה מומלצת. מה הحلופה? `npx`.

דרך נוספת לעבוד עם מודולים גלובליים היא באמצעות פקודה של NPM שנקראתakh. הפקודה הזו מופעלת באמצעותakh 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 תתקין אותו ותפעיל אותו.זה מעולה אם יש לכם כמה פרויקטים שימושיים בגרסאות שונות של `eslint` או CLI אחרים. כפי שהזכירתי קודם, מומלץ מאוד להשתמש ב-`eslint` ולא בהתקנה גלובלית.

תרגיל:

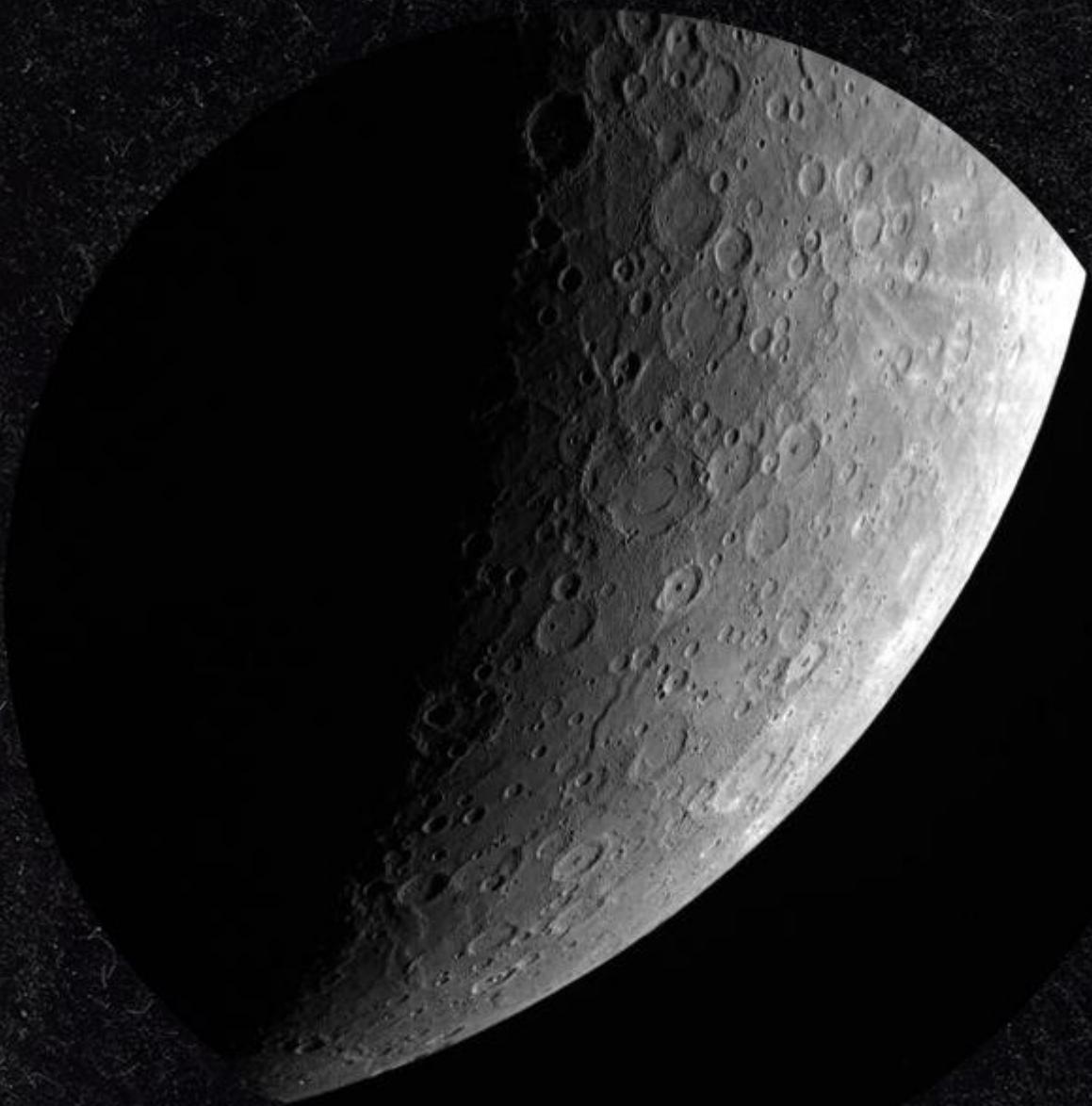
התקינו את מודול `xo` באופן גלובלי. מדובר במקרה של `eslint`. בדקו שהוא עובד על ידי הריצה של `xo` מכל תיקייה שהיא. הדוקומנטציה של המודול נמצאת פה:
<https://www.npmjs.com/package/xo>

פתרון:

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

פרק 13

כתיות און והתמונשות עם ה-CLI



כתיבת bin והتمמשקות עם ה-CLI

קל מאד לנכון CLI ב-node.js. יש לא מעט מודולים שימושיים בתהיליך ומאפשרים ליצור דיאלוגים, אнимציות וצבעים בטרמינל. אחד מהם כבר הזכרנו בפרקם הקודמים: המודול chalk שמאפשר לצבוע את הטרמינל בצבעים עלייזים.

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

```
cd ..
```

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

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

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

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

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

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

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

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

class EncryptionCLI {

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

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

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

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

יש לנו כאן קלאס. ב-`constructor` אנו מגדירים את המשתנים שנctrarך בהמשך: `key`, `iv` ו-`algorithm`. במתודה `encrypt` יש ארגומנט אחד – שם הקובץ. אנו לוקחים אותו ומשתמשים בו בקריאה ובכתיבת – בסטרים. בסוף הכתיבה נאזרן לאיירען "סוף" ונדייס `.Finished`. נבנה את הקלאס באמצעות `new` ונעביר אותו לכל מי שעושה שימוש באתם `require` `src/module.js`.

על מנת לבדוק שכל מה שכתבנו עובד, ניגש ל-`js/app`, נוצר את המודול זהה באמצעות `require` וונשותמש בו. באזזה תקינה של ה-`js/app` ניצור קובץ `test.txt` ונכנס בו כמה מילים. ב-`js/app`:
נוצר את המודול וונשותמש בו על קובץ `test.txt`:

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

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

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

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

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

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

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

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

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

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

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

למה? כיוון שמבנה הפרויקט שלי נראה כך:

```
└──bin  
└──src  
└──app.js
```

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

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

```
process.argv
```

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

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

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

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

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

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

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

באמצעות קישור סימבולי אל ספריית התקינות הגלובלית שיש במחשב שלי, בדיקת `cp -r .wkmh` g-`i` עשויה, וזאת באמצעות הפקודה:

```
npm link
```

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

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` אוסיף אותה למודול שלי ועתיק את הפונקציונליות שלו מהפרק על הסטרימים, שם היא מפורטת. הקלאס שלי ייראה如下:

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

class EncryptionCLI {

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

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

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

```

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

}

module.exports = new EncryptionCLI();

```

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

עכשו ניצור בתיקית `höö` קובץ בשם `decrypt.js`. הוא כמעט זהה לקובץ `höö`-`encrypt.js` שיצרנו בפרק:

```

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

const sourceFileName = process.argv[2];

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

EncryptionCLI.decrypt(sourceFileName);

```

השורה הראשונה היא הכרחית, היא ה-Shebang. בשורה השנייה אני צריך את המודול. שימוש לב לשתי הנקודות שיש בנתיב – כיון שאין בתיקית `höö` ואני צריך להגיע לתיקית האחות `src`, אני עולה לתיקית האב עם `..` ואז יורד לתיקית `src` ומשם לקובץ `module.js` שהוא קובץ הג'אוועסקרייפט שבו נמצא המודול שלנו.

השלב החשוב ביותר הוא לחת 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 שקיבנתי בקונסולה לבקרה ואז קריאה למתחודת `decrypt` של המודול שלי עם מה שקיבנתי מהארגומנט.

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

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

החלק הסופי והאחרון הוא הקלה

`npm link`

בתיקיה הראשית של הפרויקט. הקלה זו בעצם מאפשרת למודולים גלובליים וקשרורים לטעינה מהפעלה. זהו. אם אני אכתוב `decrypt` ובארגון השני אני אכתוב שם של קובץ שהצפנתי עם אותוCLI שכתבתי, אני אוכל לראות את הקובץ המקורי:

`decrypt .\test.txt.encrypted`

תחת השם `test.txt.encrypted.decrypted`

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

פרק 14

סוקטים - SOCKETS



סוקטים – Sockets

עד כה למדנו תקשורת עם פרוטוקול HTTP, כלומר שירות אינטרנט, זהה שמאזין לכל ניסיון תקשורת ומחזיר 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 ליבין השירות מסתיים ברגע שהользоватל נשלח חזרה אל מבקש הבקשה.

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

אבל בסופו של דבר, ב-AJAX, זו השיטה הפופולרית ביותר, השירות סטטי – אנו תמיד מאזינים לתקשורת. לעולם לא שולחים. רק הדפדפן או שירות אחר מסוגל לשלווה בקשות לתקשורת, באמצעות טעינה של הדף (הקלדה של שם המתחם או פשוט לחיצה על F5) או באמצעות שילוח בקשה AJAX. יש להזיה יתרונות, כי קל מאוד לנצל בכך כמות גדולה של משתמשים. אך יש מערכות שפטרון AJAX פשוט לא יעבוד עבורן. חשבו למשל על מערכות רפואיות או כאלה שחוובותה בהן תקשורת בזמן אמת. במערכות כאלה גם שיגור בקשה AJAX מדי שנייה עלול לא להיות מספיק.

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

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

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

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

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

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

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

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

כדי לדעת אם התוכנה הותקנה בהצלחה, הקלידו בטרמינל שלהם:

```
telnet -help
```

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

כאמור, אנו משתמשים במודול של `net` בדיק כפי שהשתמשנו במודול זהה כדי ליצור שרת HTTP. נכניס את השירות שלנו למודול שלנו. יצירת אפליקציית `Node.js` עם `package.json` עם ניצור לה תיקיות `src` ובתוך תיקיות `src` 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` שיורש מ-`net.Server`. ב-`constructor` שלו אני לא שוכח להכניס `super(this)` כפי שנדרש ממני תמיד כשהאני יורש. אני מאזין לפורט 6969 (בדיק כמו במודול `http`) ומאזין לאירוע שנקרא `connection`. האירוע הזה מופעל כאשר מתחבר לשרת שלנו, בדיק כמו האירוע המקביל ב-`http`.

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

מה שאני עושה זה לנכון לسطריםזה "ברוך הבא לשרת שלי" ולהזין לאירוע `data` שמופעלבכל פעם שmagiu איזשהו מידע מהמשתמש. זהה הדבר הייחודי שיש לנו בסוקטים – היכולת לעשות דברים כשהמשתמש מקlid. כשההמשתמש מעביר משהו, השרת מפעיל את מתודת `dataHandler` – במקרה הזה היא רק מדפסה לקונסולה.

אני מבצע `new` לクラスזה ומחזיר אותו ב-`module.exports`. זהה, יצרתי את המודולזה. בתיקייה הראשית שלuproject שלי, אני אוצר קובץ של `js/app` ובו מפעיל את המודול שיצרתי. איך? באמצעות `require`:

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

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

```
telnet 127.0.0.1 6969
```

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

הכתובת המקומית שלנו היא תמיד 127.0.0.1 – זהה כתובות שמורה המיועדת למחשב המקומי. המחשב המקומי נקרא גם `localhost` ואנו יכולים להקליד גם:
`telnet localhost 6969`
 וזה יעבד.

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

```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\barzik\node_projects> node .\app.js
Someone connected! ::ffff:127.0.0.1
Typed This

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

Welcome to my server
dddfgh

```

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

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

הו מודול מרכזי שנמצא ב-NPM. יש לו גם אתר נאה שאפשר למדוד ממנו וככל דוגמאות בכתובות <https://socket.io/> – לרוב המודולים הרצינאים 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 והסבירים שמספקים בדרך כלל ביד נדיבת ורחבת. יש כל כך הרבה מודולים مرיצים ב-`Node.js` בספר אחד או אפילו סדרת ספרים לא יכולם לכטוט אותם. במקרה, למשל, בזמן כתיבת שורות אלו, יש הסבר מקיים איך ליצור אפליקציית צ'אטים מקיפה. אני ממליץ לכם, לאחר קראת הספר, לנשות לבנות אותה.

יש מתכנתים שיכולים להעביר קריירה שלמה ב-node.js לSoloJS ולא מימוש של סוקט, אבל זה חלק חשוב מ-node.js וגם סוקטים עומדים האחורי כמו אתרים ותוכנות חשובות וכך לתרגל ולהבין זהה לאקסם – אלה סוקטים.

תרגיל:

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

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

הקיים שרת שתומך בסוקטים, יצרו setInterval שמשדרת בכל שנייה את ה-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 שירושש מ-`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 `this.connectionHandler` מתחבר לשרת (למשל עם טלנט). בכל פעם שימושו מתחבר, `connectionHandler` מופעלת.

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

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

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

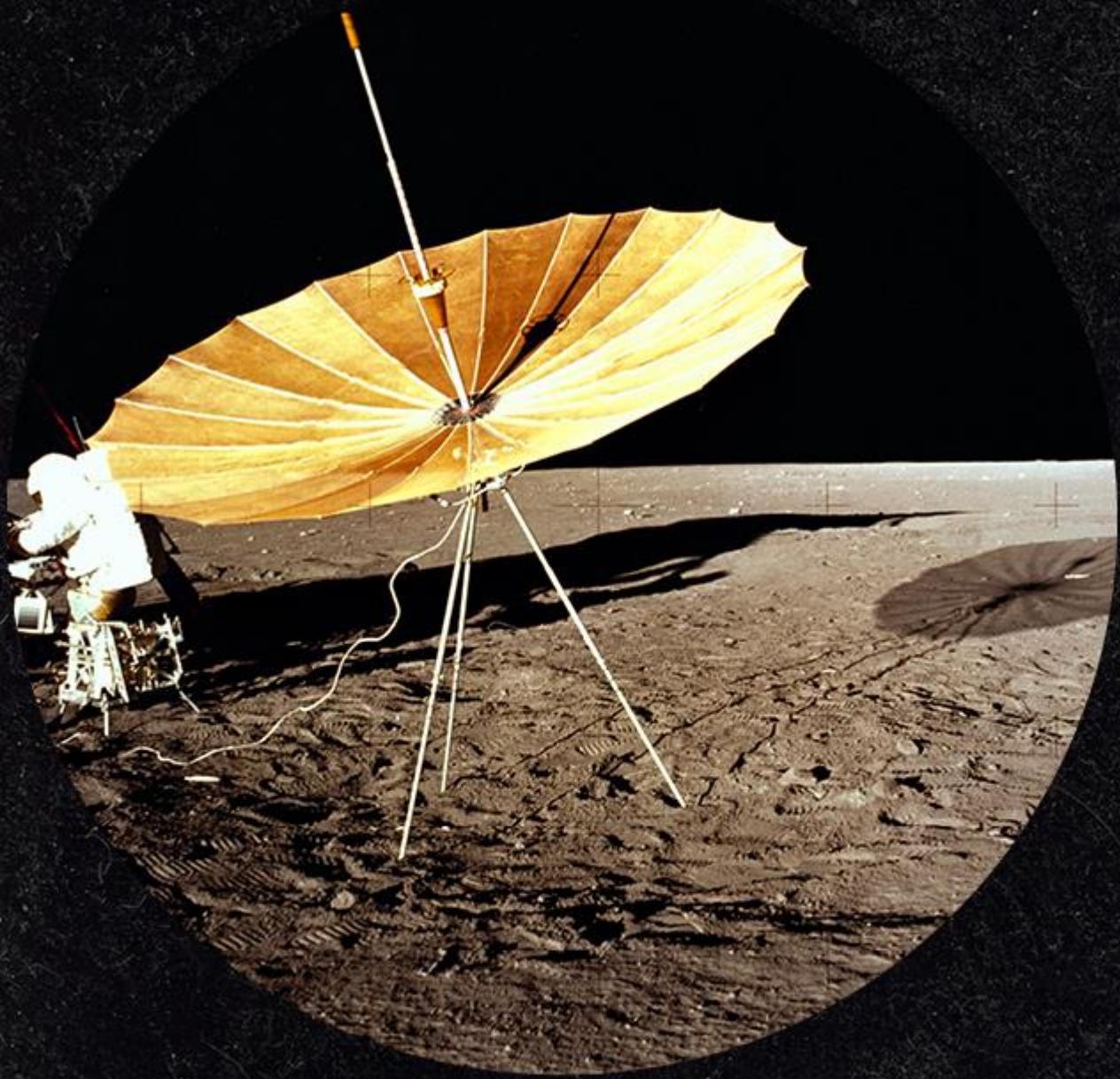
ונפעיל את השרת באמצעות `node ./app`. כדי לבדוק את השרת, נפתח טרמינל נוסף ונכתב בו:

```
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 באמצענות נזודול



קריאה משאים באמצעות מודול path

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

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

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

```
<!doctype html>
<html>

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

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

</html>
```

אני רוצה להגיע במצב שבו אני מקליד `localhost` ואני רואה את הקובץ הזה. איך עושים את זה? מרים מירום 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 גדולה: **WebServer**.

```

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

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

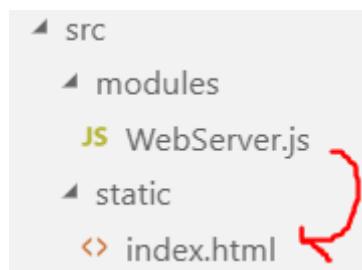
    const src = fs.createReadStream('../static/index.html'); // 
ERROR!!!
    src.pipe(response);

  }
}

exports.module = new WebServer();

```

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

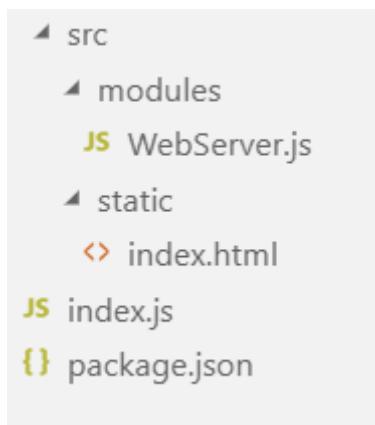


איך אני מגיע אליו? פשוט שבסוטרים! מהמקום הנוכחי שלי / . אני עולה באמצעות .. תיקיית `src` ומשם יורד אל `static/index.html`. הנתיב שאני צריך להכניס הוא: `./../static/index.html`

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

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

מבנה האפליקציה אמור להיראות כך:



נՐץ את האפליקציה כולה באמצעות `node index`. ניכנס ל-`localhost:3000` ונראה שהטרמינל מתפוץץ מהודעת שנגיאה. הודעת השגיאה טוענת ש:

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

כלומר, זה שגוי. איך זה יכול להיות?

התשובה היא `sh.js` Node מחשבת את הנתיבים מנקודת ההרצה. ככלומר, אם אני מרים את הפרויקט מהתיקייה הראשית שלו, `Node.js` חושב את הנתיב בצורה שונה לחלווטין.

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

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

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

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

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

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

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

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

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

פרק 16

PACKAGE.JSON SCRIPTS



package.json scripts

עד כה, כשרצינו להפעיל את התוכנה שלנו, נאלצנו להקליד באופן ידני `node ./app.js` או `node index.js`. כמו בדוגמה 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. מתכוון אחר שימוש בקוד שלנו לא אמרו להזכיר אותו. במקרה לשבור את הראש אם להקליד `node ./app.js` או כל entry point אחר, הוא יוכל להקליד פקודה אחרת ואני נגיד לך בדוק מה היא עשו.

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

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

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

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

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

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

סקריפטים עם שמות

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

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

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

```
npm run ahla
```

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

הסבר על הפקודה	פקודה מובנית ב-NPM
מתקינה את כל dependencies ב-package.json	install \ i
MRIIZHA AT PAKUDA TEST HEMPORUTA B-PACKAGE.JSON	test \ t
MRIIZHA AT PAKUDA START HEMPORUTA B-PACKAGE.JSON	start

משתני סביבה

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

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

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

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

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

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

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

איך אני קובע את משתנה הסביבה?

קביעת משתנה סביבה דרך הסкриיפט

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

```
set PORT=3000
```

לפני הפקודה ורשורו 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 סкриיפטים: start ו-dev. כל אחד מהם רץ עם פורט אחר. אם אקליד start ו-pm, יוכל להיכנס לשרת ה-<http://localhost:3000> רק מ-localhost:3000 ואם אקליד dev אוlocalhost:80. יוכל להיכנס לשרת ה-<http://localhost:80>.

קביעת משתנה סביבה דרך הגדרות מערכת הפעלה

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

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

חלונות – CMD

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

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

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

לינוקס/מак

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

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

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

קביעת משתנה סביבה דרך קובץ

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

```
PORT=666
NODE_ENV=development
```

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

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

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

אך מובן חשוב מאוד להקפיד שלא לשמור את `.env`. כחלק מקובצי הפרויקט שלכם! בכל האפליקציות והאתרים משנים את ההתנהוגות באמצעות משתני סביבה. משתנה הסביבה החשוב ביותר, זהה שמשתמשים בו כמעט הרבה, הוא `NODE_ENV` שי יכול 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 לך כל מיני סיבות – למשל לנו לא רוצים `console.log` בסביבה חיה, אנחנו לא רוצים שבסביבה חיה יהיו לנו קבצים לא דחוסים וכו'. אם אתם רואים `process.env.NODE_ENV` תדעו שמדובר במשתנה סביבה ותדעו גם שחשוב לקבוע אותו במחשב שלכם, אבל גם לוודאشبשת שבו התוכנה יושבת הוא `production`.
יוגדר כ-`C-production`.

dev dependencies

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

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

בדוק בשביל זה קיים **dev-dependencies**. מדובר במודולים שאנו משתמש בהם אך ורק בסביבת פיתוח. כשהאני מתקין מודול שנדרש אך ורק לסביבת פיתוח, אני אתקין אותו באמצעות הקלדה של הפלאג –save-dev. ההקלדה הזו מכניסה את המודול שהותקן ל-package.json. הנה 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`

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

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

תרגיל:

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

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

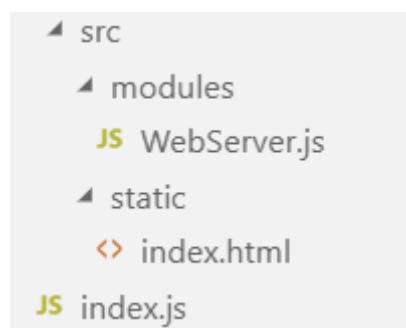
רמז: אפשר לבצע בדיקת קוד לאחר יצירה קובץ הגדרות באמצעות:

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

אם אתם זוקים לזכורת עלakh, יש לקרוא שוב את הפרק על התקנות מודולים ועל מודולים גלובליים.

פתרון:

מבנה התקינות של הפרויקט שלי נראה כך:



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

```

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

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

```

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

exports.module = new WebServer();

```

ב-`index.js` נמצא קובץ ה-`index.html` שנותען:

```

<!doctype html>
<html>

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

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

</html>

```

ואילו ב-`index.html` נמצאים הקראיה לモודול וטעינתו:

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

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

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

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

אחרי ההתקנה, ניצור קובץ הגדרות באמצעות הקלהה של:

```
npx eslint -init
```

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

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

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

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

קובץ `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",
    "eslint-plugin-promise": "^4.1.1",
    "eslint-plugin-standard": "^4.0.0"
  }
}
```

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

פרק 17

אקספלור



אקספרס

אחד המודולים החשובים ביותר ב-node.js הוא מודול אקספרס. מדובר במודול של שרת אינטרנט. זכרו את מודול `http` הטבעי של Node.js? בפרקם הקודמים למדנו איך להשתמש בו באופן בסיסי מאד, ואפשר ללא ספק לבנות שרת אינטרנט על בסיס המודול הזה. אבל האמת היא שזה קשה ודורש המון קוד, ולאורך כל הספר חזרתי והסבירתי שהכוונה של `Node.js` הוא לא במודולים הטבעיים שלו שבאים ככלי מיחל אלא בספרייה העשירה של המודולים שיש ב-node.js. אקספרס הוא בדיקת מודול זהה שמאפשר לנו להרים שרת אינטרנט בקלות יותר מכן – להוסיף לו פונקציונליות רובה ובקלות. בשורה אחת אני יכול להוסיף לאקספרס נתיבים, התנהלות, מודולי אבטחה, ניתוב, לוגים ועוד המון **פיצ'רים** שהיינו צריכים לנכון המון שורות כדי למשרתם בעצמנו.

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

ראשית, ניצור פרויקט חדש באמצעות יצרת תיינית 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? `npm i express` – נוכל לראות שאקספרס נוצר ב-`node.js` ב-`dependencies`. עכשו אפשר להשתמש בו ב-`node.js`. הדיבינו 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 שבו כבר מכירים. ראשית אנו קוראים לモודול וזו אנו יוצרים את השרת באמצעות קרייה לפונקציה הבונה במודול express ויצרים הפניה לשרת קבוע app. נודיע לשרת להאזין לפורט 3000. מהרגע זהה אנו יכולים לבצע בעצם האזנה לתנועה שMagnitude מהדפסן במתודת GET לניבים שונים.



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

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

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

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

טיפול במתודות של בקשות HTTP

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

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

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

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

The 'Response Headers' section lists the following headers:

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

יש גם מתודות אחרות שהדפדן יכול לשלוח באמצעות ג'אווהסקריפט שנמצאות לצד הלקוח.

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

כאשר אני נמצא לצד הלקוח, אני יכול לשלוח בקשות במетодות שונות אל השרת. השרת, שעליוanno אחראיהם,-Amor לדעת לטפל בבקשת האלו.

אנו יוצרים בקשות כאלה באמצעות פקודה `fetch` בג'וועהסקריפט הצד הלקוח, כולם בדף.

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

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

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

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

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

הכנסו את הקוד הבא ל-`app.js`:

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

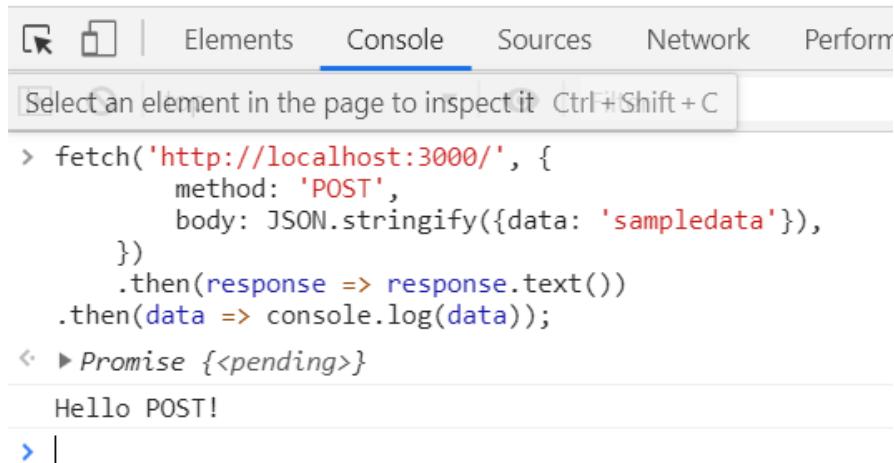
app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

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

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

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

להרוג את התהיליך שרצה בטרמינל באמצעות קונטROL + C ולפתחו אותו מחדש באמצעות node.js.app אחרת שינוי הקוד לא יעודכו ולא ירוץו). אז אפתח את localhost:3000 בדף. אדיבק את הקוד ששולח בקשה POST ואוכל לראות שבקונסולה של הדף אני מקבל Hello POST!

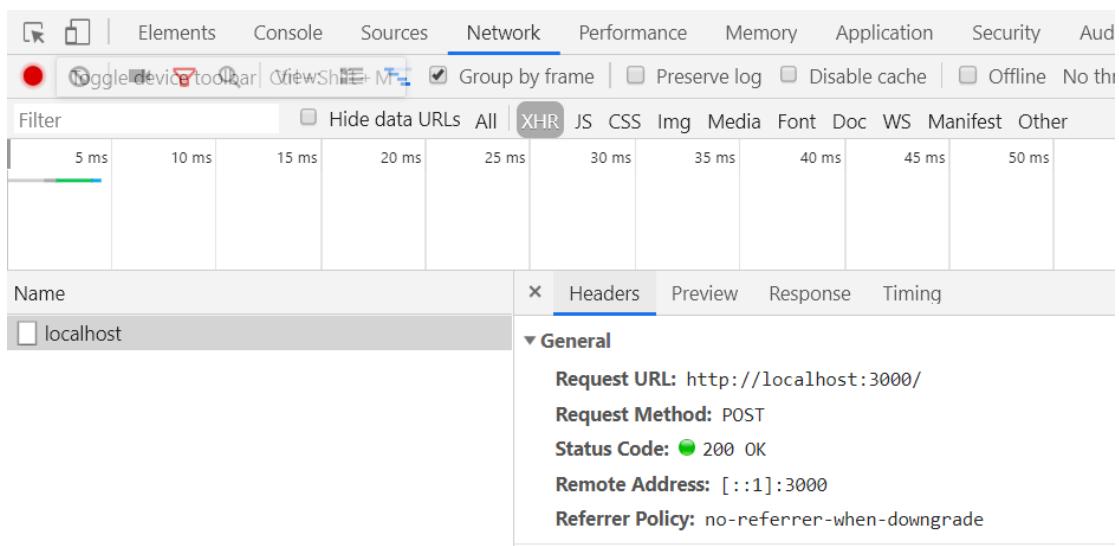


```

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.



איך ניתן את הבקשות הללו? כמו שיש לנו get כדי לטפל בבקשת GET, יש לנו post כדי לטפל בבקשת POST ובהתאם put, delete ו-patch וגם all, כדי לטפל בכל הבקשות. שימושו לב שיכ说得 במתודות של HTTPanno כתובים באותיות גדולות: למשל, GET, POST, DELETE. נshedוב על מתודות של express anno כתובים באותיות קטנות.

אנו צריכים לזכור של מרבית שמודול express עוטף את הכל – עדין מדובר במודול שהה מבכינה תפקודית ו מבחינה שימושיות לשרת ה-HTTP הבסיסי שיצרנו. כך למשל, אם אנו רוצים להחזיר דף HTML בסיסי (ולא סתם טקסט של Hello World) – אני עושה את זה בדיק כמו בשורת רגול. אם למשל ארצה לשחרר את הדוגמה שהראיתי בפרק על טיענות משאבים באמצעות path, ולשגר אל המשמש דף HTML סטטי, שנמצא בתיקייה static. אני אעשה זאת באמצעות קריאה של קובץ והזרמתו למי שנכנס לאתר. קובץ ה-`app.js` שלי ייראה כך:

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

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

app.get('/', (req, res) => {
  fs.createReadStream(path.join(__dirname,
  './src/static/index.html')).pipe(res);
});
```

אפשר לראות שאני קורא את התוצאה אל סטרום באמצעות מודול fs ומשרג אותה באמצעות `pipe` אל ה-`res` בבדיקה כמו שרת HTTP בסיסי וטבעי של Node.js. בהנחה שהקובץ index.html באמת קיים בנטייב `src/static` – אני אראה את הקובץ אם אפעיל את השרת באמצעות node `js/app`. ואכן עם הדפסן לכתובת `localhost:3000`. זה כמובן בסיסי מאוד ולא משהו שמקובל לעשות. באקספרס יש לנו מנوعי רנדורי שעלייהם נרחיב בהמשך הפרק.

ראוטינג

בתה-הפרק הקודם טיפלנו במתודות, ועכשו נטפל בנתיבים. כewish לי אתר קtan עם כמה נתיבים, למשל עמוד הבית ודף help, לא צריך לשבור את הראש. ה-.js.app שלו יוכל להכיל כמה נתיבים:

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

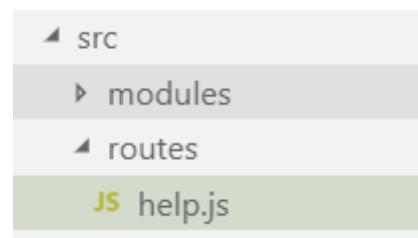
app.listen(3000, () => {
  console.log('Example app listening on port 3000!');
});

app.get('/', (req, res) => {
  fs.createReadStream(path.join(__dirname,
    './src/static/index.html')).pipe(res);
});

app.get('/help', (req, res) => {
  res.send('Help page!');
});
```

אבל מה קורה כewish לי כמה וכמה נתיבים סטטיים? 10? 100? 1,000? להכניס את הכל לתוך רשימה ארוכה ב-.js.app איננו רעיון טוב. מה עושים? בדיק בשביל זה יש לנו ראוטינג, או ניתוב בעברית, זו פשט דרך לנתח את הבקשות השונות לקבצים שונים. זה עובד בדיק כמו מודולים. אנו יוצרים מודול של ראוטינג ומיצאים אותו. מי שצורך אותו הוא האפליקציה שלנו ב-.app.js

כדי להדגים, ניצור ראוטר לדפי עזרה. מקובל לשים את כל המודולים של הראוטרים בתיקייה :routes



כأن יצרתי את `js.help`. זה מודול שבו יהיו נתיבים שונים שהיו מתחת `help`. למשל:

```
localhost:3000/help/about-me
localhost:3000/help/how-to-use
```

בקובץ הזה יהיה מודול הראوتر, שמעט הבדל קטן אחד הוא מתנהג בדיק כmo הפניות שראינו ב-`:app.js`:

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

router.get('/about-me', (req, res) => {
  res.send('About me page');
});

router.get('/how-to-use', (req, res) => {
  res.send('How to use');
});

module.exports = router;
```

מה ההבדל הקטן אך המשמעותי? במקרים של ריאוטינג אני משתמש ב:

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

ואת כל הריאוטינג אני עושה באמצעות `router.get` ולא `app.get`. בהמשך אני מיצא את המודול הזה כמו מודול רגיל.

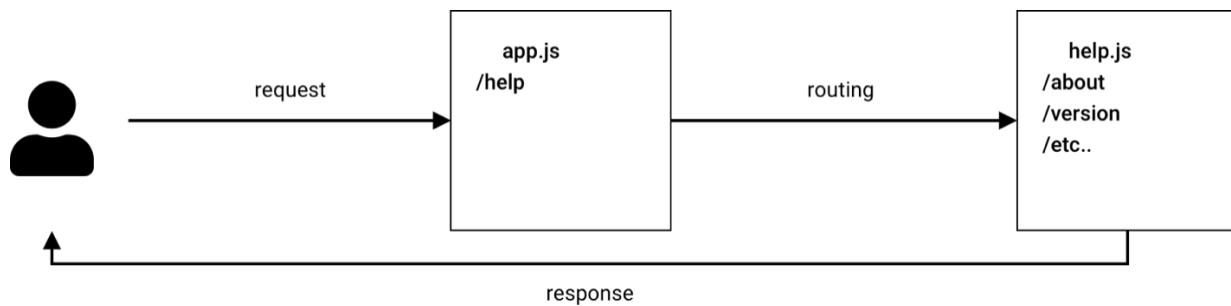
איך אני צריך אותו? באמצעות `app.use` – זהו מתודה שנותרת באופן טבעי ב-`app` של אקספרס ואפשר להשתמש בה ממש בקלות. היא בעצם מקשרת בין הקוד שרצ ב-`app` לבין הראوتر הרלוונטי, במקרה שלנו `help`.
ב-`app.use` אני מקשר בין כתובות הנתיב, שהיא טקסט פשוט, לדוגמא, לקוד שהוא משתנה שמכיל רפרנס לבין הראوتر:

```
const express = require('express');
const app = express();
const help = require('./src/routes/help');

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

app.use('/help', help);
```

בעצם ב-`app` הגדרנו את הנתיב `/help` וציינו שמי שמתפל בו הוא הראوتر. הראوتر של `help` מטפל בשני נתיבים: `about-me`, `how-to-use`, והם יהיו זמינים מיד אחרי הנתיב שהראوتر מטפל בו – במקרה זה `/help/about-me`, `/help/how-to-use` ועוד. כלומר הכתובות/`help`

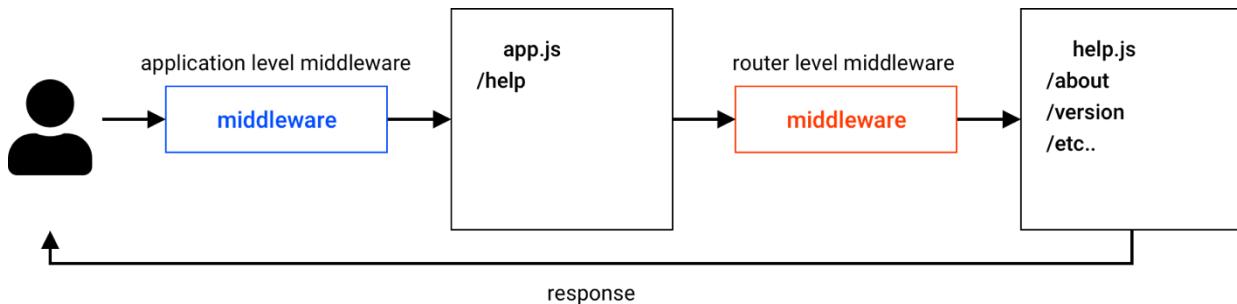


ה-`use`, שבו או משתמשים כדי לחבר את מודול הראوتر שלנו אל הנתיב, הוא מתודה שבה אנו משתמשים כדי לחבר רואוטינג אבל גם **Middleware**.

MiddleWare

אחד הkonsepций החשובות ביותר באקספרס היא עבודה עם Middlewares. MiddleWare זהה בעצם ייחית תוכנה שהבקשות והtagיות עוברות דרך ואנחנו יכולים לבצע פעולות שונות. למשל, אם יש לנו מערכותelogים, אני משתמש ב-*Middleware* על מנת לבצע רישום של בקשות במערכת. אם למשל אני מקבל מהמשמש תМОנות ואני רוצה לכווץ אותן לפני שאני כותב אותן לדיסק הקשיח – אני אעשה את זה ב-*Middleware*. העיקרון העומד מאחורי *Middleware* הוא בעצם לקבל תנועה, לטפל בה וואז להעביר אותה להמשך טיפול. מדובר בעצם בפונקציה שמקבלת שלושה ארגומנטים: (req, res, next). הארגומנט הראשון הוא הבקשה, *request*; השני הוא התגובה – *response*. כבר הסבירנו וdone באובייקטים האלה מוקדם יותר בפרק.

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



הבה נדגים באמצעות *Middleware* שביצוע לוגינג. בפרויקט שלנו ניצור תיקיה בשם *logger* ששם הוא *middlewares*, בה ניצור קובץ שנקרא *logger.js* ובו נכניס את הקוד הבא:

```
const myLogger = (req, res, next) => {
  console.log('Someone entered to the site', Date.now());
  next();
};
```

```
module.exports = myLogger;
```

שימוש לב שיש לנו כאן שלושה פרמטרים – *req*, *res* ו-*next*. אני יכול בשלב זה לקרוא את הבקשה מהлокאלי, ולכתוב לתגובה שהוא קיבל בעתיד. במקרה זה אני רק כותב ל-*console.log* ומיד אחר כך קורא ל-*next*. מה ש-*next* עשו הוא להעביר את התנועה להלאה. אם אני לא אקרא

ל-next, התנועה לא תעבור והבקשה לא תושלם, אז זה חשוב. אם יש בקשות, קריאות, כתיבה לדיסק המקומי וכו' – ה-next ייקרא מיד אחרי שהפרומים יושלמו.

איך אנו משתמשים ב-Middleware? פשוט יותר, מבצעים require לモול שלנו, במקרה הזה ל-middlewares, ומבצעים use :app.use logger.js

```
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 אצור את ה-Middleware הזה בראוטר של help שלמדנו עליון בתחילת הפרק, הוא ייראה כך:

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

router.use(logger);
```

ב-Middleware אנחנו יכולים לעשות דברים בהתאם ל-request ולשנות את ה-response. מובן שהכח הגדול של אקספרס, בדומה ל-NPM, הוא שעת רוב ה-Middleware אנו לא צריכים לכתוב בעצמנו אלא אנו יכולים להשתמש ב-Middleware שאנשים כתבו ופורסםם ב-NPM. בדיק נמו כל מודול אחר. פשוט משתמשים במודול זהה אך ורק באקספרס.

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

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

ה-Middleware זהה הוא מודול של NPM והוא מותקן בדיק נמו כל מודול:

```
npm I morgan
```

איך משתמשים בו? בדיקן כמו MiddleWare שכתבנו בעצמנו, כאשר במקרה זהה (ובכמעט בכל המקרים) יש הוראות מדוקינות להפעלה בדף המודול. במקרה של `morgan`, מפעילים אותו כך:

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

app.use(morgan('common'));
```

אם תפעילו אותו במקום המודול של `logger` באפליקציית האקספרס, תפעילו את האפליקציה ותיכנסו אל `localhost:3000`, תוכלו לראות בקונסולה את הכניסות עם מעט יותר מידע. אפשר ליצור בקלות לוגים מוחכמים, שאוגרים מידע נוסף או שכותבים לקובץ. איך? הכל בדוקומנטציה של `morgan`. הדוקומנטציה לא טוביה? ה-`Middleware` לא טוב מספיק? נחפש אחר או שנתרום לו קוד שישפר את המצב.

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

URL דינמי

モקדם יותר בפרק הסברנו על רואוטינג וראינו איך אנו מגדירים כתובות שונות לאתר באמצעות אקספרס. כתובות כאלה נקראות **כתובות סטטיות**. למה סטטיות? כי כתבונו, ממש בקוד, את הנתיבים ברואוטינג. אבל רוב האתרים מכילים כתובות דינמיות. אם יש לנו אתר שאליו מתחברים וכל אדם יש פרופיל משתמש, לא נגידיר ידנית עבור כל משתמש, רואוטינג משלו. אם יש לנו אתר שיש בו מאמרים רבים, לא נגידיר עבור כל מאמר ומאמר כתובות ברואוטינג. בדיק בSAMPLE זה יש לנו כתובות דינמית, כתובות שבה יש פרמטר מסוים שמשתנה כל הזמן ואקספרס יכול לקרוא אותו ולהציג מיד תוכאה שונה לכל משתמש ומשתמש.

הבה נדגים באמצעות רואוטינג של `user/NAME`. ה-`NAME` הוא בעצם משתנה דינמי, כלומר אם אני נכנס לאתר בכתובת: <http://localhost:3000/user/moshe> אני אזכה לראות התוצאות לפרטר `moshe`.

הדבר הראשון שנចטרך לעשות הוא להגדיר ריאוטינג שיטף בבקשת שומות לנתיב `user`.
בתיקיית `routes` בפרויקט האקספרס שלנו, ניצור קובץ שוייקרא `user.js` ונותר אותו ריק. נקבע
לו בקובץ `app.js` שמוביל את האפליקציה שלנו באופן זהה:

```
const express = require('express');
const app = express();
const fs = require('fs');
const user = require('./src/routes/user');

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

app.use('/user', user);

app.use(express.static('public'))

app.get('/', (req, res) => {
  fs.createReadStream('./src/static/index.html').pipe(res)
});

});
```

כל הקוד שלעיל אמור להיות לכם מוכר מאוד כיוון שעברנו עליו מוקדם יותר בפרק. עכשו ניגש
למודול הריאוטינג של `user` ונגידו את ה-URL הדינמי. זה הרבה יותר קל מהה שוחשבים. כאשר
אנו רוצים נתיב דינמי, אנחנו צריכים פשוט להוסיף נקודותים בנתיב, לפני שם הפורט. למשל,
אם החלטתי שאני רוצה פורטר שבו יהיה `name`, כך אני מקבל אותו:

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

router.get('/:name', (req, res) => {
  res.send(`User ${req.params.name} entered the system`);
});

module.exports = router;
```

הקוד הזה יהיה בראوتر של ה-`user`. קלומר ב-`./src/routes/user`. אני ניגש אל הפורט בכל
מקום באמצעות `req.params.name`.

`req` כאמור הוא אובייקט הבקשה שאנו מקבלים ב-`Middleware`. אחת התכונות שלו היא
`params`, אובייקט נוסף שלו ממוינים כל הפורטרים כשייש בבקשתה. קלומר אם יש פורטרים

של POST או GET – הם יהיו שם. את זה לא נראה במודול http קלאסי אלא זו פונקציונליות שיש באקספרס, אחת מרבות שעצרו למודול זהה להפוך לאחד הפופולריים ביותר.

אני יכול לקרוא לפורמטר שלי בכל שם, למשל השם המופרך ahla גם הולך – השם עצמו לא משנה כלל, הוא רק קובע איך אני אקבל אותו ב-params:
req.params

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

router.get('/:ahla', (req, res) => {
  res.send(`User ${req.params.ahla} entered the system`);
});

module.exports = router;
```

אם אני אכנס אל:

<http://localhost:3000/user/moshe>

אני אוכל לראות את הכתוב:

User moshe entered the system

אני יכול להשתמש בכמה פרמטרים בנתיב ללא הגבלה, למשל:

```
router.get('/:name/:id', (req, res) => {
  res.send(`User ${req.params.name} entered the system. The ID is
${req.params.id}`);
});
```

כאן יש לי שימוש בשני פרמטרים: id ו-name. אם אני אכנס אל:

<http://localhost:3000/user/moshe/1>

אני רואה את הכתוב המתאים.

שימוש לב שם אני מגדיר נתיב דינמי – אני חייב להכניס את כל המספרים. אני לא חייב להשתמש רק ב-router.get אלא יכול להשתמש גם ב-post או בכל מתודה אחרת.

tabniot

עד עכשו החזרנו שני סוגי תגיות – דף HTML סטטי, שלו קראנו באמצעות `fs.read` או תגובה פשוטה באמצעות `res.send`. אבל בחים האמיתיים פעמים רבות אנו צריכים להשתמש בתבניות – ככלומר שילוב של דף סטטי עם נתונים דינמיים. בדוק בשביב זה אנו צריכים להשתמש בתבניות. יש כמה סוגים של מבנים שאקספרס תומך בהם ואנו נלמד כאן על `ejs`, הסוג הפופולרי ביותר לאתרים סטטיים פשוטים. אנו נלמד עליו בעת כדי להציג איך מנוע רנדום עובד, למרות שבעולם האמיתיסביר להניח שתעבדו עם ריאקט, אנגולר או שע.

`ejs` לא בא כברירת מחדל עם אקספרס ויש להתקינו באמצעות:

```
npm install ejs
```

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

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

מהנקודה הזו אני יכולם להשתמש במנוע התבניות. ניצור תיקיה מהנתיב הראשי שנΚראת `views`. השם הוא ברירת מחדל והוא מכיל את כל התבניות. בהה ניצור תבנית בשם `index.ejs` ומכניס בה HTML רגיל וסטנדרטי:

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

עכשו אנו יכולים להשתמש בתבנית זו בדוק כמו בקובץ סטטי. הדרך להשתמש בה היא פשוט להשתמש בMETHOD `render` שנמצאת ב-`result` ולהכניס את שם התבנית, במקרה שלנו `index`. אם למשל אני רוצה לקרוא למבנה הזה מהנתיב המרכזי והראשי, `-ejs.app` אני אכתוב (עוד לפני הרואTING):

```
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 לדוחף פנימה משתנים, אלו משתנים שבא לי – בתור הארגומנט השני של מתודת `render`. למשל, בואו נכניס `subtitle`. אני אצור אובייקט עם `subtitle` ופושט עביר אותו כארגומנט השני, ממש ככה:

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

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

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

אם אני אכנס אל:

<http://localhost:3000/>

אוכל לראות את הטקסט המלא כפי שהכנסתי אותו. כאמור, זה יכול להיות משהו נחמד יותר. אנו יכולים להכניס את הפרמטרים האלה מכל מקום, למשל מתוך URL דינמי:

```
router.get('/:name/', (req, res) => {
  res.render('index', {name: req.params.name});
});
```

וההדפסה תהיה מראה כזה:

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

אני גם יכול להכניס מערכים – מה שיש ב-*אזע* 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});
});
```

והמבנה תיראה כך:

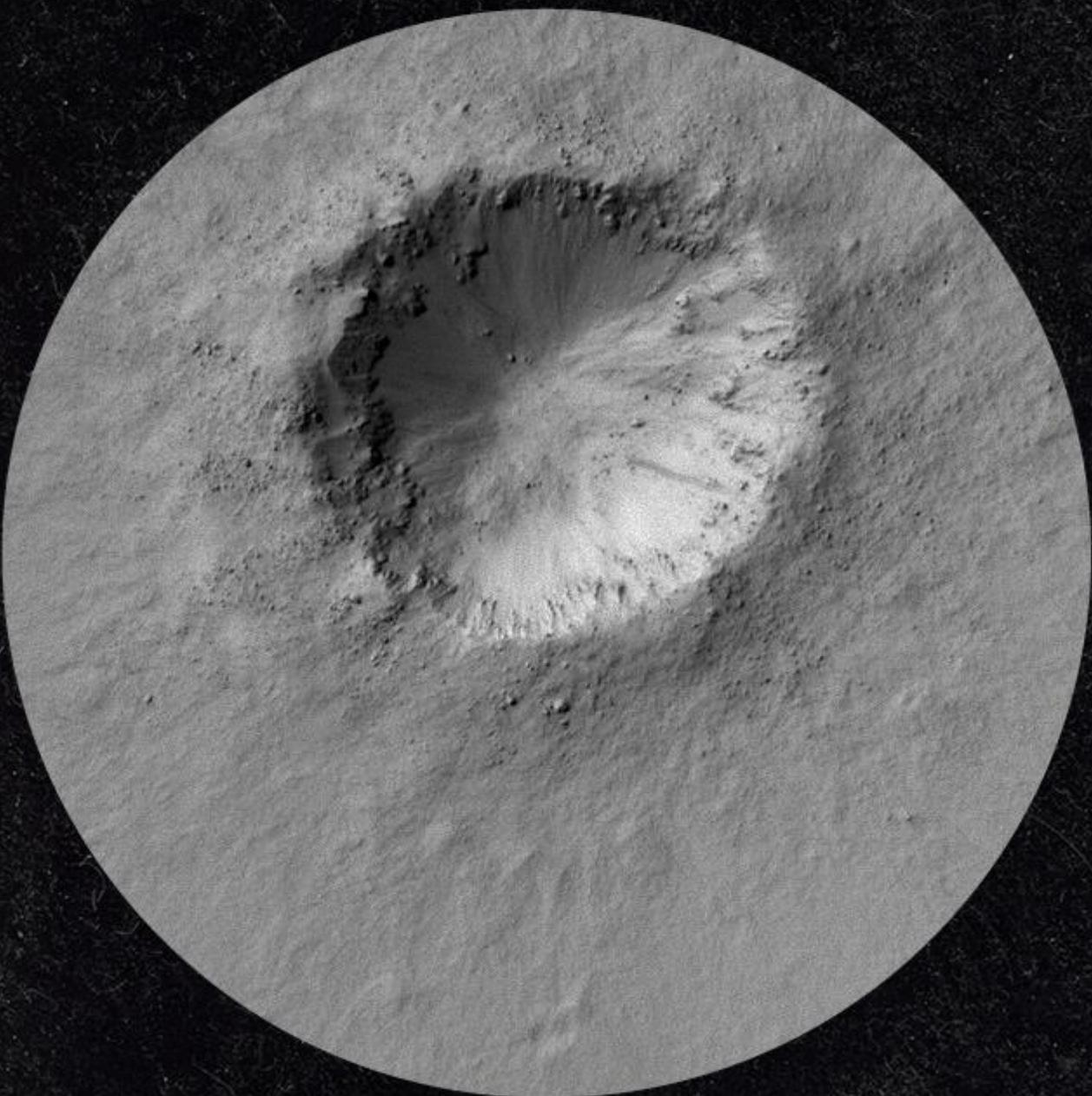
```
<html>
  <h1>Hello <%=params.name%></h1>
</html>
```

בתבניות *אזע* יש המון אפשרויות נוספות, כמו תת-tabיות, לולאות ותוספות נוספות, אבל הבסיס מספיק ברוב האפליקציות. ברוב המקרים אנו משתמשים בספרייה צד שלישי כמו ריאקט על מנת לבנות את התבנית, בעוד בצד השרת אנו משתמשים באקספרס כדי לבנות את ה-API – ככלומר את הממשק *AJAX* פונה אליו ואת התבנית הבסיסית שטוענת את ספריית הריאקט.

אפליקציה אמיתית באקספרס תכיל רואטרים שיגדרו את הנטיבים, או endpoint בrama מקצועית שקיימות *AJAX* יוכל לפנות אליה. באמצעות *Middlewares* יהיה בקרה אבטחתית, זיהוי ואוטנטיקציה. במקרים מסוימים יהיה המידע שיועבר לרואטינגן. מי שיקבל את המידע זה צד הלוקוח, שמציג אותו באמצעות ריאקט, אנגולר או שע, וביהם ספר זה אינו עוסק.

פרק 18

חיבור ל-MYSQL



חיבור ל-MySQL

תוכנה אינה רק קוד אלא גם מידע. מידע מאוחסן במסדי נתונים. יש מגוון גדול מאוד של מסדי נתונים מסוגים שונים שלכל אחד יש יתרונות (וחסרונות). מסד נתונים מסוג MySQL הוא אחד מסדי הנתונים הנפוצים בעולם. ספר זה אינו מלמד מסדי נתונים והפרק הזה מ寧ה הבנה בשאלות בסיסיות ב-MySQL. במידה שאין לכם ידע בנושא, אפשר לדלג על הפרק הזה ולהמשיך לפרק הבא.

כמו בכל דבר, יש לא מעט מודולים שמתחרבים ל-MySQL. אנו נבחר במודול הפופולרי ביותר mysql ונתקין אותו באמצעות npm. למודול זה יש דוקומנטציה נרחבת מאוד וברורה למדי. אנו עוברים על הבסיס פה, רק כדי לראות עד כמה זה קל. אתם מוזמנים לקרוא את הדוקומנטציה בעצמכם פה: <https://www.npmjs.com/package/mysql>

אנו נציג בשרת MySQL שעבוד. אני שוב יוצא מנקודת הנחה שאתם מכירים מספיק MySQL כדי להרים שרת זה בעצמכם. אפשר להרים MySQL על כל מערכת הפעלה: חלונות, מק או לינוקס. ההתקנה היא קלה וההרצה עוד יותר. אם יש לכם מסד נתונים שרצ במחשב, סביר להניח שהוא נמצאlocalhost בפורט 3306, אבל מסד נתונים יכול להיות בכל מקום, כמובן.

פרק זה אני משתמש בדוגמה במסד נתונים שנמצא על המחשב שלי, localhost בשם זה הוא test שמתחרבים אליו עם שם משתמש root וסיסמה שהוא 123456. הטבלה היא clients ויש לה שלושה שדות: id (המפתח הראשי), name ו-city.

מבנה שבאתרי אמת לעולם אל תשתחשו 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 – כולם היא תבלום את כל הפעולות שיבאו אחריה. זו אחת הסיבות שחשוב לבצע את החיבור פעם אחת ולשמור את הרפרנס זהה במקומם שקל להגיע אליו. החיבור הוא פשוט ובעצמאות נדרש לספק את השירות שעליו מסד הנתונים, שם מסד הנתונים והסיסמה 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 הזה יש לנו חיבור. אם נריץ את הקוד הזה (למשל נבדיק אותו ב-`node ./app` ונריץ את הקוד באמצעות `node ./app`) נראה שהתהליך נותר פתוח. הרצת הקוד לא מסתיימת כיוןSCP שכל עוד החיבור פתוח, הקוד רץ.

חשוב מאד לציין שהקוד המובא כאן הוא רק לצורך הדוגמה. לעולם לעולם אל לנו להכניס סיסמות בקוד המקורי של האפליקציה שלנו. זוכרים את משתני הסביבה שעלייהם למדנו בפרק קודם יותר? זהו מקום מצוין להשתמש בהם. למשל, ליצור משתני סביבה בנוסח הזה:

```
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 ייסגר והקוד יפסיק לרוֹץ.

בדרכ נכל לנו לא יוזמים חיבור ישיר אלא מבצעים pooling, כלומר יוצרים כמה חיבורים, שומרם אוטם בצד ומשתמשים בהם לפי הצורך. מודול MySQL מאפשר לנו לעשות את זה בקלות ובדרך דומה מאוד לחבר ישיר באמצעות Method `createPool` או `createPool`. מוגדר זו זהה אחת לאחת למתודות `createConnection` אלא שבניגוד אליה, אנו צריכים לומר כמה חיבורים צריך ליצור בצד. כך זה נראה:

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

אנו לא צריכים לבצע `connection`.`pool` מהנקודה הזו, 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 Method `get` ייועודית.

הערה חשובה: אם אתם מקבלים את השגיאה זו:

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

הריינו את השאלה הבאה בשורת ה-MySQL:

```
ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password  
BY 'password';
```

כשה-word password היא הסיסמה שלכם.

שאילתת בסיסית

אחרי שיש לנו חיבור, אנו יכולים להתחיל לעבוד. נתחיל עם שאלה בסיסית של SELECT. אפשר לבצע שאלות ישרות מ-pool או מ-connect query. השאלה הזו מקבלת שני ארגומנטים. הארגומנט הראשון הוא השאלה עצמה, הארגומנט השני הוא קולבק שיש בו שלושה ארגומנטים: שגיאה, תוצאה (ש망יעה כאובייקט) ומידע על השדות. ככה זה נראה:

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

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

pool.query('SELECT * FROM `clients`', (error, results, fields) => {
  // error will be an Error if one occurred during the query
  if (error) throw error;
  // results will contain the results of the query
  console.log(results);
  // fields will contain information about the returned results
  fields (if any)
  console.log(fields);
});
```

אנו יכולים להכניס כל שאלתה שהיא אל מתודת `query` ותמיד נקבל תוצאה או שגיאה. זה עד כדי כך קל. גם `INSERT` עובד או כל שאלתה אחרת. כן, כולל `DROP`. מודול MySQL של `Node.js` הוא בסופו של יומ גשר – גשר בין הקוד שלנו למסד הנתונים. כל מה שצריך לעשות הוא להחליט איזו שאלתה אנו שואלים.

המרת הקוד לעובדה עם פרומיסים ולא עם קולבקים

אפשר לראותות כמה זה קל. אבל יש עם זה בעיה: הקוד עובד עם קולבקים ולא עם פרומיסים. פתרון הבעיה הזה הוא קל יחסית, כיון שאפשר להמיר כל קולבק לפרומיס באמצעות:

```
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);
});
```

זהרי טכניקה שמתכונתי ג'אווה סקריפט אמורים לשנות בה. אנו יוצרים פונקציה ושם אנו מוחזרים פרומיס. בתוך הפרוםיס אנו מבצעים את הקריאה האסינכרונית עם הקולבק ובתוך הקולבק מבצעים resolve או reject. עכשו אפשר לקרוא לפונקציה שיצרנו ולקבל פרוםיס 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 פרוםיס

קריאה רגילה לפונקציה שמחזירה פרוםיס

אם חיבטים להעביר
את החיבור

از בונגע לקולבקים, אין בעיה. פשוט צריך לזכור שבחיהם האמיתיים ולא בספר או בדוגמאות. נעתוף תמיד את הקוד של הקולבקים בקוד שמחזיר פרוםיס.

Prepared Statement

בעיה גדולה יותר עם כתיבת Queries ישירות באופן זהה היא שמדובר בפתח לצורות אבטחה גדולות. אנו תמיד חייבים להשתמש ב-prepared statement כדי להימנע מ-SQL injections. קוד ללא prepared statement הוא קוד סופר בעייתי. ברגע שモמחה 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 ואנו יוצאים אליה את הנתונים. זה נראה מסובך אבל המימוש של זה הוא פשוט. כותבים את ה-Query כרגיל אבל במקום הנתונים מציבים סימן שאלה [?] וublisherים את הארגומנטים שמחליפים את סימן השאלה לפי הסדר:

```
pool.query('SELECT * FROM `clients` WHERE id = ? ', [1], (error,
results) => {
  if (error) reject(error);
  console.log(results);
});
```

כאן יש לנו נתון אחד, ה-`id`. אנו מחליפים אותו בסימן שאלה וublisherים את הערך שלו במערך. במקרה הזה יש לנו רק נתון אחד. אם יש לנו כמה, אין בעיה – רק צריך להקפיד על הסדר במערך:

```
pool.query('SELECT * FROM `clients` WHERE name = ? AND city = ?',
['Moshe', 'Petah Tiqwa'], (error, results) => {
  if (error) reject(error);
  console.log(results);
});
```

סימני השאלה יכולים לבלב, אבל זה באמת פשוט – בשאלתה שלכם אסור שייהיו נתונים שמאגים מבחן. אנו מציבים סימן שאלה בכל נתון כזה וublisherים את הנתונים מבחן במערך לפי הסדר של סימני השאלה. זה נעשה כי ברוב המקרים הנתונים לשאלות מגיעים מקלט של המשתמש ואנחנו ממש לא רוצים לקבל פה injection:

```
pool.query('SELECT * FROM `clients` WHERE name = ? AND city = ?',
  ['Moshe', 'Petah Tiqwa'], (error, results) => {
  if (error) reject(error);
  console.log(results);
});
```

אפשר לראות כמה קל להשתמש ב-MySQL יחד עם Node.js באמצעות מודול ייחודי, ומודולים כאלה קיימים לכל מסד נתונים שיש. זו הגדולה והכוה של Node.js.

פרק 19

עליה לפורודקהשו



עליה לפודקשן

למدىנו איך מרים מירויים שרת HTTP, ללא אקספרס ועם אקספרס, איך מרים מירויים שרת שתומך בסוקט, ונשאלת השאלה – איך מעלים הכל לרשות? אם יצרתי אתר או API או שירות כלשהו מבוסס Node.js, איך אני מעלה אותו לרשות? איך אני מאפשר לאנשים אחרים להשתמש בו? איך אני עולה לסייע אמיתית? זה נקרא עליה לפודקשן – סביבת ייצור אמיתית.

יש כמה דרכים להעלות קוד של Node.js. הראשונה היא לשכור שירות אמיתי, Private Server או מכונה וירטואלית על שירות – Virtual Private Server, ולהפעיל את הקוד ממש בדיקון כמו שהפעלנו אותו מהמחשב בזמן הלימוד. צריך לזכור שבסוףו של דבר שירות זה מחשב, ויש לו מערכת הפעלה (חלונות או לינוקס). לעיתים אין מערכת הפעלה גרפית ומה שיש לנו זה טרמינל בלבד. אנו מתחברים מרוחק לטרמינל ומפעילים את האתר, את האפליקציה או את השירות מבוסס ה-Node.js באופן דומה למחשב שלנו.

דרך שנייה היא באמצעות שירותי ענן. יש לא מעט שירותים ענן בעולם: AMAZON, AZ'OR והרוכקו 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 שירותים ענן כאלה ולכל שירות ענן זה יש מדריך מסוים משלו המסביר איך לפרש עליו קוד Node.js. בפרק הזה נראה את שתי השיטות.

עליה לפודקשן עם שירות

אם אתם מתכנתים PHP או מתוכנתים בשפות אחרות, העליה לפודקשן היא פשוטה – להעלות קבצים אל השירות, וה-Apache או ה-XhNg יודעים לטפל בהזה. אבל ב-Node.js אנו לא יכולים לעשות את זה, אנחנו בעצם צריכים את השירות. אנחנו צריכים לדאוג שהוא כל הזמן יהיה באוויר. אם תרגלتم כמו שצריך, ראייתם שאם אתם לא מרכיבים את תחילה Node.js שארחrai להרים את השירות שלכם – השירות שלכם לא יעבד. אם יש תקלת – התהיליך יקרוס וצטרכו להרים אותו מחדש, או אם אתם עושים ריסט למחשב, צטרכו שוב להריץ אותו מחדש. זה בסדר נאש לומדים ומתרגלים, אבל פחות כאשר מדובר באתר שמשרת ללקוחות. אנחנו צריכים לדאוג לכך שהטהיליך יירץ כל הזמן. זהו האתגר האמיתי כאשר אנו פורשים קוד של Node.js בשרת שלנו.

גם כאשר השרת עבר ריסטרט וגם כאשר יש לנו תקלת מסויימת – התהילך תמיד צריך להיות למעלה. תהשבו על גמד קטן שבכל פעם מרים `node ./app.js`. node כאשר האפליקציה נופלת. לगמך הקטן הזה יש שם: `2mk`. נחשו מה? זה מודול של `Node.js` שדואג שהקוד שלנו יהיה כל הזמן למעלה.

מתקינים אותו על השרת באמצעות התקינה גלובלית של מודול.

`npm pm -g`

אחרי התקינה, נקליד:

`pm2 start app.js`

האפליקציה תתרום לאויר. אם תעשו ריסטרט למוגנה או תסגורו את הטרמינל, השרת עדיין יהיה באויר. אפשר להריץ כמה וכמה שירותים ולראות אותם באמצעות:

`pm2 list`

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

`pm2 start app.js -watch`

ומשנו את הקוד שלכם, האפליקציה תיתען מחדש.
אם אתם רוצים לעזור סופית את האפליקציה פשוט תקלידו:

`pm2 stop app.js`

הდוקומנטציה העשירה של `2mk` מסבירה היטב איך לעשות את כל הפעולות האלה. אם עברתם על כל פרקי הספר, לא צריכה להיות לכם בעיה להסתדר עם הדוקומנטציה הזאת. אפשר להגדיר שם כמה סביבות לכל אפליקציה, להגדיר משתיי סביבה שונות ועוד דברים מעניינים. אם אתם רוצים להרים את אפליקציית `Node.js` שלכם על שרת משלכם, `2mk` זו הדרך.

עליה לפרויקטן בענן

לכל סביבת ענן יש הדרכים שלה להעלות אפליקציית Node.js – בהתאם לצרכים. בחברות גדולות ובעלי קהנות יש אדם שתפקידו הוא להעביר את הקוד לסביבת הענן. התפקיד של האדם זהה נקרא DevOps וזהו הלוחם של שתי מילימ: Operations-Developer. 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 ידידותיות למפתחים בודדים. הפופולרית שבינן היא Heroku שמספקת גם סביבה חינמית למפתחים בודדים. נלמד עליה בפרק זה.

על מנת לעבוד עם Heroku, יש ליצור שם חדש ולהתקין על המחשב שלכם את Heroku toolbelt. מדובר ב-CLI פשוט שמאפשר לכם להקליד בטרמינל את הפקודה Heroku ובעמצעותה לעשות פעולות שונות. מיד לאחר ההתקנה נקליד login heroku. נקליד את הסיסמה ונוצר מפתח התחברות. בעצם זה לחוץ על Yes, Yes, Yes ולספק פרטיהם בכל פעם שנדרש מאייתנו. אחרי שעשינו את זה. ניצור בתיקית האב של הפרויקט שלנו קובץ שנקרא procfile – כן, ללא סויומת קובץ. הקובץ הזה מכיל את כל ההוראות לפরיטה של הפרויקט שלנו. במקרה של כל הדוגמאות בספר זה יהיה מש浩 בסגנון: `node ./app.js`.

```
web: node app.js
```

נשמר את הקובץ. עכשו להעלאה. כדי להעלות את הפרויקט שלנו ל-Heroku, אנו נדרשים לעבוד עם מנהל גרסאות מסוג גוט. גיט וניהול גרסאות אינם נלמדים בספר זה אבל זה כלי חשוב מאוד למפתחים. נקליד Heroku create כשאנו בתיקייה הראשית של הפרויקט שלנו. ואחרי כן:

```
git push Heroku master
```

וזאת בהנחה שאתם רוצים לבצע deployment ל-`master`. הקפידו שנל הקבצים שלכם אכן יהיו בגרסה זו. הפעילו את האפליקציה באמצעות:

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

והיכנסו אליה באמצעות open `heroku`. שימושו לב שכבר תועברו לנתיב `sh-heroku` יצרה עבורכם. שימושו לב שם יוצרתם שרת ואתם רוצים לגרום לו לעבוד, הקפידו לא להכנס פורטים במספריים בשרת שלכם. במקומם:

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

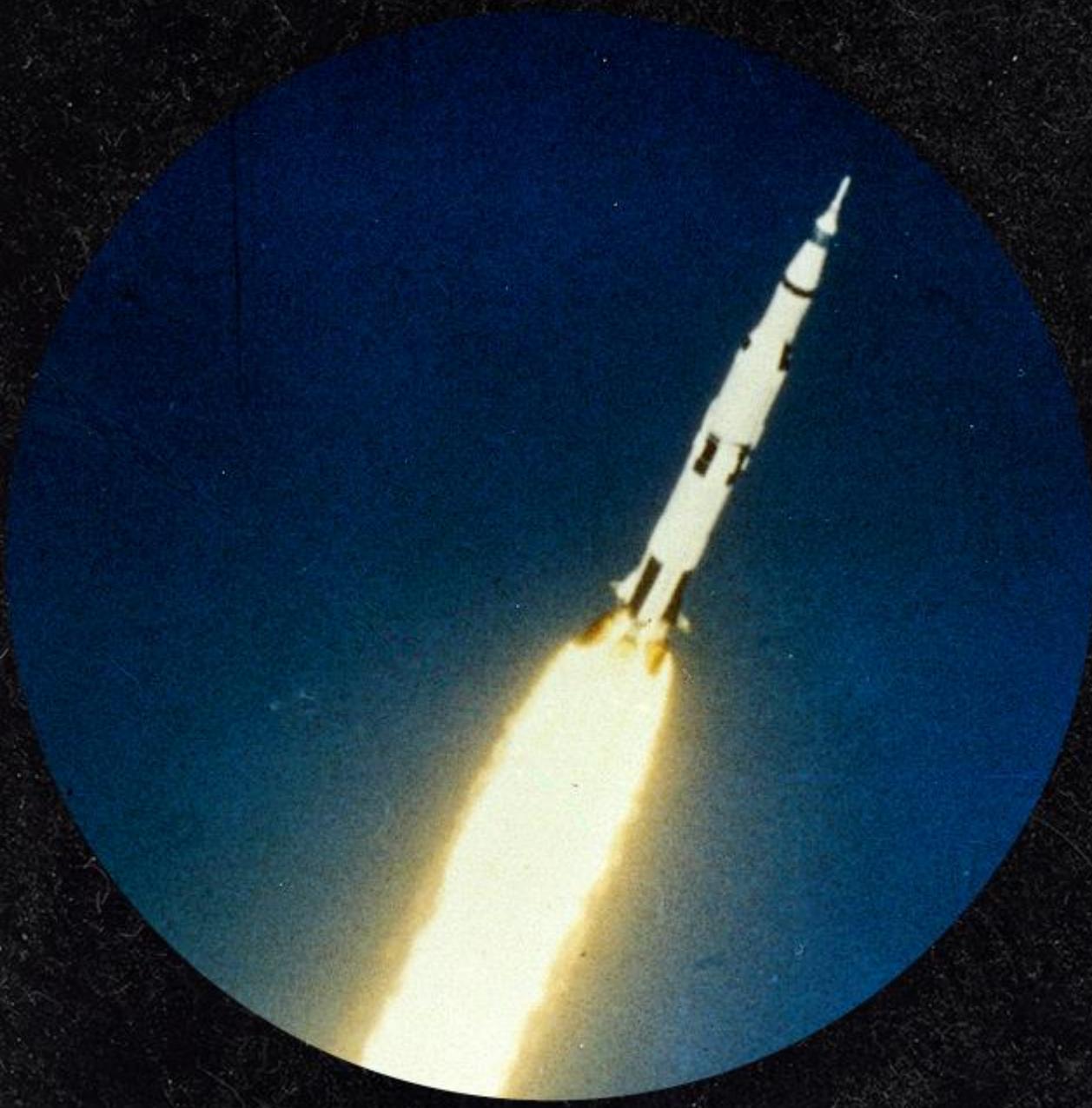
או מהهو בסגנון, הקלידן:

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

אחרת השירות שלכם לא יעבוד-`heroku` (או בכלל בסביבת ענן אחרת).
כאמור, המדריך הזה הוא ברמת `Hello World`. לאפליקציה ממשית זה לא יספיק, כמובן, אבל
זה יהיה מעולה על מנת להראות לכם איך האתר שלכם נראה באינטרנט האמיתי. בשביל אחר
אמיתי – ובטעות בשביל אחר מרובה משתמשים – תצטרכו לחקור יותר את הדוקומנטציה של
`Heroku` ואולי גם להתייעץ עם איש `DevOps`.

פרק 20

סינום



סיכום

אם אתם יודעים ג'אווהסקריפט היטב, ועברתם על כל פרקי הספר זהה ותרגלתם את כל התרגילים – אתם כבר אמורים להבין היטב איך Node.js עובדת ואיך התוכניות שלך רצות. עכשו, כל מה שצרכי הוא ניסיון. לא ידע תיאורטי. שפת תכנות ופלטפורמות שונות לומדים תמיד דרך הידים – ניסיון מעשי. אפשר לצבור את הניסיון המעשית הזה בקלות גם בלי עבודה בחברה.

השאלה שתמיד שואלים היא – איך? איך צוברים ניסיון? למידה תיאורטיבית זה טוב ויפה, אבל יש צורך גם בניסיון פרקטני. מהניסיון שלי, הקהילה בארץ מודד חמה ותומכת ואפשר לפגוש אותה בכמה וכמה מקומות. האנשים שם ישמחו לסייע לכם להשתלב בתחום בהתאם ליכולות ולשאייפות שלכם.

מייטאפים

הדרך הטובה ביותר לפגוש אנשים שעובדים עם Node.js ביוםיום היא להגיע למייטאפים. יש בישראל לא מעט מייטאפים למתכנתים Node.js, שרובם מרכזים באתר meetup.com. הגיעו למייטאפים, שמעו הרצאות מעניינות ודברו עם אנשים שם. הם יכולים לסייע לכם בנושא התקדמות הלאה.

נוסף על מייטאפים יש האקטונים. זו הזדמנות נדירה להצטרף לצוות של מתכנתים מנוסים יותר, ללמוד מהם ולבוד כיצד בפרויקטים אמיתיים.

קובץ דיוון

בפייסבוק יש כמה וכמה קבוצות בעברית המוקדשות לג'אווהסקריפט ול-node.js. הקהילה שם היא חמה ומקצועית. הצטרפו לקבוצות, קראו את הדיוונים והשתתפו בהם. זה יועיל מאוד לדעת מהקצווי שלכם ויכoonו אתכם לעבודות ולפרויקטים.

גיטהאב

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

התנדבות בעמותות ובمיזמים

לא מעט עמותות, מיזמים כמו כניסה פתוחה, למשל, וארגוני מחפשים מתכנתים בתחום דרכם שיווכלו לתרום ולסייע. ההתנדבות במיזמים כאלה יכולה לסייע מאוד להתחיל לעסוק בקוד אמיתי לצד מתכנתים מנוסים יותר.

בין שבחרתם באחת הדרכים שלעיל או בדרך משלכם – חשוב מאד לתרגל, לתרגל, לתרגל. שפה ופלטפורמה הן בדיקן כמו שפה בעולם האמיתי. אם לא תשתמשו ותרגלו – תשחחו. ב-`Node.js` אפשר לעשות המונן דברים. אפשר ליצור תוכנות לשרתים, אבל יש גם פלטפורמות המאפשרות להשתמש ב-`Node.js` על מנת ליצור אפליקציות לנידדים, תוכנות למחשב ביתי ואפילו תוכנות ל-`IoT` – האינטרנט של הדברים. בחרו את אחד התחומיים שימושכם ונסו ליצור פרויקט משלכם. יהיו קשיים, יהיו בעיות – לא הכל אפשר ללמוד מספר או מקורס, טובים ככל שיהיו. נתקלתם בבעיה? חפשו את השגיאה בגוגל ונסו לפתור אותה עצמאית, חפשו באתר StackOverflow או נסו להיעזר באחת מקבוצות המתכנתים בפייסבוק. מה שחשוב הוא לא להתייאש. `Node.js` באמת נמצאת בכל מקום, קל להפעיל אותה והיא יופי של פלטפורמה לכל מתכנת ג'אווהסקייפ. בהצלחה!

נספח: בדיקות אוטומטיות ב-Node.js

מאת דניאל נץ חברת Elementor

מה זה בדיקות אוטומטיות?

תתארו לעצמכם מגדל ג'נגה גבוה, ואתם יושבים עליו בזמן שבאופן אקראי נשלפים ממנו לבנים. מרגנישים את הפחד? אז זהה השגרה בארץ ה-Node.js. הקוד שלכם יושב מעל מגדל כזה הבניי מודולים משתנים כל הזמן, ובו בזמן אתם וחברי הצוות שלכם גם מכניםים קוד חדש שיכول להכיל באגיים חדשים או רgresיות בדברים שעבדו קודם.

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

از איך מייצבים את המגדל? איך בכל זאת בונים כלים ושירותים יציבים דוגמת Netflix ו-ebay, מהשירותים הייציבים והסיקילבליים בשוק.

כדי לענות על השאלה הזאת, בואו נסתכל על תוכנה לא קוד, אלא נאוסף של חזים אשר ממומשים באמצעות קוד. ככלمر, כshmoudol חושף פונקציונליות כלשהי, הוא בעצם יוצר חזה שהפונקציונליות הזאת לא תשתנה בעתיד.

לדוגמה, נניח שיש לנו 2 מודולים של מחשבון; `euclid.js` ו-`pythagoras.js`:

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

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

`euclid.js`

```
function add({ x1, x2 }) {
    return x1 + x2;
```

```

        }

    function subtract({ x1, x2 }) {
        return x1 - x2;
    }

module.exports = { add, subtract }

```

כרגע, ברור לנו לcoldם שני המודולים אינם זהים, שכן בעוד שמודול pythagoras יכול רק לחסר מספרים, המודול euclid יכול הן לחסר והן לחבר. בעצם, נניח שמיימשתי ב-`pythagoras` גם יכולת חיבור, אז באמצעותה מימשתי מחדש גם את יכול החישור כז:

[pythagoras.js](#)

```

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

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

module.exports = { add, subtract }

```

אם הרגע הפכתי את `euclid` ל-`pythagoras`? ברור שלא. למורת שני המודולים כתעת בעלי יכולות זהות, `euclid` לא נהפך ל-`pythagoras` בכלל שיש הבדל בקלט בין מה ש-`pythagoras` מבין מה ש-`euclid` מצפים לקבל; בעוד ש-`pythagoras` מצפה לקבל 2 ארגומנטים (`a` ו-`b`), המודול `euclid` מצפה לקבל אובייקט עם 2 מאפיינים (`x1` ו-`x2`); יש להם חוזים שונים!

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

אם נגידיר את תוכנה zusätzlich של חוזים ממומשים, נוכל להוכיח שהיא והגרסה החדשה של `pythagoras` עדין ממחשת את החוצה שהוגדר בגרסה המקורית שלו (`subtract(a, b)`) – זה עדין אותו המודול.

אבל רגע! האם אכן הגרסה החדשה של pythagoras עדין ממחשת את החוצה הקודם? כדי שנוכל לקבוע בוודאות שהגרסה החדשה מכבדת את החוצה הישן, אנחנו צריכים דרך להוכיח שעובד כל קלט, הגרסה החדשה תחזיר את אותו פלט כמו הגרסה הישנה.

בואו נבדוק כמה דוגמאות של סוגי קלט ופלט על הגרסה המקורית:

pythagoras.use.js

```
const { subtract } = require('./pythagoras'); // @1.0.0

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

// 2. it should support string values
console.log(subtract('3', '2')); // Outputs: 1
```

אוקי, אז ראיינו שהגרסה המקורית מחסרת בהצלחה ערכים מסוג `number` או מסוג `string` ומחזירה תוצאה נכונה – זהו החוצה של המודול.

כעת, נzin את אותו הקלט לגרסה החדשה וננוודא שהחוצה עדין מתקיים:

pythagoras.use.js

```
const { add, subtract } = require('./pythagoras'); // @2.0.0

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

// 2. it should support string values
console.log(subtract('3', '2')); // 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 `3-2` מחזיר `"3-2"`? ציפינו לקבל `1`, החוצה נשבר!

בגרסה הראשונה המתואدة `subtract` תמכה בארגומנטים מהסוגים `number` ו`string`. ובגרסה השנייה זיהינו רגרסיה – חוצה שהוא מכובד בגרסה הקודמת, הופר בגרסה החדשה. מה שגרם לרגרסיה הוא שמייסנו 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, וכך איבדנו את התמייה במחרוזות בפועלות החיסור.

למעשה, הבדיקות שהרגע הרצינו, היו בבדיקות ייחודיה, ובעזרתן גילינו שההミוש החדש לא עונה יותר על החוצה שהגדרנו, או במילים אחרות - מצאנו באנו!

בדיקות אוטומטיות

כל מה שחרס לנו על מנת להבטיח את המשך קיום החוצה של המודול הוא לוודא שאנו חנו מרכיבים את סט הבדיקות הזה שוב אחורי כל עדכון של המודול. וכך ניש לנו עדיין שני אתגרים:

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

על מנת להתמודד על האתגר הראשון נגדיר את התוצאה הרצוייה, ובמידה ומקבלת תוצאה שונה נזורך שגיאה:

```
pythagoras.err.js
const { add, subtract } = require('./pythagoras');

// ...

// 2. it should support string values
const actual = subtract('3', '2');
const expected = 1;

if (actual !== expected) {
  throw new Error(
    `contract 'it should support string values' failed: ${expected} but actual is ${actual}`
  );
}

// ...
```

וכדי שנוכל להרייך את הבדיקות בקלות, נוסף סקורייפט test בקובץ package.json כך:

package.json

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

עכשו נוכל הרץ test run npm ובדעת 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... שבשביל לבדוק 3 שורות קוד, הרגע כתבתי 8 שורות, אתם לגםרי צודקים. ישנוו כלים שמקצרים ומייעלים בהרבה את כתיבת והרצה הבדיקות האוטומטיות. הם מתחולקים בעיקר לשתי קבוצות:

1. מריצי בדיקות - Test Runners דוגמת Jest ו Mocha, אשר אחראים על הרצת הבדיקות והפקת דוחות תוצאה.
2. פריימורוק בדיקות - Assertion libraries דוגמת Chai ו assert-chai שמאפשרים לבטא בקלות את תנאי הבדיקה, ומיצרים תיאור שגיאה אינפורטיבי במקרה של כשלון.

Mocha

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

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

לצורך הוספה של Mocha לפרויקט, נרץ בשורת הפקודה:
 npm install --save-dev mocha

ונשנה את הסקריפט test שב-package.json לשימוש ב-Mocha להרצת הבדיקות:

package.json

```
{
  "name": "pythagoras",
  "version": "2.0.0",
  // ...
  "scripts": {
    "test": "./node_modules/.bin/mocha *.test.js"
  },
  "devDependencies": {
    "mocha": "^6.2.0"
  }
  // ...
}
```

הערה:

לצד ההוספה לפרויקט, ניתן גם להתקין את *mocha* בצורה שתיהיה זמינה להריצה מכל מקום במחשב ע"י הparameter -g:
 npm install -g mocha

חיסרון בכך זו, הוא - להיות ומדובר בהתקינה גלובלית, כל הפרויקטים על המחשב המדובר יישתמשו באותה גרסה של הספריה, ולפעמים זו לא התוצאה הרצויה.

כעת שהוספנו את mocha לפרויקט, בואו נחזור 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);
});
});
});
});
```

אם נריז את הטעט:

```
$ npm run test
```

נקבל את הפלט הבא:

```
> pythagoras@2.0.0 test
...
> mocha *.test.js
```

```
pythagoras
    subtract()
    add()
```

2 passing (16ms)

1 failing

1) pythagoras

```

    subtract()
should support string values:
AssertionError [ERR_ASSERTION]: Expected values to be strictly equal:

```

'3-2' !== 1

...

למעשה, הבדיקה עם דומה מאוד לקוד הבדיקה הראשון שכתבנו, רק שעכשיו mocha מאפשרת לנו לארגן את הבדיקות באמצעות מתודות describe ו-it (עליה נרחב במשך), ומדפיסה אוטומטית את הפרש בין התוצאה הרצוייה לתוצאה בפועל. והחבילת assert מאפשרת לנו לבטא בקלות את התנאים להצלחת הבדיקה, בלי הצורך בכתיבה משפטית זו ארוכים.

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

describe

כפי שראינו בדוגמה מלמעלה, בлок describe מגדר נושא שהוא אנו חנו הולכים לבדוק, אבל לא מכיל את קוד הבדיקות עצמו. ניתן לקוון כמה בложקי describe אחד בתוך השני. לדוגמה, כשאנו רוצים לכתוב בדיקות על מתודה () add של מודול pythagoras אנו יכולים לבטא את זה כבלוק בשם 'add' מקוון בתוך בלוק בשם:'pythagoras'

```

describe('pythagoras', () => {
  describe('add()', () => {
    ...
  });
});

```

הפרמטר הראשון title מקבל את התיאור המילולי של הבלוק, והפרמטר השני fn מכיל את תוכן הבלוק בצורה של מתודה אונומית.

ניתן להשתמש בבלוק describe לא רק כדי לetest בדיקות על חלק טכני של הקוד (כמו מודול או מתודה), אלא גם לפי כל נושא העולה על הדעת. mocha משתמש בהיררכיה של בלוקי describe **לארגון התוצאות**, דבר שמקל מאוד על הבנת הדוחות.

it

כל קובץ בדיקות של mocha חייב להכיל בлок `it` אחד לפחות. בлок `it` מכיל את קוד הבדיקה עצמו והוא יימצא בדרך כלל מוקון בתוך בлок `describe`, אך מבחינה טכנית זה לא נדרש.

הפרמטר הראשון של `title` מקבל את התיאור המילולי של הבדיקה, והפרמטר השני `fn` מכיל מתודה אונומית שהיא הבדיקה עצמה.

החוקיות לפיה נקבעת הצלחת הבדיקה פשוטה מאוד - אם הקוד זורק שגיאה, הטעט נחשב לכושל, אחרת - הצלחה. לדוגמה, הבדיקה הבאה תחשב להצלחה למורות שהיא ריקה מקוד בכלל שהמתודה האונומית 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 נשמש בפרוייקט בדיקות דוגמת `assert` או `chai` לצורך כתיבת הבדיקה, ואלו לאחר מכן יזרקו שגיאה מסווג `AssertionError` במקרה שבדיקה נכשלה.

בינתיים נשמש במתודות בסיסיות של `assert` לצורך הדוגמאות, ונפרט על זה יותר בהמשך.

כל:

2-basic-assert.test.js

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

describe('basic usage', () => {
  it('true assertion', () => {
    assert.strictEqual(1 + 1, 2);
  });

  it('false assertion', () => {
    assert.strictEqual(1 + 1, 42);
  });
});
```

הבדיקה 'true assertion' תעביר, כי ההצהרה $1+1=2$ נכונה, מה שקשה לומר על $1+1=42$ ולכן הבדיקה 'false assertion' תכשל.

מחזור חיים

לפעמים בדיקה תדרוש פעולות הכנה לפניה ופעולות ניקוי אחריה. לדוגמה נניח שאנו בודקים מודול שמבצע שאלות על מסד נתונים. ייתכן שאנו בוטה נרצה להביא את מסד הנתונים במצב ידוע לפני הבדיקה, ולהציגו במצבו ההתחלתי אחרי.

בשביל זה יש ב-mocha את המethodות: `before`, `after`, `beforeEach`, `afterEach`. המethodות האלה נקראות `hooks` – ווומ כי הם מושרים לאיירועים בזמן ריצת הבדיקות.

כל המethodות האלה מקבלות רק ארגומנט אחד - `fn` שהוא מетодה אוננייה שמכילה את הפעולות שיש לבצע בכל שלב.

לדוגמה:

```
before(() => {
  // code...
});
```

בדרך כלל hook מופיע בתוך blök describe והוא חל על כל הבדיקות שבו. אם קיימים כמה בלוקים describe אחד בתוך השני, כל hook חל על הבלוק שבו הוא מופיע ועל כל הבלוקים שבתוכו.

المethodות `before` ו-`after` רצות לפניהם תחילת ביצוע הבדיקה הראשונה בבלוק, ולאחר מכן סיום ביצוע הבדיקה האחרונה בבלוק עליו הן חלות – בהתאם.

المethodות `beforeEach` ו-`afterEach` רצות לפניהם ואחריהם ביצוע של כל בדיקה ובדיקה בבלוק עליו הן חלות – בהתאם.

المmethod `beforeEach` רצתה לפניהם המmethod `describe` והmethod `before` רצתה לאחריו המmethod `after`.

במקרה שיש `hook`-ים בבלוקים מקוונים, המethodות `beforeEach` ו-`before` רצות של בלוק חיצוני יותר רצות לפניהם המethodות הנ"ל של בלוק פנימי, ומmethodות `after` ו-`afterEach` רצות של בלוק חיצוני יותר רצות לאחריו המmethodות הנ"ל של בלוק פנימי.

הערה:

ניתן לכתוב `hook` גם בגוף קובץ הבדיקות, מחוץ לכל ה-`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



מבנה בדיקה

מקובל לחלק כל בדיקה ל-3 חלקים:

1. החלק הראשון - **Arrange**. בחלק זהה אנחנו נכין את כל נדרש לצורך ביצוע הבדיקה ונביא את המודול שబבדייקה למצב הרצוי.
2. החלק השני - **Act**. זה החל שבו בפועל קוראים לפונקציה שబבדייקה עם כל הארגומנטים שהכננו בשלב הראשון.
3. החלק השלישי - **Assert**. פה נמצאת הבדיקה עצמה. בחלק זהה בודקים את הפלט של הפונקציה שబבדייקה, ואם רלוונטי, גם את מצב התוכנה אחרי הביצוע.

בדיקות במבנה זהה נקראות בדיקות **Triple A** או **AAA**.
במקרים רבים החלקים הללו מסומנים בתוך גוף הבדיקה בהערות.

4-triple-a.test.js

```
it('is a well structured test', () => {
    // Arrange
    const text = 'Elementor';

    // Act
    const actual = text.length;

    // Assert
    assert.strictEqual(actual, 9);
});
```

פרימורק בדיקות

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

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

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

המודול `assert` מכיל רשימה של מתודות כשל אחת מהן מיועדת לבדיקת תסרים נפוץ מסויים, ובמקרה של כישלון היא יודעת להפיק הודעה שגיאה מפורטת שיעזרת להבין מה בדיקת השتبש. לדוגמה נשתמש במתודה `:strictEqual`

```
it('will fail', () => {
    const actual = 'foo';
    const expected = 'bar';

    assert.strictEqual(actual, expected);
});
```

הבדיקה תיכשל עם השגיאה:

```
AssertionError [ERR_ASSERTION]: 'foo' === 'bar'
+ expected - actual
```

```
-foo
+bar
```

השגיאה מפרטת מה היה הערך הרצוי, ומה הייתה התוצאה בפועל, ומה ההבדל ביניהם. כמובן שגם דוגמה בסיסית ביותר, ו-`assert` מכיל מתודות נוספות לבדיקות מורכבות בהרבה. להלן כמה מהן:

`assert.ok(value)`

וק שימושי כדי לבדוק אמיתות (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 שהערכים לא זהים. מנגד, `notStrictEqual` יכשל דווקא במקרה שהערכים זהים. לדוגמה:

```
assert.strictEqual('foo', 'foo'); // PASS  
assert.strictEqual('foo', 'bar'); // FAIL
```

תנ"ה

```
assert.notStrictEqual('foo', 'foo'); // FAIL  
assert.notStrictEqual('foo', 'bar'); // PASS
```

חשוב לציין שבשני המקרים ההשוויה תבוצע באמצעות האופרטור == כלומר ללא המרווח. ולכן הבדיקה הבאה תיכשל:

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

ולכן גם השוואת 2 אובייקטים זהים תיכשל, בגלל שאופרטור `==` משווה אובייקטים לפי המצביע. השגיאה גם תכיל הودעה על כך שההשוואה נכשלה בגלל השוני במצביע 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, !assert.deepStrictEqual(actual, expected))  
                                         expected)
```

במקרים בהם שחריך לוודא את שוויון הערכים בשני אובייקטים יש את המתודה `deepStrictEqual` המבצעת השוואת שוויון הערכים בין שני אובייקטים בצוורה רקורסיבית. ההשוואה בין זוג של ערכים שאינם אובייקט או מערך עדין מתבצעת באמצעות האופרטור `==`. הפעם עם `deepStrictEqual` השוואת שוויון הערכים בין שני אובייקטים זהים תעבור:

```
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'
}
```

ועובד בצורה זהה רק שנכשל דוקא 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 בדוק מה שהמתודה throws עשויה. היא מקבלת כารוגמנט פונקציה שאמורה לזרוק שגיאה, והבדיקה עוברת במקרה שאכן נזרקת שגיאה.

ספריות נוספת

כמו בכל נושא אחר ב-Node.js, גם בתחום **פרימורט** בדיקות יש מבחן של אפשרויות שונות. נראה הראואה ביותר לציין היא ספריית **chai**. ב-**chai** יש מבחן עשיר יותר של סוגים בדיקות כמו כן הספרייה תומכת בשלוש סוגיות של כתיבת קוד, ויש עבורה עולם של תוספים. לדוגמה בסגנון כתיבה **expect**, בדיקות נראים כך:

```

        const { expect } = require('chai');

                const foo = 'bar';
const beverages = { tea: ['chai', 'matcha', 'oolong'] };

        expect(foo).to.be.a('string');
        expect(foo).to.equal('bar');
        expect(foo).to.have.lengthOf(3);
expect(beverages).to.have.property('tea').with.lengthOf(3);

```

ספריות mock

עד כה התייחסנו בעיקר לבדיקות של פונקציות טהורות - פונקציות שאין להן השפעות חיצונית (side effects) ולא תלויות חיצונית.

העולם האמיתי לעומת זאת מלא בפונקציות שתלויות אחת בשניה, וגם בפונקציות או שירותים חיצוניים שונים.

להמחשה, נסה לבדוק פונקציה שמקבלת נתיב לקובץ, ומחזירה את מספר השורות באותו הקובץ:

line-count.js

```

const { promises: fs } = require('fs');

async function getLineCount(path) {
const str = await fs.readFile(path, { encoding: 'uft-8' });
return str.split('\n').length;
}

module.exports = { getLineCount }

```

הבעיה היא שהפונקציה `getLineCount` קוראת ל-`fs.readFile`, פונקציה חיצונית, בנוסף צריכה גם קובץ פיזי כדי לעבוד. איך נכתב בדיקה במקרה כזה?

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

נתקין את `sinon` משורת הפקודה:
`npm install --save-dev sinon`

כעת, בעזרה `sinon` נוכל לבדוק את הפונקציה שלנו בצורה הבאה:

0-intro.test.js

```
const sinon = require('sinon');
const assert = require('assert');

const sandbox = sinon.createSandbox();

afterEach(() => {
  sandbox.restore();
});

describe('stub demo', () => {
  it('should return correct line count', async () => {
    // arrange
    const { promises: fs } = require('fs');
    const { getLineCount } = require('./line-count');

    sandbox.stub(fs, 'readFile').resolves('line1\nline2')

    // act
    const actual = await getLineCount(null);

    // assert
    assert.strictEqual(actual, 2);
  });
});
```

בבדיקה, אנחנו משתמשים במתודה `stub` של `sinon` כדי "לדרוס" את המתודה `readFile` של `fs`. מימוש דמה שלנו שטמיך מחזיר את הערך `'line1\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` שעליו קיימת (בין היתר) המתודה `resolves` שבuzzורתה ניתן להגדיר מה `stub` יחזיר כשיופעל. שימוש לב, שאנו קוראים לו `resolves` ולא `returns` כי המתודה `readFile` אמורה להחזיר `Promise`.

בהמשך אנחנו מרים את הפונקציה `getLineCount` עם `null` במקום נתיב הקובץ, ובודקים שההתוצאה היא אכן 2 שורות, בדיקות כפי שצופינו.

בראש הבדיקה ייצרנו sandbox (עליו נרჩיב בהמשך) וב-`each` `afterEach` אנחנו מAppending את ה-`restore` sandbox כדי שהבדיקות מבודיקה אחת לא ישפיעו על הבדיקות הבאות אחריה. קריאה ל`restore` על `sandbox` מבטלת את כל ה"דרישות" שעשינו באותו ה-`sandbox` ע"י `mock` או `stub`.

עכשו נניח שאנו רוצה לוודא שהmethod `getLineCount` אכן משתמש בקידוד utf-8 בזמן קריאת הקובץ `string`. כאן באה לעזרתינו היכולת השנייה של `stub`, והוא ש-`stub` גם משמש כ-`spy`.

source

```

describe('spy demo', () => {
  it('should use utf-8 for reading the file', async () => {
    // arrange
    const { promises: fs } = require('fs');
    const { getLineCount } = require('./line-count');

    const stub = sandbox.stub(fs, 'readFile').resolves('line1\nline2');

    // act
    await getLineCount(null);

    // assert
    stub.calledOnceWithExactly(null, 'utf-8');
  });
});

```

הפעם אנחנו שומרים את ה-`stub` שייצרנו במשתנה `stub` על מנת שנוכל לתחקור אותו בסוף הבדיקה ע"י `calledOnceWithExactly` שמוודאשה-`stub` אכן נקרא בדיק פעם אחת, עם בדיקת הפרמטרים `null` ו-`'utf-8'`.

הmethod `calledOnceWithExactly` זמינה עקב העובדה שב-`stub` כל `stub` הוא גם `spy`.
כעת נפרט קצת יותר על יכולות השונות של `stub`, ועל השימוש בהם.

spy

האפשרות "לרגל" אחריו קריאות למетодות. בעזרת `spy` נוכל לדעת כמה פעמים נקרה method מסויימת, ואילו ערכיהם הועברו לה כארגומנט.

spy מוחזיר פונקציה שאפשר לקרוא לה כמו כל פונקציה אחרת, אך בנוסף לזה, יש לה מבחר מתודות SMAF שמאפשרות 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 "דמה" בעלת התנהגות מוגדרת, ולקבוע מה תחזיר מתודת הדמה עם אחת מהפונקציות או rejects, returns, resolves, throws. כל זה גם כן可以用 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

הmethod stub מאפשר להציג ערכים שונים בהתאם לערכי הארגומנטים ולמספר הפעם שה-stub נקרא. בנוסף ל-stub יש את כל היכולות של fake.

3-stubs.js

```
const stub = sinon.stub();

stub.withArgs('foo').returns('bar');
stub.withArgs('heyyy').returns('hoooo');
```

```

console.log(stub()); // Outputs: undefined
console.log(stub('foo')); // Outputs: 'bar'
console.log(stub('heyyyy')); // Outputs: 'hoooo'

```

בנוסף ליכולות האלו, stub מאפשר מתודה בתוך אובייקט, דבר שמאוד שימושי בדרישת של תלויות ממודולים חיצוניים.

3-stubs.js

```

const obj = {
  greet: () => 'hello!'
};

const stub = sinon.stub(obj, 'greet');

stub.onFirstCall().returns('hi!');
stub.onSecondCall().returns('howdy!');

console.log(obj.greet()); // Outputs: 'hi!'
console.log(obj.greet()); // Outputs: 'howdy!'

```

mock(obj)

בשונה מכל היכולות שכבר פירטנו mock מיעוד יותר לכתיבה וידוא, ולא רק לדרישת אובייקטים. כשלורדים או עוטפים אובייקט ב-**mock** ניתן להגדיר את הציפיות ממנו באמצעות **expects** ו-**verify**, ולאחר השימוש לוודא שכל הציפיות אכן התקיימו באמצעות **verify**. בנוסף, **let**-**mock** יש את כל היכולות של **stub**.

4-mocks.js

```

const obj = {
  greet: () => 'hello!'
};

const mock = sinon.mock(obj);

mock.expects('greet').once().returns('hi!');

console.log(obj.greet()); // Outputs: 'hi!'

mock.verify(); // PASS

```

createSandbox

היות `sinon` דורס את המימוש המקורי של פונקציות ואובייקטים, צריכה להיות דרך להחזיר את האובייקט למקומו בסיום הבדיקה כדי ששינויים מבדיקה אחת לא ישפיעו על הבדיקות האחרות.

מסיבה זאת, מומלץ תמיד שלא להשתמש ב-`sinon` לשירות אלא קודם ליצור `sandbox` (ארגז חול) ע"י קרייה `createSandbox`, ובסוף כל בדיקה (למשל ב-`afterEach`) לקרוא `restore` כדי להחזיר הכל לקדמותו.

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 רוצים לבדוק פונה לרשות לקבלת מידע ולביצוע פעולות? נमובן שנitinן לדוחס מתודות במודול `http` לצורך הבדיקות, הבעיה היא שמאוד קשה "לזייף" שרת `http` אמיתי בגל המורכבות של הפרטוקול.

בשביל זה קיימת ספרייה בשם `nock`. עם `nock` ניתן לתפוס שאלות `http` לפני שהן יוצאות לרשות, ולהחזיר תשובה שנראית כאילו היא בא משרת אמיתי.

נתקין את `nock`
`npm install --save-dev nock`

עכשו, נניח שיש לנו מתודה שמחזירה את מספר הכוכבים שיש לריפו של GitHub-Elementor

```
github.js

const fetch = require('node-fetch');

const getElementorRepoStarCount = async () => {
  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 }
```

ואנחנו רוצים לבדוק 2 תסרים; בהינתן Sh GitHub מגב עם קוד 200, המתודה שמתבקשת סטוס 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');

    // Act
    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 ספריית http אחרת, כי בפועל הוא מתלבש על מודול http הסטנדרטי של Node.js.

כמו ב-hosins, גם ב-nock חשוב להזכיר את המצב לקדמונו בסיום כל בדיקה ע"י `nock.clearAll()`. אך בנגד ל-hosins, ב-nock כל זיווף מחזיק רק לקריאה אחת. לאחר קריאה אחת לכתובות שעלייה עשינו nock, הזיווף נבטל והקריאה הבאה תבוצע לכתובות הרשות האמיתית. את הקריאה ל-`nock.clearAll()` כדאי לעשות במקרה שבדיקה תיכשל לפני שהיא תספיק לבצע את השאלתא, וכן ה-nock ישאר פעיל לבדיקה הבאה.

סוגי בדיקות

כשאנחנו מדברים על בדיקות תוכנה אוטומטיות, אנחנו מדברים על הרבה סוגים שונים של בדיקות, גם במטרת הבדיקה, וגם באופן הביצוע שלהן.

בדיקות יחידה

בבדיקות יחידה אנחנו תמיד מתיחסים לתוכנה כאל אוסף יחידות – חלקים בלתי ניתנים לחולקה, ובודקים כל "יחידה" עצמאית בבודוד משאר התוכנה כדי לוודא שהיא מתפקדת כמו שהיא. הערך שבבדיקות יחידה נונטוות הוא כפוף: מעבר לעצם העובדה שבבדיקות יחידה מאשרות שכל יחידה ויחידה פועלת כמו שצריך, הם גם גורמות לנו לתכנן ולכתוב את הקוד בצורה יותר מודולרית כדי שנוכל לכתוב בדיקה לכל חלק בלי תלות בחלוקת האחרים.

למעשה קיים סגנון פיתוח שנקרא Test Driven Development (פיתוח מכון טסטיים) או בקיצור TDD שבו כתבים את הבדיקות מראש ולאחר מכן כתבים את המימוש על מנת לגרום בדיקה לטעון את המפתחת לכתוב את הקוד בצורה מאוד מודולרית עם תלויות מוגדרות היטב. TDD מחייב את המפתח לכתוב את הקוד בצורה מאוד מודולרית עם תלויות מוגדרות היטב. חשוב לציין שגם רק דרך אחת, ולא חייבים לעשות TDD על מנת לכתוב בבדיקות יחידה אינטואיטיבית.

עם זאת, לביקורות יחידה יש כמובן גם חסרונות. הן אמנם מודאות תקינות של כל יחידה ויחידה, אך לא מבטיחות דבר לגבי התוכנה בשילמותה, בעיקר ככל שהਮורכבות של התוכנה גדולה. בנוסף, זה לא טריוויאלי למצוא איזון טוב בין כמהות ואיזורי הטסטיים לבין אינטואיטיביותם ו齊מודם בין הטסטיים ליחידות אותן הם בודקים.

בכתיבת ביקורות יחידה חשוב מאוד לוודא שבודקים את התוצאה של היחידה ולא את הדרך שבה התוצאה הושגה, אחרת קל להגיע במצב שכל שינוי בקוד ידרשו גם שינוי בבדיקה.

בדיקות קומפוננטה

סוג נוסף של ביקורת הוא ביקורות קומפוננטה. בניגוד לביקורות יחידה שבודקים את התוכנה מבפנים החוצה, בבדיקות קומפוננטה אנחנו מתיחסים לכל המודול כאל יחידה אחת, ומתחמקדים בבדיקה החוצה שהמודול חושף, ככל מרחב פנימה.

אחד מהבעיות העיקריות בבדיקות קומפוננטה היא עמידות לשינויים. ככל מרחב שינויים במימוש התוכנה לא אמורים "לשבור" טסטיים קיימים, ואם טסט נשבר קרוב לוודאי שגם רגסיה ויש לתקן את המימוש.

בדיקות קומפוננטה מתאימות במיוחד לפיתוח זמייש (אג'לי). הן בתסריט שבו אנחנו מפתחים backend לאפליקציית client, ובין אם אנו מפתחים שירות בארכיטקטורת מיקרו-סרייסים, תמיד יש ערך גדול ביכולת להגדיר ממשקים ברורים בין הרכיבים השונים בשלב מוקדם בתהילין, ולאחר מכן לפתחים אחרים להסתמך עליהם. כיסוי של אותן המשקים בבדיקות קומפוננטה מאפשר בהשכמה לא גדולה לוודא שבגרסאות הבאות של השירות אותן המשקים ימשיכו לתפקיד צפוי.

בנוסף, בבדיקות קומפוננטה הן נקודת התחליה הטובה לכתיבת בדיקות, שכן בדרך כלל ניתן לנכונות את ההתקנות הצפויות של שירות Node.js במספר לא גדול של בדיקות.

superTest

ב-node.js ניתן לבצע בדיקות קומפוננטה בצורה מאוד אלגנטית באמצעות ספירה בשם `supertest`. הספירה `supertest` מאפשרת "לזייף" שאלות `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 `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.deepEqual(body, { hello: 'Elementor' });
});
```

בדיקה, אנחנו צריכים את האפליקציה שלנו בשלום, ובאמצעות המתודה get מזיפים שאלת GET לנטייב greet עם ערך name בשלמה header שמצין שאנו מUnoינימム לקלט JSON חזרה, ובעזרת המתודה expect אנחנו מודאים שאנו קיבלו JSON, עם סטטוס 200 והערך של hello הוא אכן Elementor כפי שלחנו בשאלתא.

סוגים נוספים של בדיקות

עד כה הتمקדמו בבדיקות וolidציה - בדיקות אשר מודאות שהתוכנה פועלת בהתאם לציפיות.
אבל גם תוכנה שעובדת להפליא לא שווה הרבה אם קשה להבין ולהזדקק אליה.

eslint

צעד ראשון לקרוא הוא הימצאות 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-linting של כל קוד JavaScript באשר הוא.

כדי להתחיל להשתמש ב-eslint יש לבצע 2 צעדים:

1. להתקין אותו ע"י `npm install --save-dev eslint`
 2. להיות ותקן כתיבה הוא דבר די סובייקטיבי, צריך לספר ל-`eslint` מהו התקן שאנונו מעדיפים ע"י הרצה של `init` ומענה על כמה שאלות:

```
$ ./node_modules/.bin/eslint --init
```

? How would you like to use ESLint? **To check syntax, find problems, and enforce code style**

? What type of modules does your project use? **CommonJS (require(exports))**

? Which framework does your project use? **None of these**

? Does your project use TypeScript? **No**

? Where does your code run? **Node**

? How would you like to define a style for your project? **Use a popular style guide**

? Which style guide do you want to follow? **Airbnb:**
<https://github.com/airbnb/javascript>

? What format do you want your config file to be in? **JavaScript**

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

עכשו כדי לבדוק מה eslint נותן לנו נכתב תוכנה קצרה:

```
app-pre.js
var the_answer = 42;

console.log('The Answer to the Ultimate Question of Life, the
Universe, and Everything is ' + the_answer);
```

:eslint ונרים את
\$./node_modules/.bin/eslint **/*.js

```
./app-pre.js
1:1 error Unexpected var, use let or const instead    no-var
1:5 error Identifier 'the_answer' is not in camel case  camelcase
3:1 warning Unexpected console statement            no-console
3:13 error Unexpected string concatenation      prefer-template
3:91 error Identifier 'the_answer' is not in camel case  camelcase
```

X 5 problems (4 errors, 1 warning)

2 errors and 0 warnings potentially fixable with the `--fix` option.

ואללה! eslint מיזהה את הבעיות הבאות:

- השתמשנו ב-var כשעדיף להשתמש בהשתמש ב-let או const הטעויים יותר.
- שם המשתנה the_answer הוא לא ב-camelCase כפי שהוגדר בתקן של Airbnb.
- אנחנו משתמשים באופרטור + כדי לחבר את הערך של the_answer לטקסט של ההודעה, כשיותר קריא היה להשתמש ב-Template literal.
- השתמשנו ב-console.log(). בפודקשן אין סיבה להשתמש בו ולכנן ההתראה.

עכשו נתקן וננסה שוב:

app-post.js

```
const theAnswer = 42;

/* eslint-disable-next-line no-console */
console.log(`The Answer to the Ultimate Question of Life, the
Universe, and Everything is ${theAnswer}`);
```

ביצענו את השינויים הבאים:

- שינו את שם המשתנה theAnswer ל-theAnswer ב-camelCase.
- הרכזו עליו עם const כי אנחנו לא מתכוונים לשנות אותו.
- השתמשנו ב-Template literal כדי להזrik את הערך של המשתנה לגוף ההודעה.
- והוספנו הערה eslint-disable-next-line מעל הקראיה console.log() כדי להסביר לנו שeslint שאנחנו משתמשים ב-console במודע וזה אינה טעות.

:eslint Nariz

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

\$

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

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

npm audit

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

בשביל זה קיימת ב-`npm` פקודה `audit`.

כשMRIIZIM `audit` מול חבילה `js.Node`, הוא סורק את כל התלויות של אותה חבילה ומדוחה אם מי מהتلויות סובלת מבעיית אבטחה.

למשל כשMRIIZIM `audit` מול חבילה המכילה גרסה בעייתית 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 שמו פיעם בסיום הפלט כל מה שצריך לעשות כדי לפסור 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 אוטומטית מעדכן את החבילה הבועיתית לגרסה בטוחה.

از מה יוצא לי זהה?

בדיקות אוטומטיות פותחות את הדלת לפיתוח גמיש (אג'יל). על מנת לאפשר פיתוח מהיר, יכולת לבצע שינויים בקוד ולהביא אותם לקווק בזמן קצר מבלוי לאבד מאיכות המוצר, חייבת להיות דרך "להקפיא" את הממשק החיצוני של הקוד - החוצה שלו מול הלקוחות, בצורה נזאת שעדיין יהיה ניתן לבצע שינויים במימוש ולהוסיף יכולות חדשות בלי פחד מתמיד שהשינוי ישבור מהו שעבד עד כה.

בדיקות אוטומטיות בשילוב עם תהליכי CI/CD (תהליכי אוטומטי שמריצ' בדיקות לפני שככל פיסת קוד נכנסת למערכת ניהול הגרסאות, וכן לפני שהיא נפרשת בשרת את נשלחת ללקוח) נותנות לבדוק את היכולת הזאת.

כדי למסם את היתרונות של הבדיקות האוטומטיות בפיתוח גמיש, חשוב לתת את הדעת על האיזורים אותם חשוב לנכונות בבדיקות מול איזוריים שנמצאים פחות בשימוש או שהם פחות מורכבים, ולהתחליל את כתיבת הבדיקות דוקא מהמקומות המורכבים והנמצאים בשימוש תדיר. בנוסף, כאשר שלמרות הכל נמצא באג בתוכנה, לאחר תיקונו, כדאי לכתוב טסט שמנסה את הבאג הזה. כך נוכל לוודא שהבאג הספציפי הזה כבר לא יחזור על עצמו וaicות המוצר תשתרף.

ניתן לבדוק את הcoverage (coverage), כלומר את אחוזי הקוד (לפי כמה שורות) שמכוונות ע"ו טסיטים באמצעות הריצה של mocha עם המפתח coverage-. חוכמת השבט גורסת שהcoverage (coverage) האופטימלי של בדיקות הוא עד 85% מהקוד, משלב זה והילך הוספה של בדיקות חדשות לא מחייבת ערך שווה להשקעה.

באמצעות כיסוי בדיקתי של קטיעים קרייטיים לפעולת המערכת, ניתן להגיע למחזורי שחרור קצרים עם אחוזי רגرسיה נמוכים מאוד, מפותחים רגועים ולקוחות מאושרים.

תודה רבה על הרכישה של הספר!

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

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

העתיק הזה נמכר ל:

anguru@gmail.com

בנוסף לדף זה - הקובץ מסומן בטביעת אצבע דיגיטלית - כלומר בתוך דפי הספר נחברים פרטי הרוכש באופן שקוף למשתמש. כדאי מאד להמנע מהעתיקה של הספר ללא רכשו אותו באופן חוקי. אם ברצונכם להעביר את הספר למי שהוא אחר במתנה - העבירו לו את הפרטים שלכם באתר ומיחקנו את העתק ש נמצא ברשותכם.

תודה וקריאה נעימה!