

שםuproיקט:

HabitSpace

פרטי הסטודנט

טלפון	כתובת	ת.ז.	שם
522958147	ארז 18 א.	325048486	זיו איסמיאילוב

פרטי המכללה

שם המכללה: אוניברסיטת סינגלובסקי

סמל המכללה: 75023

מסלול הכשרה: לוחמים להייטק

פרטי המנהה האישי

תפקיד	טלפון	כתובת	שם המנהה

חתימת הסטודנט

חתימת המנהה האישי

חתימת הגורם המוצע מטעם מה"ט

`\${DevOps_Project}`

Ziv_Ismailov

```
> # export DEVOPS_PROJECT=HabitSpace
```

```
> # sudo apt install $DEVOPS_PROJECT
```

הקדמה

בעידן שבו מהירות, יעילות ואמינותם הם הכרחיים לכל תהליך פיתוח תוכנה, פרויקט זה נועד להציג מערכת מתקדמת של אינטגרציה ואוטומציה מלאה עבור אפליקציות מבוססות ענן. הפרויקט עוסק בהקמת פתרון CI/CD חדשני, הבניוי סביב אפליקציה בתצורת app-3-tier – המשלבת משק משתמש, לוגיקה עסקית ומסד נתונים – ומציג גישה מקיפה לניהול מחזור חyi פיתוח התוכנה.

מטרת הפרויקט היא לפתח ולישם פיצלון אוטומטי שמאפשר לא רק לפרסום אפליקציות במהירות, אלא גם לשמר על רמה גבוהה של אבטחת מידע, אמינות, וקנה מידה. המערכת שואפת לגשר על פערים בין צוותי פיתוח, תפעול, ואבטחת מידע, וליצור סביבת עבודה הרמוניית המבוססת על תשתיות ענן חזקות, יציבות וחסכוניות.

קהל היעד של המערכת הוא חברות טכנולוגיה המעוניינות למנף את התשתיות שלהן באופן חכם. המערכת מותאמת社会组织ים שambilנים את החשיבות של אוטומציה ומעוניינים לשדרג את יכולותיהם, גם אם זה כרוך בהשקעה מסוימת בתשתיות. התוצאה היא חיסכון כספי לטוווח הארוך, פרישה יעילה של אפליקציות בענן, והגברת אמון המשתמשים.

בפרויקט זה נשלב כלים ותשתיות מוביילים, בהם:

- **Docker**: לניהול ופרישה של קונטינרים בצורה יעילה.
- **GitHub Actions**: לתהליכי CI/CD מבוססי קוד.
- **Kubernetes**: לניהול תהליכי CD עם ArgoCD.
- **Terraform**: לניהול תשתיות בענן כקוד.
- **AWS**: לשימוש בענן כשירות מרכזי לאחסון, ניהול ותפעול.

כלים אלו ישולבו יחד לפתרון הוליסטי, הנשען על הבנה עמוקה של Linux, המשמשת כבסיס לכל הרכיבים. במהלך הפרויקט נבחנו שיטות עבודה מיטביות, נציג דוגמאות להרצתה ובנייה סביבה שמאפשרת צמיחה עתידית.

יחודיות הפרויקט טמונה בשילוב של מתודולוגיות DevOps עם פתרונות טכנולוגיים מוביילים, תוך التركيز על פתרונות חכמים וautomatisms שמדגשים יעילות, גמישות ושיתוף פעולה בין מחלקות בארגון.

מетодולוגיות עבודה

תהליך העבודה בפרויקט נבנה מותך שאיפה לדמות סביבת פיתוח אמיתית בארגון טכנולוגי, החל מפיתוח הקוד ועד לפרישתו בסביבת הייצור. אחת המטרות המרכזיות הייתה להבין לעומק את הדינמיקה והקשרים בין צוותי הפיתוח, הפעול, ואבטחת המידע, ולמצוא דרכי לשפר את שיתוף הפעולה באמצעות מетодולוגיות וכליים מתקדמים.

הפחשת חיכוך בין צוותים

אחד הביעות הנפוצות בארגונים היא חיכוך בין צוותים שונים, במיוחד במקרים שלבי הפיתוח, הבדיקות, והפרישה. הפרויקט שם דגש על ייצרת פיפלין אוטומטי באמצעות כלים CI/CD כמו ArgoCD ו-GitHub Actions. ס תהליך לדוגמה: מפתח הקוד שולח Pull Request ב-GitHub. הכלי מבצע בדיקות אוטומטיות (Unit Tests, Integration Tests) ושולח עדכונות לצוות הרלוונטי. ס יתרון: מצoom טעויות אנוש ושהרו מהיר ואמין יותר של גרסאות.

גישה מודרנית לניהול תשתיות

כלים Terraform היוו את הבסיס להקמת תשתיות כקוד (CI), גישה שמאפשרת לא רק פרישה אוטומטית אלא גם שחזור מהיר של סביבות: דוגמה מעשית: שימוש ב-Terraform להגדרת שירותי AWS, מסדי נתונים, ורשתות. **יתרונות:** ס תשתיות שמתעדכנות באופן מתמיד בהתאם לצרכים העסקיים. ס שמירה על גרסאות ותיעוד מובנה לכל שינוי.

בחירה בתשתיות AWS

AWS נבחרה כסבירת הענו העיקרי, בשל היכולת שלה לספק שירותים אמינים ואמישים

דוגמאות לשימושים:

- **EC2** – להקמת שירותי גמישים ומובקרים.
- **S3** – לאחסן נתונים בצורה מאובטחת.
- יתרונות AWS: תחזקה ברמות שונות שומותאות לכל צורך עסקית, לצד
 - עליות משתנות ואמישות.

עבודה עם Kubernetes

ניהול אפליקציות מבוססות **Kubernetes** דורש הבנה عمוקה של הטכנולוגיה לצד יכולת- לישם את הפתרונות בצורה אופטימלית:
- שימוש עיקרי: פרישה וניהול של אפליקציות בענן תוך שמירה על זמינות גבוהה וקנה מידה.

אתגרים:

- הגדרת Cluster באופן נכון.
- ניהול משאבים כדי להבטיח שהמערכת פועלת בצורה חלקה.
- שימוש במנגנוני אבטחה כמו RBAC לניהול גישה.

תהליך עבודה אישי

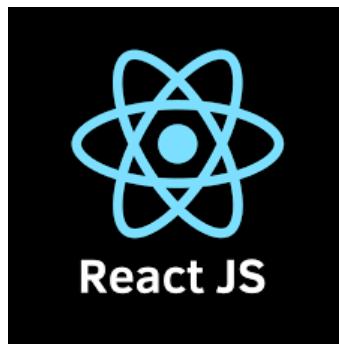
עבודה עם טכנולוגיות אלו דורשת ריכוז רב ויכולת להעמק בפרטם. מצאי שעבודה מסודרת, לצד כלים כמו מחרבת ינית, מסיעת לשמר על שליטה בתהליכי המורכב:

- רישימת שימושות יומיות: מסיעת לתעדך שלבים ולמנוע הבלבול.
- מעקב אחר דוקומנטציה: עבודה נcona עם Kubernetes ו-Terraform מהיבת קרייה מתמדת של דוקומנטציה רשמית. הדוקומנטציה אינה רק כלי עזר – היא המקור להבנה عمוקה ומדויקת של כל טכנולוגיה.

`\${TOOLS}`

```
> # list tools/ | grep -v "jenkins"  
Docker Github_Actions Git Helm  
Kubernetes Terraform AWS Karpenter  
Javascript React PostgreSQL ArgoCD
```

```
> # ./Explain_in_more_details.sh
```



שימוש ב-React ו-JavaScript בפרויקט

```
2 import React, { useState, useEffect } from 'react';
3 import { PlusCircle, CheckCircle, Circle } from 'lucide-react';
4 import axios from 'axios';
5 import styled from 'styled-components';
6
7 // Use window.location.hostname to determine environment
8 const isLocalDevelopment = window.location.hostname === 'localhost';
9 const API_URL = isLocalDevelopment
10 ? 'http://localhost:5000' // Use HTTP for local development
11 : 'https://habits.zivoosh.online/api'; // Changed to HTTPS
12
13 console.log('Using API URL:', API_URL); // For debugging
14
15 const client = axios.create({
16   baseURL: API_URL
17 });
18
```

האפליקציה
שפותחת
 מבוססת על
 טכנולוגיות
JavaScript
 ו-**React**
 שתינה
 מהוות את

היסודות ליצירת ממשק משתמש דינמי ו互動י. הבחירה בטכנולוגיות הללו לא הייתה מקרית – הן הוכיחו את עצמן כפתרונות מודרני ומוביל לפיתוח **אפליקציות רשת (Web Applications)** בעשור האחרון.

המטרות העיקריות:

React ו-JavaScript משמשו למימוש צד הלקוח (Frontend) של האפליקציה, תוך שמירה על חוויית משתמש חלקה ומתקדמת. המטרה הייתה ליצור ממשק המאפשר למשתמשים לבצע פעולות בצורה פשוטה ו互動יבית, תוך שמירה על עיצוב גמיש ומודולרי שמתאים לכל מסך.

יתרונות השימוש בפרויקט

- **חוויות משתמש אינטראקטיביות:** השימוש ב-React אפשר לי ליצור אפליקציה שנראית ועובדת בצורה חלקה, ללא טעינה מחודשת של הדף.
- **פיתוח מהיר וגיוש:** בזכות המודולריות של React, ניתן היה לשלב בקלות רכיבים קיימים בפרויקט ולבנות רכיבים חדשים בעת הצורך.
- **אינטגרציה פשוטה עם Backend:** JavaScript ו-React משלבות היטב עם כל API.

בסיס הנתונים של הפרויקט – PostgreSQL

המטרה:

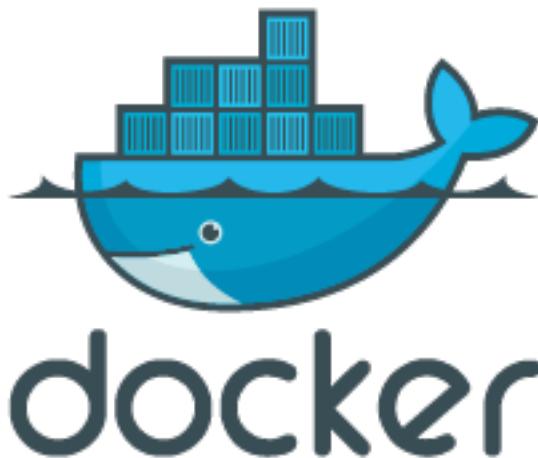
אחד מיסודות הנתונים הרלוונטיים המתקדמים ביותר בקוד פתוח, **PostgreSQL**, שימש בפרויקט שלי כבסיס לאחסון וניהול נתונים. המטרה הייתה לספק מערכת אמינה, מהירה וგמישה לטיפול נתונים של האפליקציה.

הבעיה ש-PostgreSQL פותר:

1. **ניהול נתונים מורכבים:** אפליקציות מודרניות דורשות ניהול נתונים רלוונטיים בצורה מסודרת, עם קשרים בין טבלאות ויכולת לבצע שאלות מורכבות.
2. **אמינות וגבויים:** הדרוש במערכת שmbטיחה שמירה על הנתונים גם במצבים של כשלים או עומסים קבועים.
3. **ביצועים וgamishot:** מערכת שמסוגלת לטפל בكمויות נתונים גדולות תוך שמירה על מהירות תגובה גבוהה.

שימושים בפרויקט:

PostgreSQL שימש לניהול בסיס הנתונים של האפליקציה. הטבלאות שתכננתי כללו נתונים על משתמשים, פעולות שבוצעו במערכת, והגדרות שונות. שילבתי את בסיס הנתונים עם שפת Backend – JavaScript.



מטרה:

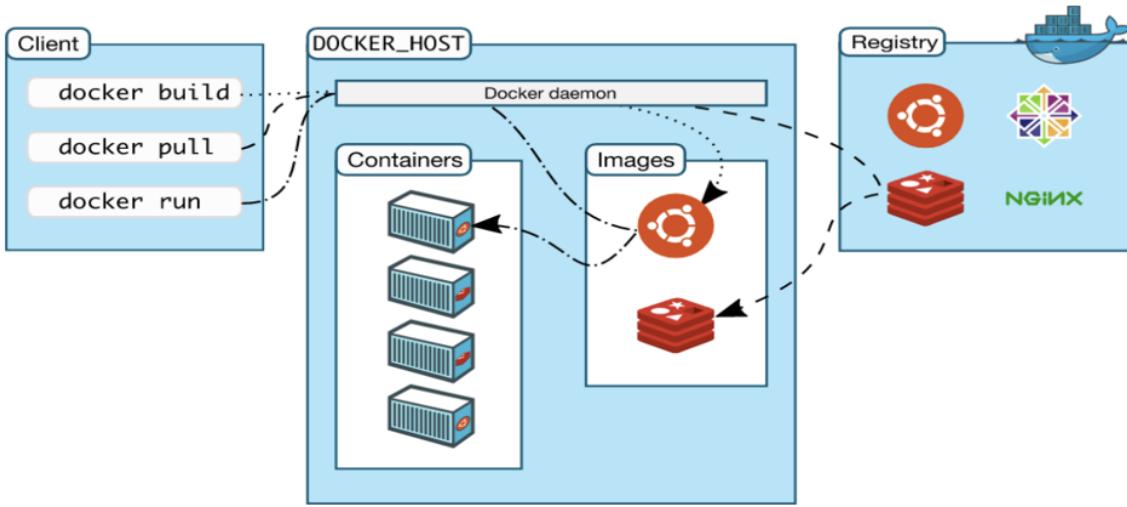
Docker הוא כלי מבוסס קוד פתוח לניהול קונטейינרים, שנועד לפתרו את אחת הביעות הגדולות ביותר בתחום הפיתוח: "**זה עובד במחשב שלי, אבל לא בשרת**". המטרה העיקרית שלו היא להבטיח שהקוד יפעל בצורה זהה בכל סביבה, ללא תלות במערכת הפעלה, התלוויות או הספריות.

הבעיה ש-Docker פותר:

1. **חוסר תאימות בין סביבות:** בעבר, העברת קוד מסביבת הפיתוח לשרתים הבדיקה והיצור הייתה כרוכה בבעיות התאמה רבות (גרסאות שונות של ספריות, הגדרות מערכת שונות וכו').
2. **מורכבות בניהול תלויות:** התקנה ידנית של תלויות ומערכות יצרה תהליכי ארכויים ומועדים לטעויות.
3. **תשתיות לא עקביות:** פלטפורמות שונות דרשו התאמות ספציפיות, דבר שהוביל לעובדה כפולה ולשגיאות.

פתרונות שמציע Docker:

- **יצירת תמונות קונטיאינר (Images)** שמכילות את כל מה שהאפליקציה צריכה לצריכה: קוד, תלויות, מערכת הפעלה בסיסית, ועוד.
- **הרצה של קונטיאינרים** בכל מערכת הפעלה שתומכת ב-Docker, תוך שימוש באותה תמונה בדיקות.
- **הקטנת פערים** בין צוותי פיתוח ותפעול (DevOps) באמצעות סביבות עקביות.



Docker Buildx

Buildx הואCLI שמרחיב את היכולות של Docker לבניית תמונות, במיוחד כאשר נדרשת תמיכה בארכיטקטורות רבות. מטרה: פתרון הבעיה של בניית תמונות שמתאימות לשרתים עם ארכיטקטורות

Name	Last Pushed	Contains
zivism/habit-tracker-backend	about 3 hours ago	IMAGE
zivism/habit-tracker-frontend	about 3 hours ago	IMAGE

שונות, כמו x86_64 ו-arm64 (לדוגמה, שרתים מבוססי ARM לעומת מחשבים וגלים).

*ניתו לראות בתמונה שתי תמונות שנבנו על ידי buildx ונשמרו מיד לאחר סיום

הבנייה ב-Dockerhub.

שימוש בפרויקט: במהלך הפרויקט, Buildx שימש לבניית תמונות אוניברסליות שיכולות לפעול הן בשרתים חזקים מבוססי Intel והן במכשורים מבוססי ARM.

\${Source_Code_Management}

```
> # git add .
```

```
># git commit -m "lets save but use a weird word"
```

ERROR: git branch conflict

```
> # sudo apt uninstall --force jenkins
```

Git

Git הוא מערכת לניהול גרסאות מבוארת (Distributed Version Control System), המשמשת למעקב אחר שינויים בקוד ותיאום עבודה בין מפתחים. הוא מאפשר:

1. מעקב אחר שינויים: כל שינוי בקוד נשמר כהיסטוריה, עם אפשרות לשחזור גרסאות קודמות.
2. עבודה במקביל: מאפשר למספר מפתחים לעבוד על אותו פרויקט במקביל, מבלי לדחוס את עבודות האחר.
3. ניהול ענפים (Branches): מאפשר לעבוד על פיצ'רים שונים בנפרד ולממזג אותם לקוד המרכזי (Main).

דוגמה לפקודות בסיסיות:



GitHub

```
git init  
git add .  
git commit -m "Initial commit"
```

GitHub הוא פלטפורמת אירוח מבוססת ענן שמאפשרת לנו ניהול מאגרי Git באופן מרכז (Centralized). בנוסף, GitHub מספקת כלים מתקדמים כמו:

1. **שיתוף פעולה:** אפשרות למספר מפתחים לעבוד על אותו פרויקט ולבצע Pull Requests כדי להציג שינויים.
2. **ניהול גרסאות בענף:** כל השינויים נשמרים במקום מרכזי, עם אפשרות גישה מכל מקום.
3. **שירותים נלוויים:** GitHub מספקת כלים נוספים כמו GitHub Actions לאוטומציה של תהליכי ופרישות.

הו אלי מובנה בתוך GitHub שמאפשר לבצע אוטומציה של תהליכי מבוססי קוד. הוא מאפשר להגדיר תהליכי (Workflows) שבבסיסים על אירועים ב-GitHub, כגון:

- ביצוע Push למאגר.
- יצירת Pull Request.
- יצירת Tag (Tag) חדש.

מטרות GitHub Actions:

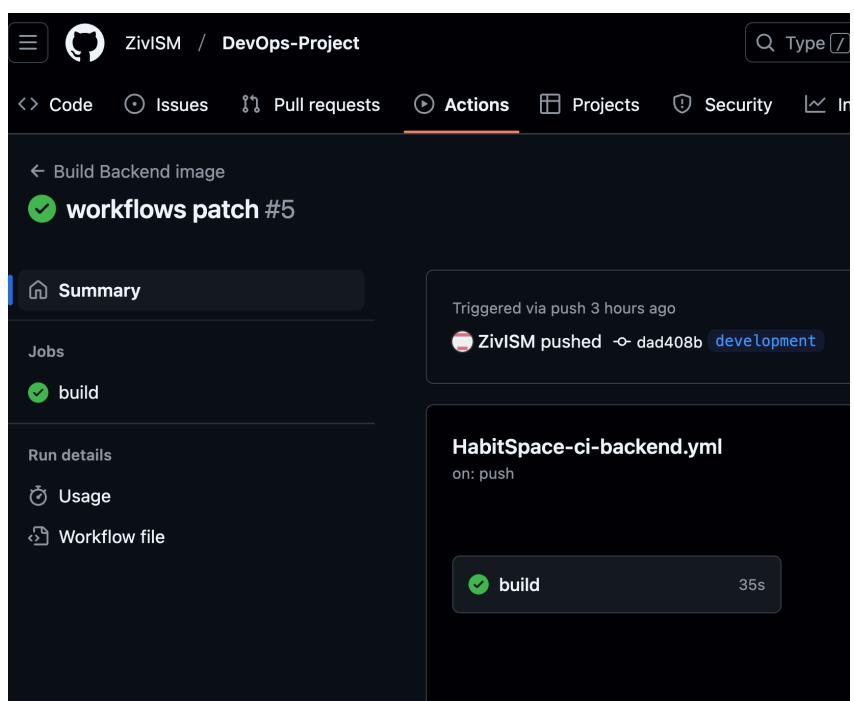
1. **אוטומציה של CI/CD:** שילוב ושחרור שינויים באופן אוטומטי.
2. **בדיקות ובניה אוטומטיות:** הרצת בדיקות יחידה, בניה קוד, ופרישת אפליקציה.
3. **שיקיפות וניהול קל:** מעקב אחרי תהליכי ישירות מממשק GitHub.

הוא תהליך אוטומטי שמוגדר בקובץ YAML ונשמר Workflow

בתיקייה .github/workflows מכיל:

Trigger.1 (אירוע): מה

מפעיל את Workflow, לדוגמה, לאחר Push למאגר.



Jobs.2 (משימות):

סדרת משימות שיבוצעו, כמו התקנת תלויות, בניה קוד, והרצת בדיקות.

Steps.3 (שלבים):

פעולות מפורטות בתוך כל Job.

בפרויקט HabitSpace: ישנו שני תהליכי workflows לבניית תМОנות frontend &

backend& וספק אוטם לכלים הבאים בשרשראת האוטומציה.

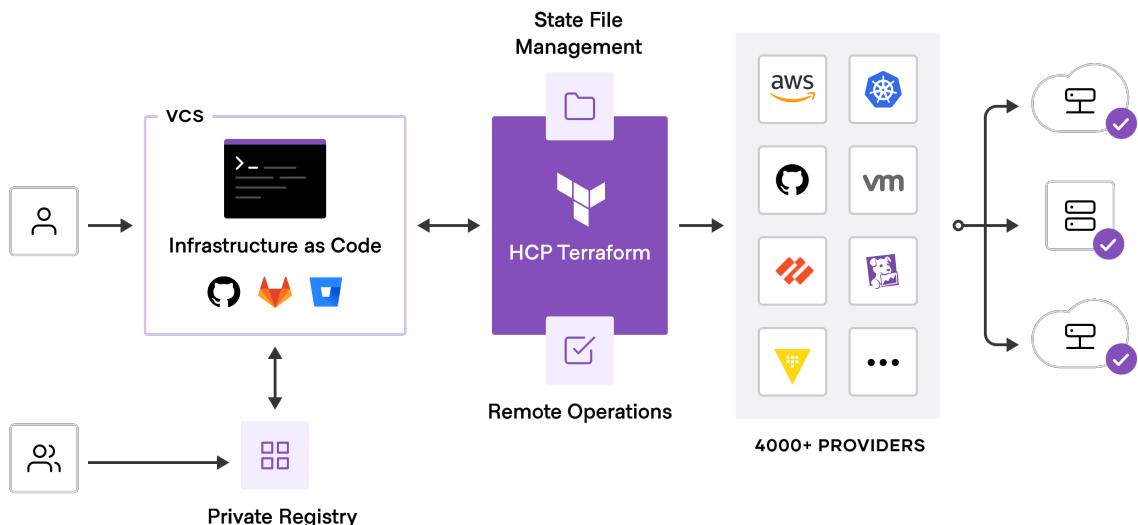
`\${Terraform}`

># **terraform plan**

ERROR: “no good: terraform.tfstate missing. So as your job”

># **aliwurtyhcla**

ERROR: “command aliwurtyhcla not found...”



Terraform

לנהל את כל התשתיות כקוד (IaC), ולהציג בצורה מסודרת ואוטומטית את המשאבים הדרושים בענן. המטרה שלי הייתה להבטיח שכל סביבה – פיתוח, בדיקות או ייצור – תהיה עקבית וניתנת לשחזור בצורה מהירה ופושאה.

בפרויקט השימושי ב-Terraform לניהול תשתיות ב-AWS. הגדרתי בעורתו שרתי, EC2, אחסון S3, הגדרת ענן פרטני ועוד המונ. הכל נטה לי את האפשרות "להכיר" על התשתיות שאני צריך בקובץ קוד ולתת ל-Terraform לטפל בפרישה בפועל. אחת התכונות החשובות הייתה האפשרות לראות מראש מראות כל השינויים הצפויים באמצעות `terraform plan`, לפני שמבצעים אותם בפועל.

אחד האתגרים שנתקלתי בהם היה ניהול קובץ State, שבו נשמרים כל הפרטים על התשתיות המנוהלות. כדי לשמור על יציבות התשתיות, השימושי בשירות S3 לניהול הקובץ בסביבה מבוזרת. בנוסף, העבודה דרשה הכרות עמוקה עם המשאבים של AWS כדי להימנע מטעויות בהגדרות.

Terraform נתן לי לא רק שליטה מלאה על התשתיות, אלא גם ביטחון בכך שכלי שינוי הוא מותען, מסודר ונitin לחזרה בכל רגע. הכלி זהה הדגים עבורי את העוצמה של גישת IaC ושיפר משמעותית את תהליכי ניהול הענן בפרויקט שלי.

```

tra-repo / environments / development / main.tf / module-infrastructure / secrets
1 module "infrastructure" {
2   source      = "../../modules"
3
4   #####
5   # GENERAL
6   #####
7   environment = "development"
8   environment_short = "dev"
9   project = "HabitSpace"
10  aws_region = "us-east-1"
11  tags = {
12    "Project" = "HabitSpace"
13    "Environment" = "Development"
14  }
15
16 #####
17   # NETWORK
18 #####
19  vpc_cidr = "10.0.0.0/16"
20  num_zones = 2
21  enable_nat_gateway = true
22  single_nat_gateway = true
23
24  domain_name = "zivoosh.online"
25  github_repo = "https://github.com/ZivISM/DevOps-Project.git"
26
27 #####
28   # EKS
29 #####
30  eks_enabled = true
31  k8s_version = "1.27"
32  eks_managed_node_groups = {}
33
34  enable_aws_load_balancer_controller = true
35  enable_aws_efs_csi_driver = true
36

```

*בתמונה ניתן לראות את ה-main שלuproject. הקובץ מאגד בתוכו את כל

המודלים הנדרשים כדי להרים את התשתייה בענן.*

\${Kubernetes}

```
># kubectl apply -f .
```

```
shit`s .getting_real .io/HabitSpace-dev unchanged
```

```
># kubectl get pods
```



kubernetes

ניהול ואורקסטראציה של Kubernetes

كونטינרדים

Kubernetes הוא כלי מרכזי בניהול האפליקציה שלי בפרויקט. המטרה העיקרית שלי הייתה להשתמש בו כדי לנהל את הקונטינרדים של Docker בצורה מאורגנת, עם דגש על זמינות גבואה, קנה מידה דינמי, וניהול פשוט של עדכונים ושינויים.

פתרונות Kubernetes פתר עבורי, ועוד מיליוני משתמשים בכל העולם, כמה בעיות מרכזיות. ראשית, במקומות לנוהל קונטינרדים בודדים ידנית, הכלי מאפשר לנוהל אותם בצורה מדויקת ומסונכרנת באמצעות Cluster. היכולת הזאת חשובה במיוחד כי בפרויקט שלי היו כמה רכיבים נפרדים – משק משתמש, הלוגיקה העסקית ומסד הנתונים – וכל אחד מהם רץ בكونטינר מסוילו. בנוסף, Kubernetes טיפול באיזון העומסים בין הקונטינרדים, וכך שלא הייתי צריך לדאוג להתאמת הביצועים בהתאם לעומס המשתמשים.

תכונות `cluster` הוא נושא מורכב שמאגד בתוכו שלל עקרונות תכננות ורטות. למשל: הגדרת קליטת הנתונים של `cluster` מבוחן לענן הפרטי, הצורך בעבר כמו רכיבים מסוימים (`ingress-controller-nginx, alb ingress and more`) לפני שתגיע אל האפליקציה עצמה.

Kubernetes – ניהול חבילות ב-Helm

מה זה **Helm**?

הוא כלי לניהול חבילות עבור Kubernetes, שמאפשר לפרוס, לעדכן ולנהל אפליקציות בצורה פשוטה ומודולרית. במקום לכתוב ולנהל קבצי YAML ארוכים ומורכבים לכל רכיב במערכת, Helm מספק דרך ישרה לאירוע את ההגדרות הללו בחבילות שנקראות **Charts**. לכן, הדרך הכי טובה ליצא את האפליקציה HabitSpace היא לאגור אותה בחילה גמישה שהיא יכולה להיות מותקנת בכל cluster בקלות

מטרת Helm

- **אוטומציה של פרישה וניהול:** ההפחתת הזמן הדרוש לפרישת אפליקציות מורכבות באמצעות קבצים מוכנים מראש.
- **ניהול גרסאות:** הבטחת עקביות בכל פרישה, עם אפשרות לחזור אחריה לגרסה קודמת במקרה של תקללה.
- **הפחתת שגיאות:** ארגון כל ההגדרות הנדרשות בקבצים סטנדרטיים, מה שמקטין את הסיכון לשגיאות אנוש.



איך זה עובד?

1. **Kubernetes – Helm Chart:** חבילה שמכילה את כל קבצי ההגדרות של

über או אפליקציה מסוימת, כולל:

◦ **Templates:** קבצי YAML דינמיים שמאפשרים להשתמש בערכיים

משתנים.

◦ **Values.yaml:** קובץ שמכיל את הגדרות ברירת המחדל של Chart, שאוטו אפשר להתאים לנסיבות שונות.

2. **Release – Release:** כל פעם שמתkinים או מעדכנים Chart, נוצרת גרסה שנקראת Release. זה מאפשר מעקב אחרי שינויים וניהול גרסאות קל.

aws

המטרה:



המטרה העיקרית הייתה להקים תשתיות גמישות, אמינות, ויעילות לכל רכיבי המערכת, תוך שמירה על תחזוקה מינימלית ועלויות מותאמות.

הבעיה ש-AWS פותרת:

- תשתיות פיזיות מורכבות:** במקומות נחלה של שירותי פיזיים או להשקיע בתשתיות מקומיות יקרות, AWS מספקת פתרון מבוסס ענן שבו כל משאב ניתן להתקאה לפי דרישתך.
- חוסר גמישות:** הצורך להרחיב או לצמצם משאבי במהירות, בהתאם לעומסי עבודה משתנים, הוא אתגר מרכזי בפתרונות מסורתיים.
- זמןנות ואמינות:** פרישת אפליקציות בסביבות קרייטיות מחייבת זמןנות גבוהה של שירותים עם גיבויים והთאוששות מהירה מכשלים.

שירותים עיקריים שנעשה בהם שימוש:

- .1 EC2 (Elastic Compute Cloud)
- .2 S3 (Simple Storage Service)
- .3 certificate manager + Route53
- .4 VPC
- .5 Margate+EKS
- .6 IAM (Identity and Access Management)

שימושים בפרויקט:

- הקמתי תשתיות באמצעות **Terraform**, שהגדירה את כל המשאבים של AWS בצורה אוטומטית ומתועדת.
- ניצلت את היכולות של AWS להרחיבת משאבי בהתאם לצורך, במיוחד בסביבות בדיקות וייצור.
- דאגתי לאבטחת מידע באמצעות שילוב נכון של IAM וניהול רשותות פרטיות (VPC).

יתרונות:

1. **גמישות:** AWS מאפשרת לי לשנות את גודל המשאבי ולהוסיף רכיבים חדשים בלחיצת כפתור.
2. **אמינות:** שירותים כמו RDS ו-S3 סיפקו פתרונות יציבים ומאובטחים.
3. **תחזוקה נמוכה:** מרבית השירותים נוהלו על ידי AWS, כך שנחسقو ממני שימושות תחזוקה שוטפות.

אתגרים:

1. **עלות:** היה צורך בעקב שוטף אחרי עלויות כדי להימנע מההוצאות מיותרות, במיוחד בסביבות בדיקות.
2. **מורכבות השירותים:** AWS מציעה מגוון עצום של שירותים, מה שדרש ממני להשקיע זמן בלמידה ובבחירה השירותים הנכונים לפרויקט.
3. **אינטרגרציה:** חיבור בין שירותי שונים כמו EC2, RDS ו-S3 דרש הבנה טוביה של התלות ביניהם.

ArgoCD

> # argocd do yo thing

ArgoCD: OK<3

GitOps – פרישת אפליקציות מבוססת ArgoCD

בפרויקט שלי לניהול פרישות גישת **GitOps**. המטרה אוטומטי, מסודר וסקופי דרך Git ומושם יישורות



המטרה: ArgoCD שימוש Kubernetes באמצעות היהה לייצור תהליכי שבו כל שינוי בקוד מנוהל בסביבת הייצור.

פרישות: פרישות Kubernetes באמצעות kubectl באמצעות ארכוקות, מבבלות לטעויות.

argo

2. חוסר עקביות: כשהועבדים עם מספר סביבות (פיתוח, בדיקות וייצור), נדרש לוודא שכלי סביבה מסונכרנת עם המצב הרצוי (Desired State).

3. מעקב אחר שינויים: קשה לעקוב אחרי שינויים בתוצאות Kubernetes כאשר אין תהליך מבוסס גרסאות.

איך ArgoCD עובד?

ArgoCD מנהל סביבה רעיון **GitOps**, שבו מאגר ה-Git משמש כמקור האמת (Source of Truth). זה אומר שכלי שינוי בתשתיות או בקוד צריך להישנות במאגר ה-Git, ו-ArgoCD דואג לסנכרן את השינויים האלה עם סביבת Kubernetes.

תהליך העבודה:

- 1. מאגר Git המקורי:**(ArgoCD) מקשר למאגר Git שמכיל את קובצי YAML או ה-Helm Charts של הפרויקט.
- 2. סנכרון אוטומטי:** הכלים משווהו בין המצב הנוכחי בפועל ב-Cluster לבין המצב הרצוי המוגדר במאגר.

3. **פרישה ועדכוןים:** ArgoCD מישם כל שינוי חדש ב-Cluster, או מאפשר

פרישה ידנית לפי הצורך.

בפרויקט HabitSpace, Argo סידי מותקן בעצמו על ידי helm, והאפליקציה של Argo סידי מזינה תמידית מקור האפליקציה וכל אינטראול של x שניות, Argo מותקן את האפליקציה החדשה בcluster.

```
# gitops-repo/base/applications/dev.yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: habitspace-dev
  namespace: argocd
spec:
  project: habitspace-dev
  sources:
    - repoURL: https://github.com/ZivISM/DevOps-Project.git
      targetRevision: development
      path: app-repo
      helm:
        valueFiles:
          - $values/gitops-repo/environments/development/values.yaml
    - repoURL: https://github.com/ZivISM/DevOps-Project.git
      targetRevision: development
      ref: values
  destination:
    server: https://kubernetes.default.svc
    namespace: habitspace-dev
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
  syncOptions:
    - CreateNamespace=true
```

`\${karpenter}`

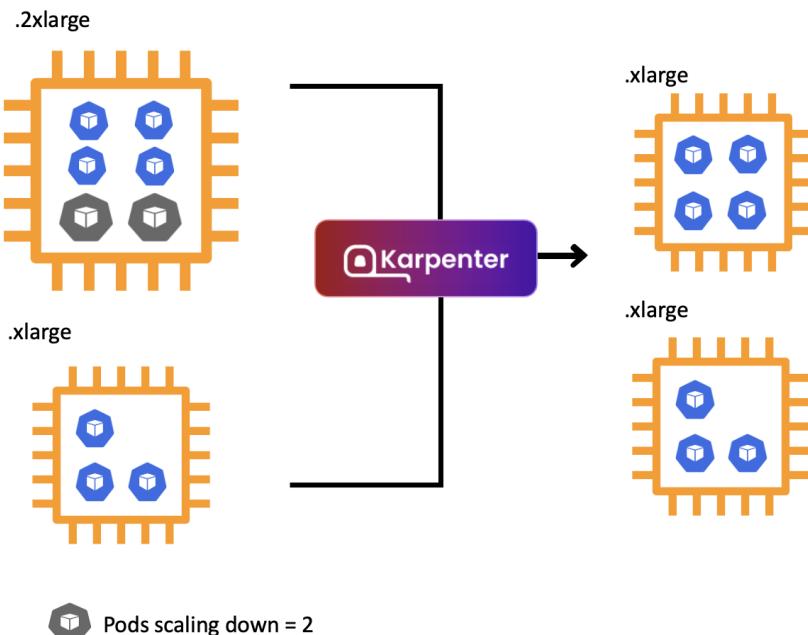
># **karpenter nodes please?**

karpenter: “shut up”

># **kubectl get pods**

ERROR: Karpenter broke everything

NAMESPACE	NAME	READY	STATUS	RESTARTS
argocd	argocd-application-controller-0	0/1	Pending	0
habitspace	habitspace-dev-backend.	0/1	Pending	0
habitspace	habitspace-dev-postgresql-0	0/1	Pending	0
karpenter	karpenter	1/1	Running	0
kube-system	aws-load-balancer-controller	0/1	trying to	0



Kubernetes – אופטימיזציה משאביים ב-Karpenter

המטרה:

הוא כלי אוטומטי לניהול והרחבת משאבי Kubernetes, שותקידו להבטיח שהמערכת משתמשת במשאבים שלו בצורה יעילה, תוך התאמה לעומסי העבודה המשתנים. מטרתו הייתה לעזור לנו להנני חכמה את ה-Cluster שלי, במיוחד כאשר היה צורך להרחביו או לצמצם משאביים על בסיס דרישת.

הבעיה ש-Karpenter פותר:

1. **贊購 משאביים:** לעיתים קרובות, משאביים מוקצים יתר על המידה, מה שסוביל לעליות מיותרות.
2. **ניהול ידני של Nodes:** הגדלת קיבולת ה-Cluster>Dורשת לרוב תהליכי ידני של הוספת Nodes.
3. **עיכובים בפרישה:** עומסים פתאומיים עלולים לעכב פרישות אפליקציות אם המשאביים הזמינים ב-Cluster אינם מספקים.

איך Karpenter עובד?

Karpenter פועל ברמת ה-Node, ומנהל אוטומטית את הוספת והסרת ה-Nodes ב-Cluster בהתאם לעומסי העבודה.

1. **זיהוי דרישות:** Karpenter מזהה Pod שמתכוון למשאים (Pending Pod) וambilן אילו משאים חסרים.

2. **הקצת Nodes:** הכלי יוצר Node חדש (בדרך כלל ב-AWS או ספק ענן אחר) עם הקיבולת המתאימה לצורך.

3. **הסרת Nodes:** אם Node אינו בשימוש, Karpenter מסיר אותו באופן אוטומטי כדי לחסוך בעליות.

שימושים בפרויקט:

בפרויקט שלי, Karpenter שימש לייעול ה-Cluster ולהפחיתת עלויות. לדוגמה:

- כשהעומס ירד, הוא הסיר את ה-Nodes שלא היו בשימוש כדי לחסוך בעליות AWS.

יתרונות:

1. **אוטומציה מלאה:** אין צורך לנצל ידנית את המשאים של ה-Cluster.

2. **חיסכון בעליות:** הוספה והסרת Nodes לפי דרישת מבטיחה שהמערכת לא צורכת משאים מיוחדים.

3. **תגובה מהירה לעומסים:** המערכת מתאימה את עצמה בצורה אוטומטית לצרכים של האפליקציה.

אתגרים:

1. **קונFIGורציה נכונה:** היה צורך להגדיר נכון את Karpenter כדי למנוע ייצור Nodes מיוחדים.

2. **תלות בספק הענן:** הפרויקט היה תלוי בשירותים של AWS להרחבת המשאים.

3. **עיקומת מידע:** נדרש זמן ללמידה ולהבין כיצד Karpenter משתלב עם שאר הכלים ב-Cluster.

Karpenter היה תוסף שימושתי לפרויקט, במיוחד בתהליכי ניהול-h-Cluster כצורה חכמה ו邏輯ית. הוא הדגיש את החשיבות של אוטומציה בניהול משאביים ואת יכולת להפוך את Kubernetes לכלי עצמאי יותר. בנוסף לכך, הטמעת הכלי היא אתגר בפני עצמו ולכון המונע קרייה, ניסוי וטעיה וסיעור מוחין הן המפתח להצלחה.

> # ./RUN_ALL.SH

מבנה האפליקציה והתקיות בפרויקט

הפרויקט מבוסס על שלוש תקיות עיקריות שכל אחת מהן ממלאת תפקיד חשוב בתהליך העבודה: **infra-repo, app-repo, gitops-repo**. התקיות האלה מייצגות את שלושת התחומים המרכזיים של הפרויקט: קוד המקור של האפליקציה, ניהול הגרסאות והפרישות (GitOps), וניהול התשתיות.

1. קוד המקור של האפליקציה - app-repo

תקייה זו מכילה את קוד המקור של האפליקציה HabitSpace, שנכתבה ב-React ו-JavaScript, ואת ההגדרות הנדרשות לבניית התמונה של Docker.

- **תפקיד:**
 - alarach את קוד האפליקציה ואת כל התלוויות הנדרשות.
 - להגדיר את תהליך בניית התמונה של Docker באמצעות .Buildx

• מה קורה ב-push? – GitHub Actions Workflow המתאים

Workflow Actions מזזה את השינוי ומבצע Push למאגר זהה, המתאים כולם:

- בניית תמונה Docker חדשה באמצעות .Docker Buildx
- העלאת התמונה ל-Registry
- עדכון הערכים (Values) ב-repo-gitops כך ArgoCD יזהה את התמונה החדשה ויתחיל בפרישה.

(GitOps - ניהול פרישות gitops-repo .2

תקייה זו משמשת כ"מקור האמת" (Source of Truth) לניהול כל ההגדרות של Helm Charts וה-Kubernetes.

• תפקיד:

- לאחסן את קובצי Helm Chart שגדירים כיצד לפרוס את האפליקציה.
- לעדכן את קובץ-values.yaml עם המידע על התמונה החדשה שנוצרה.
- התחליק: לאחר ש-GitHub Actions מעדכן את הערכים של Helm Cluster, ArgoCD מזהה את השינוי ומחילה את תהליך הפרישה ב-Chart. התחליק כולל התקנת Helm Chart חדש שמשתמש בתמונה Docker החדש של Kubernetes.

3. ניהול תשתיות - infra-repo

תקייה זו אחראית לניהול כל התשתיות בענן באמצעות Terraform.

• תפקיד:

- להגדיר את כל המשאבים הדרושים, כולל Cluster של Kubernetes, ingress, services, configmaps, secrets ועוד.
- להבטיח שה-Cluster מוכן ומותאם לפרישת האפליקציה.
- לפני הכל: לפני כל פרישה, יש להריץ את `terraform apply` מהתקייה זו. תהליך זה דואג:
 - שה-Cluster מוגדר ונגיש.
 - שכל המשאבים הנדרשים (כמו ingress controllers, secrets) קיימים ופועלים.
 - services-ים ופועלים.

זרימת העבודה:

1. Push ל-app-repo

- מפתח מבצע שינוי בקוד המקור ומבצע Push.
- GitHub Actions מזהה את השינוי ומפעיל את ה-Workflow.

.2. **בנייה תמונה חדשה:**

◦ ה-Workflow יוצר תמונה Docker חדשה באמצעות Buildx ומעלה

.Registry-ל.

.3. **עדכון ב-gitops-repo:**

◦ ה-Workflow מעדכן את קובץ ה-.yaml ב-Values.yaml, עם

המידע על התמונה החדשה.

.4. **פרישה ב-Kubernetes:**

◦ מזזה את השינויים ב-o-s gitops-repo ומתחילה את הפרישה

ב-Cluster,(Cluster,Helm Chart) מעודכן.

.5. **ניהול תשתיות עם infra-repo:**

◦ לפני כל זה, נדרש לוודא שה-Cluster מוכן באמצעות Terraform

.services-ingress, secrets כולל הגדרות כמו