משרד הדיגיטל הלאומי
רשות התקשוב הממשלתי | gov

הנחיות ראש רשות התקשוב הממשלתי

עקרונות פיתוח מערכות כהיערכות לענן		
פרק ראשי: טכנולוגיות	מספר הנחיה: 4.2.4	
פרק משני: מיחשוב ענן	מס' גרסה: 2.1	בתוקף מ-22.5.2024

תוכן עניינים

הנחיה זו מתווה קווים מנחים, ועל כן תוכנה בגדר המלצה עבור קהל היעד.

1. הקדמה

הנחיה זה מתייחסת לפיתוח אפליקציות בכלל ולהיבטים השונים של פיתוח מערכות למצב של Cloud Ready בפרט. ההנחיה מתייחסת ל-Best Practices שיש ליישם בתהליכי פיתוח מערכות. מומלץ לאמץ הנחיות אלו כסטנדרט בתהליך הפיתוח להכנת המערכות למעבר לענן.

הנחיה זו מתייחסת לתחומים שונים החל מתהליכים תומכי פיתוח, תכנון ארכיטקטורת המערכת, פיתוח המערכת ועד לממשקי המערכת כגון, בסיסי נתונים, אחסון ולוגים. מערך הדיגיטל רואה בכל ההיבטים האלו מרכיבים חשובים שיש להתייחס אליהם בפיתוח והכנת המערכות למעבר לענן.

2. קהל יעד

- מנהלים ברשות התקשוב הממשלתי
- מנהלי היחידות מונחות רשות התקשוב במשרדים
- מנהלי יישומים
- מנהלי טכנולוגיות

3. מטרת ההנחיה

להנחות את קהל היעד בדבר הפעילויות המומלצות בבואם לתכנן ולפתח מערכות חדשות.

החל מפרסום הנחייה יש להמנע מלפתח מערכות חדשות בטכנולוגיות ייעודיות של ספק ענן מסויים (Cloud Native) או בטכנולוגיות אשר לא יהיה ניתן להעבירם בפשטות לענן.

4. הגדרות ומושגים

CONTINUOUS INTEGRATION - CI

תהליך אוטומציה עבור מפתחים שבו שינויים בגרסת הקוד עוברים תהליך של קומפילציה (build), נבדקים באופן אוטומטי (test) ומשולבים במערכת לניהול הקוד הארגוני.

CONTINUOUS DELIVERY / DEPLOYMENT - CD

תהליך אוטומציה לפריסת השינויים שנעשו בקוד לסביבת הייצור ולסביבות נמוכות (כגון סביבת הבדיקות וסביבת אינטגרציה).

CONTAINER

יחידה של תוכנה שאורזת קוד וכל התלויות שלו, כך שהיישום פועל במהירות ובאמינות מסביבת מחשוב אחת לאחרת.

STATE

שמירת מידע או תלות בנתונים מחוץ לתחום השירות. מידת העצמאות / תלות תקבע עם הוא משתמש ב-state ולכן יוגדר statefull או לא משתמש ב-state ולכן יוגדר stateless.

ראה מילון מונחי התקשוב הממשלתי.

5. תהליכים תומכי פיתוח

פרק זה מתייחס להיבטים שונים של ניהול הפיתוח. פרק זה אינו עוסק במערכת מסוימת, אלא בתהליכים הקיימים בארגון על מנת לתמוך בהיבטים השונים של פיתוח המערכת. ככל שתהליכים אלו אינם קיימים בארגון באופן מוסדר יש לפעול להסדירם.

5.1 ניהול קוד במאגר מרכזי (Source Code Repository)

בעולם ניהול התצורה הקוד נשמר בשרת מרכזי שבו ניתן לשתף קוד, לשחזר גרסה, לאחזר גרסה, להשוות גרסאות ולתת הסבר לכל שינוי בקוד. קוד הנמצא במאגר קוד משותף, כגון Git, מאפשר שיתוף של קוד בין מפתחים בצוות, בין מפתחים בצוותים שונים וכן שימוש חוזר בקוד בפרויקטים שונים.

כמו כן, ניתן לבצע סריקות על הקוד, סריקות פיתוח מאובטח, סריקות לחולשות, לבצע בקרה על הקוד, להפעיל בדיקות אוטומציה של פונקציונליות לפני כל העלאת גרסה ועוד.

קוד זה הינו הבסיס לגרסה של הפרויקט שעולה לסביבות הבדיקות, האינטגרציה והייצור. אין להעלות קוד שנמצא על תחנת עבודה של מפתח ישירות לסביבות יעד שונות כגון סביבת בדיקות, סביבת אינטגרציה וסביבת הייצור. כל קוד שעולה לסביבות אלו עולה מתוך מאגר הקוד המרכזי שנמצא בשרת לאחר שעבר את כל הבדיקות כפי שיוגדר בנהלי העבודה.

5.2 ניהול קוד באמצעות ספק חיצוני

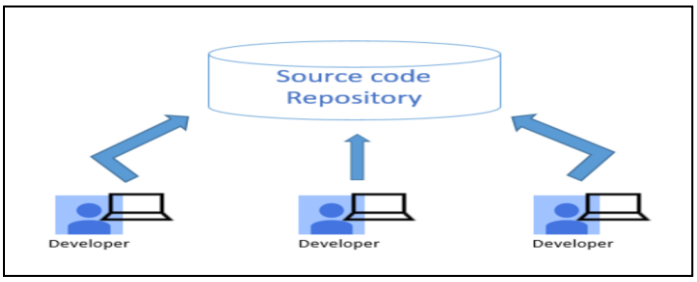
ניהול קוד מספקים שונים יכול להיות מורכב, אך אפשר להפחית סיכונים ולשפר את הסיכויים להצלחה.

תכנון:

- על המשרד להגדיר חלוקת תפקידים ואחריות ברורה: הגדרת מטריצת RACI (Responsible, Accountable, Consulted, Informed) לכל ספק, תוך ציון התוצרים ולוחות זמנים ברורים.
- יש ליצור סטנדרטים לקידוד, פרוטוקולים לניהול גירסאות והנחיות לתיעוד מול ספקים.
- על המשרד למנות נציגים מצוות משרדי למעקב אחר התקדמות הספק.

5.3 ניהול טכני:

- יש להשתמש בשיטה "Single Source of Truth" - מקור יחיד אמיתי. שימוש במערכת ניהול גירסאות מרכזית (לדוגמה, Git) לאחסון כל גרסאות הקוד מכל הספקים, המאפשרת מעקב ושיתוף פעולה קל.
- שימוש בבדיקות אוטומטיות להבטחת איכות הקוד ותאימות בין רכיבים שונים.
- אינטגרציה ופריסה רציפה (CI/CD): הקמת CI/CD pipelines לשילוב ופריסה יעילה של עדכוני קוד כדי למזער סיכונים.
- קביעת פרוטוקולי אבטחה נוקשים לגישת ספקים, סקירות קוד (code reviews) וסריקות פגיעות.
- שימוש בספריות קוד פתוח על ידי הספק מחייב שיקול דעת ויידוע של המשרד. השימוש בספריות קוד פתוח מאומתות מראש נעשה כדי להפחית את זמן הפיתוח ולשפר את איכות הקוד, תוך התחשבות ברישוי ובהשלכות של בעיות אבטחה.
- על המשרד לדרוש ולשמור את התיעוד המפורט של כל הרכיבים המפותחים על ידי הספקים שונים לעיון עתידי.



5.4 ניהול קוד בהתאם למדיניות ארגונית (Branching / Merging)

יש מספר שיטות לניהול הקוד בתוך מאגר מרכזי – שיטות שונות ל- Branching (ניהול גרסאות של הפיתוח מתוך מאגר הקוד) ול- Merging (מיזוג הקוד בסיום הפיתוח). יש להכיר את השיטות השונות הקיימות ולבחון את שיטת העבודה שמתאימה לארגון. מומלץ לנהל את הקוד באופן אחיד ברמה הארגונית.

עקרונות הבסיס לניהול גרסאות הקוד:

1. שבירת הקוד למודולים לא תלויים. כל רכיב עצמאי ינוהל באופן עצמאי במוצר ניהול התצורה על מנת שיהיה קל יותר לשחרר גרסה.
2. לעבוד עם גרסה מבוססת מאפיין (feature branches) לפיתוח שינויים, תיקוני באגים ופיתוחים חדשים.
3. למזג את השינויים לתוך הגרסה הראשית (main branch) בהתבסס על מדיניות הארגון (לדוגמא: בדיקת הקוד לפני המיזוג; הפעלת סקירת קוד; ביצוע סקר קוד; הרצת build על מנת לבדוק שקוד לא "נשבר"; הרצת בדיקות יחידת – Unit test).
4. לשמור על הגרסה הראשית (main branch) ברמת עדכניות שמוכנה לעבור לייצור.
5. עקרון שחשוב מאד לזכור הינו Merge Frequently (מיזוג באופן תדיר) על מנת למנוע התנגשויות בקוד (conflicts).

5.5 קובץ קונפיגורציה - הפרדה של מידע "סביבתי" כערכים המוטמעים בתוך הקוד

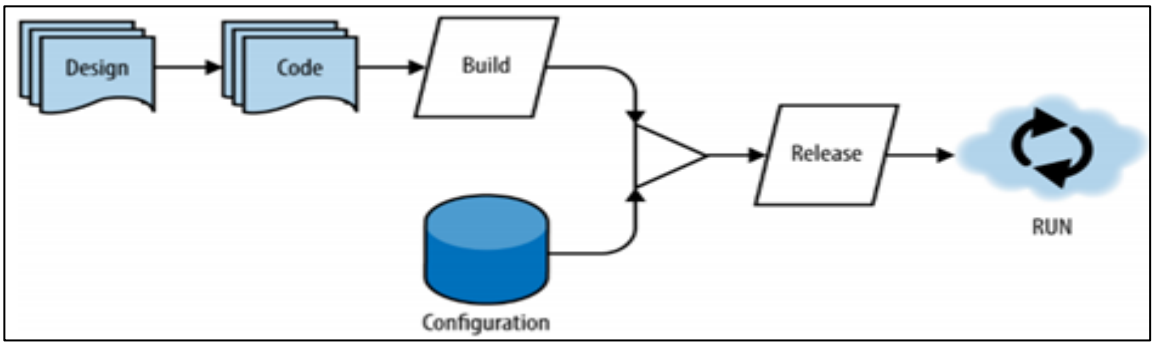
ערכי הקונפיגורציה של פרויקט (ערכי key-value) יופרדו משכבת הקוד וינוהלו באופן נפרד.

שמירת הפרמטרים בקובץ/שרת קונפיגורציה עם ניהול גרסאות, מאפשרת גמישות הפתרון ויכולת התאמתו לסביבת הענן. ככל שמתקדמים לאוטומציה בתהליך הפצת פרויקט לסביבות השונות (בדיקות, אינטגרציה, ייצור) כך עולה הצורך לנהל ערכי קונפיגורציה (בתצורה של key-value) נפרדים לכל סביבה בהתאם. קוד האפליקציה חייב להיות זהה בין הסביבות השונות ולכן חשוב שלא תתבצע מניפולציה על שימוש בערכים שונים בסביבות שונות דרך קוד (if…else). אסור שבתוך הקוד יופיעו קטעי קוד של if … then… שמשנים את הערך בזמן ריצה. כמו-כן אסור להכיל בתוך הקוד ובתוך קבצי הקונפיגורציה נתונים חסויים (מפתחות גישה, סיסמאות).

דוגמא לערכים שמשתנים לפי סביבת יעד:

- ערך ה- connection string בפניה לבסיס נתונים. ערך זה משתנה בפניה לבסיס נתונים שנמצא בסביבת הטסט לעומת פניה לבסיס נתונים הנמצא בסביבת הייצור.
- פניה ל- API - בפיתוח מערכת יש צורך להפנות ל API בכתובת של הטסט ואילו בסביבת ייצור הערך יהיה תואם לסביבת הייצור. במקרה שיש מספר סביבות ייצור הערך יכול להשתנות בהתאם.

- כתובת IP- אין להכניס כתובת IP לתוך הקוד.



המחשת ההפרדה בין קובץ הקונפיגורציה לבסיס הקוד.

5.5.1 ניתן ליישם עקרון זה במספר אופנים:

- בקובץ נפרד, קובץ קונפיגורציה (לדוגמא במערכת net. קובץ הקונפיגורציה הינו web.config או app.config , במערכת java קובץ הקונפיגורציה הינו config.properties, בפרויקט net core. קובץ הקונפיגורציה הינו appsettings.json). בכל סביבת פיתוח (.net, angular, java) התהליך מתנהל באופן שונה, כך שיש ללמוד את התהליך המותאם לכל סביבת פיתוח.
- ניהול ערכים בתוך כספת (vault)
- מוצרי ה- CI/CD מאפשרים להגדיר בתוך המוצר (בתוך ה- pipeline של פרויקט) ערך של key-value. תהליך של build שבונה את הפרויקט יבצע את השילוב של הערכים הקיימים בקובץ ה-config לתוך גרסה הסופית של הפרויקט. תהליך ה -release מתאר את הפרויקט שקומפל לתוך יחידה אחת שמוכנה להפצה לסביבת היעד (טסט, אינטגרציה, ייצור).

בעבודה עם קוברנטיס ולצורך ניהול קבצי קונפיגורציה, מומלץ לעבוד עם helm. helm יוצר אובייקט שנקרא chart שבתוכו נמצאים כל קבצי הקונפיגרוציה הנדרשים על מנת לפרוס resources בקוברנטיס לסביבה שהוגדרה:

- קבצי Manifest: קבצי YAML המגדירים את המצב הרצוי של האפליקציה, כגון deployments services, secrets, and config maps.
- Templates: תבניות אלו מאפשרות להתאים אישית את האפליקציה עם ערכים ספציפיים, כמו בקשות משאבים, פרמטרים של תצורה ומשתני סביבה.
- קבצי Values: מכילים את הערכים בפועל שמוכנסים לתבניות (Templates) כדי לקבוע את תצורת האפליקציה עבור הצרכים הספציפיים.

עבודה עם helm charts מפשטת את הפריסה, מועילה לניהול גרסאות, להגדרות ולשימוש חוזר.

מידע נוסף על ניהול סיסמאות ניתן לקרוא בהנחיית יה"ב 5.13.

5.6 DevOps

עולם הפיתוח הפך לעולם שבו יחידות המחשוב נדרשות לתת מענה מהיר ורצוף לצרכים משתנים של לקוחות. המעבר לפיתוח בארכיטקטורת שירותים מאפשר לממש את הצורך בהפצות תכופות של יחידות קטנות.

במערכות מונוליטיות ההפצה הייתה של חבילה גדולה ולא בוצעה הפצה תכופה של הפרויקט בשל מורכבות ניהול השינויים בבסיס קוד גדול.

בנקודת זמן זו יש, לכל הפחות, להתחיל ללמוד ולהכיר תהליכי DevOps וניתן להתמקד בתהליכי ה- CI/CD (Continuous Integration/ Continuous Deployment) שיאפשרו הפצה מהירה בתהליכים אוטומטיים. הטמעת תהליכי DevOps בארגון הינו תהליך שכולל שינוי תרבות ארגוני, שינוי בארכיטקטורת פרויקטים וליווי התהליך על ידי מומחה טכנולוגי.

אחת המטרות החשובות בתהליך זה הינה להבין שעל מנת ליישם ארכיטקטורת פיתוח של שירותים יש צורך בשיתוף פעולה צמוד בין צוות הפיתוח (dev) לבין צוות תשתיות (ops, operations). צוות התשתיות הינו שותף לפרויקט בהקמת תהליכי אוטומציה ופיתוח יכולות תשתיתיות חדשות בהתאם לצרכי הפרויקט. יש לשלב את צוות התשתיות בתהליך הפיתוח החל מתחילת התכנון ועד לסיום הפיתוח

הטמעת תהליכי DevOps בארגון מחייבת התגייסות, למידה ופיתוח של תהליכי אוטומציה. אין להקל בזה ראש אבל ניתן להתחיל בקטן, לבנות תהליך לפרויקט ראשון ומשם להתקדם לשלבים מתקדמים יותר. מומלץ ללמוד את שפות ה- script כגון Bash , Powershell או Python בהתאם לצורך הארגוני. בשפות אלו ייבנו ה- scripts לתהליכי האוטומציה.

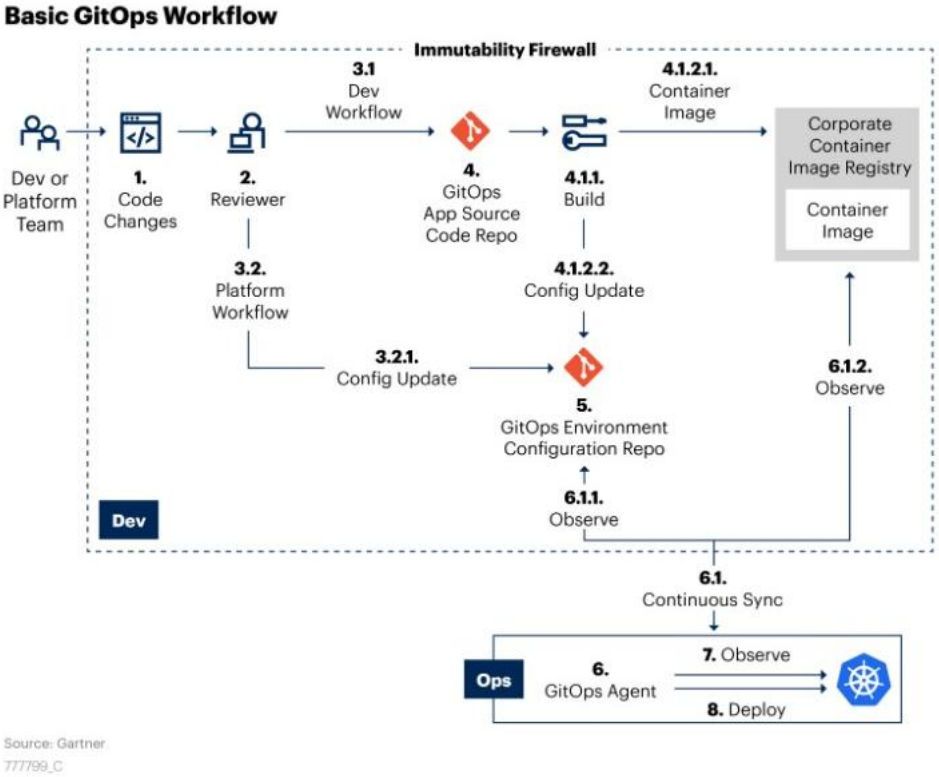
במשרדים שטרם התחילו להטמיע תהליכי devops ההמלצה הינה להתחיל בקטן, לבחור פרויקט אחד שניתן להתנסות דרכו במוצר לבניית תהליך CI/CD ראשוני ומשם להתקדם לתהליכים נוספים. מומלץ להשקיע בפיתוח בארכיטקטורה מבוזרת של שירותים שניתן לארוז בהמשך ל – container. המעבר לשימוש ב- containers מחייב תהליכי אוטומציה והטמעת תהליכי Devops בארגון.

שימוש ב- containers מחייב ניהול קונטיינרים (Orchestrator). אחת מהפלטפורמות הנפוצות לכך היא Kubernetes או בקיצור k8s.

Kubernetes היא פלטפורמת קוד פתוח לאוטומציה של פריסה וניהול קונטיינרים. בעזרת קוברנטיס ניתן להקים ולנהל בצורה יעילה אפליקציות מבוססות קונטיינרים. שימוש בקוברנטיס מאפשר להשיג יתרונות רבים כגון שיפור בסקלאביליות, איזון עומסים, אבטחה ועוד.

ספקי ענן מציעים פתרונות מבוססי קוברנטיס ושירותים מנוהלים , לדוגמא AWS EKS, AWS Fargate, GCP GKE, GCP Autopilot.

בשיטת הפריסה ל- k8s בתהליכי DevOps העקרון המומלץ הוא GitOps. GitOps מאפשר למפתחים לפרוס ולשלוט ביישומים באמצעות קבצים הצהרתיים (declarative) בלבד, המאוחסנים במאגר ניהול גרסאות Git. GitOps משתמש במאגרי Git כמקור אמת יחיד (single source of truth) כדי לספק תשתית כקוד. GitOps מחזיק מאגר Git שמכיל תמיד קבצים הצהרתיים של התשתית הרצויה כרגע (desired) בסביבת הייצור (או כל סביבה אחרת לפי הצורך) ותהליך אוטומטי שיגרום לסביבת הייצור להתאים למצב המתואר במאגר. אם מפתח רוצה לפרוס אפליקציה חדשה או לעדכן אפליקציה קיימת, הוא רק צריך לעדכן את Git והתהליך האוטומטי המתרחש מטפל בכל השאר. כאשר תצורה מעודכנת נדחפת למאגר Git המערכת מבצעת אימות עצמי ומעדכנת את התצורה באופן אוטומטי.



במשרדים שהטמיעו תהליכי devops מומלץ להתקדם לאריזה ל - containers ולהקמת תהליכי אוטומציה מול כלים/שירותים המתממשקים לפיתוח, כגון פרסום לפורטל השירותים (api portal catalog) ופרסום המערכת בסביבת השונות ללא שינוי ידני של ערכי הקונפיגורציה.

5.6.1 DevSecOps

כאשר ה- Security מוכנס לתוך תהליכי ה-DevOps, ניתן להתחיל לעבוד במתודולוגיית ה-DevSecOps.

יש לשלב את צוות אבטחת מידע בתהליך הפיתוח החל מתחילת התכנון ועד לסיום הפיתוח. שילוב מוקדם של צוות אבטחת מידע והנחיות אבטחת מידע יעזרו לשלב את ההנחיות כבר משלב תכנון המערכת ויחסכו שינויים בשלבים מתקדמים של הפרויקט. כמו כן, חשוב מאד לשלב בפיתוח את הנחיית הפיתוח המאובטח. תהליך הפיתוח משלב בתוכו את עקרונות הפיתוח המאובטח SSDLC (Security Software Development Lifecycle).

יש מספר שלבים שבהם ניתן לבצע את סקירת הקוד המפותח:

- בשלב הפיתוח - התקנה על עמדות הפיתוח. הטמעת כלי סריקה ואכיפת מדיניות, על מנת לזהות ולהתריע על חריגה ממדיניות כתיבת קוד ומדיניות אבטחת המידע.
- שלב בניית הקוד – בניית הקוד בכלי ניהול חיצוני (שרת BUILD) – על מנת לזהות בעיות הנובעות משילוב קוד של מספר מפתחים, וזיהוי בעיות הנובעות מהתקנות חיצוניות על מחשבי המפתחים. בשלב זה ניתן להכניס גם רכיבי אבטחת מידע לסריקת הקוד בצורה סטטית וסריקת רכיבי צד ג'.
- שלב פריסת הקוד (CD) – פריסת התוצרים בהתאמה לסביבת הפריסה (שילוב קונפיגורציה יעודית), בדיקת התוצרים בסביבת הריצה שלהם – בדיקות API, בדיקות אוטומציה, בדיקות מקצה לקצה (E2E), בדיקות אבטחת מידע של אפליקציה בסביבת הריצה שלה. יש לפעול להכנסת בדיקות אוטומציה מסוגים שונים על מנת לאפשר זרימה של תהליך פריסת הקוד.

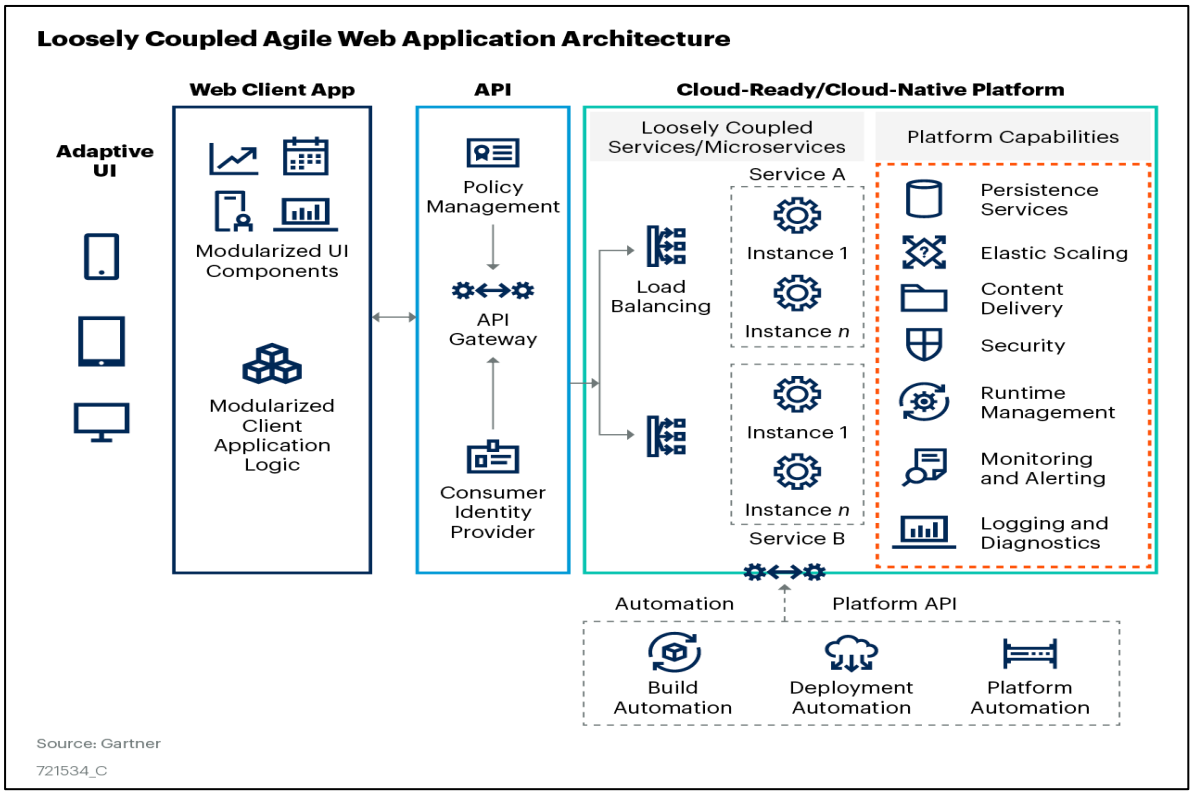
יש ליישם את הנחיית יה״ב 5.13 פיתוח מאובטח בשלבי הפיתוח. ההנחיה נמצאת באתר של יה״ב, הנגיש לממונה הגנת הסייבר במשרד.

6. תכנון ופיתוח המערכת

6.1 ארכיטקטורה של מערכת שמתאימה לפיתוח אג׳ילי

השלב הראשון בפיתוח מערכת שמותאמת לתהליכים אג׳יליים (גמישים וזריזים) הינו לבנות ארכיטקטורה שתומכת בשינויים, שמתאימה לצרכים של היום ויכולה להשתנות בהתאם לצרכים של המחר.

בפרק זה תפורט ארכיטקטורה של מערכת שמחולקת לחלקים שונים, שמעצם החלוקה מאפשרת גמישות לשינויים. העיקרון מאחורי ארכיטקטורה זו הינו חלוקת המערכת: ממשק למשתמש חכם שמותאם לצרכי הלקוח, קוד צד שרת מבוסס שירותים שרצים על פלטפורמה גמישה. חלוקה זו מאפשרת עצמאות לכל רכיבי הפתרון בביצוע שינויים ושדרוגים, בבחירת טכנולוגיה מתאימה בהתאם לצרכים, ובהגדלה וצמצום השימוש בכל רכיב.



הסבר לתרשים (משמאל לימין):

- **Adaptive UI** - גישה למערכת ה– Web בכל מכשיר (נייד, desktop).
- **Web client App** - צד ה- client (צד לקוח) של המערכת.
- **Api** - מוצר ניהול ה- Api (Api Managemnet) שמנהל את הפניות לצד השרת.
- **Cloud-ready/Cloud-Native Platform** – צד השרת. בענן או onprem אותו עקרון.
  - **Loosely coupled services/ microservices**
    - מוצר ה- Load balancing לאיזון וניהול הפניות.

- צד השרת בו יותקנו השירותים sevices (ה- Instances מציג את מספר המופעים של אותו שירות).
    - **Platform capabilities** - מוצרים שונים להשלמת תחום האבטחה, הלוגים, ניטור ועוד.

6.2 WEB CLIENT APP

צד הלקוח של יישום המערכת הינו חלק עיקרי במערכת. אם בעבר רוב הפיתוח התרכז בצד השרת, היום, בשל יכולות חזקות של הדפדפנים, אנחנו רואים יותר ויותר העברה של משקל הפיתוח לצד הממשק ללקוח. פיתוח בצד הממשק מייצר חוויית לקוח מהירה יותר וחוסך פניות לשרת ככל שניתן.

משתמשי המערכות מצפים לקבל מערכות מותאמות לכל מכשיר ולכן בפיתוח המערכות יש להתאים את ה UI/UX לצרכי המשתמשים, כגון פיתוח מערכות מותאמות לטלפון הנייד בתצורה של responsive web page.

- פיתוח מערכות חדשות יעשה ע"י הפרדה בין ממשק המשתמש לבין צד השרת.
- יש לבחון התאמה של סביבת הפיתוח המתאימה ביותר למערכת על פי הדרישות הקיימות במערכת.
- יש לבחון את יכולות הפיתוח הקיימות במשרד ולבחור את סביבת הפיתוח המתאימה ביותר, כגון Angular, React, Vue.
- צד ה-Client (להלן "צד הלקוח") עלול להפוך להיות המונוליט החדש. מומלץ להטמיע תצורת עבודה המחלקת את צד הלקוח לקבוצת רכיבים לא תלויים ברמת ניהול הקוד, כך שניתן יהיה לשחרר גרסאות לא תלויות של רכיבים (micro frontend).
- למערכות צד Client יש שכבת שירותים שמתקשרים עם צד השרת. שירותים אלו הם web client, לדוגמא, באנגולר יש HTTP client service, שמאפשרים פניה מצד הלקוח לשירות API.

6.3 פיתוח שירותי API

ה API מייצר שכבת הפרדה בין שכבת ממשק המשתמש לשכבת השירותים שעומדים מאוחריו.

ככל שה- API קל יותר להבנה על ידי המפתח שמעוניין להשתמש בו, כך זמן הפיתוח שלו יצטמצם ויקצר את זמן הפיתוח סביבו. לפיתוח API פנים-ארגוניים זוהי נקודה עקרונית ואין להקל בה. משמעות הדבר הינה שככל שיש ממשק API ברור לשימוש ושנותן מענה לצורך, פחות תהיה גישה ישירה לבסיס הנתונים/ לממשקים בצד השרת.

ישנם מספר סוגי API שכדאי לציין:

Inner APIs	שירותים אלו חושפים ממשקים לשימוש פנימי של מערכת. מערכות אלו יכולות להיות שירותים מסוגים סוגים (miniservice, microservice, macroservice, legacy system service)
Outer APIs	שירותים אלו מיועדים לשימוש של לקוח חיצוני, יושבים על גבי ה – Inner Api's ובנויים מראש לשימוש של לקוח חיצוני שפונה לקבלת מידע
API mediation	שכבה זו מייצרת את הקשר בין שכבת השירותים הפנימית (Inner APIs) לבין השכבה החיצונית (Outer APIs). שכבה זו מנהלת את הלוגים, אבטחה, הזדהות ותקשורת של כל הפניות. שכבה זו יכולה גם לתרגם את הפניות בין השכבה החיצונית לשכבה הפנימית

6.3.1 Data Formats

JSON	שפה זו נגזרת משפת Javascript. היא תומכת בשילוב של name/value ורשימות. שפה זו נמצאת בשימוש נרחב בשנים האחרונות היות שהיא קלת משקל ומשתלבת היטב עם דפדפנים ומכשירי מובייל.
XML	שפה זו היתה נפוצה בשימוש של שירותים. בשל התגיות הקיימות בשפה זו היא כבדה יותר לשימוש.

6.3.2 שכבת ניהול שירותים (API Mediation layer/API Gateway)

בניית שכבת הפשטה של השירותים הקיימים בארגון עבור הלקוחות שצורכים את השירות, תאפשר גמישות בפיתוח שכבות שירותים נוספות ללא תלות בשירות בשכבה הפנים ארגונית.

צד הלקוח (מובייל, דפדפן, IVR, שירות) מנותק מצד השרת על ידי שכבת הפשטה שמייצרת את הקישוריות בין השירותים.

שכבה זו עוטפת את השירות הפנימי (inner API) וחושפת שירות לשימוש צד הלקוח (outer API). הפשטה זו מייצרת קלות גישה לכל הפונים ומסירה את התלות בין צד הלקוח לצד השרת.

גם עבור יישומים פנים משרדיים מומלץ לבנות את הארכיטקטורה על בסיס שכבת הפשטה זו.

בשיטה זו ניתן להשתמש ברכיבים מסוגים שונים בחלק הפנימי שנחשפים כשירות כלפי חוץ. ניתן לשים לב באיור בנספח ב' שיש שירותים בגדלים שונים. זו המחשה לצרכים השונים שקיימים, החל משירותי microservice ועד לחשיפת שירותים שפונים למערכות מונוליטיות. כולם חושפים ממשק כלפי שכבת הניהול.

בשכבה זו ניתן לפרסם קטלוג של שירותים לשימוש לקוחות שמעוניינים לצרוך שירותים שחשופים. השירותים הקיימים בשכבה הפנימית יכולים לפנות אחד לשני לביצוע משימות וכן יכולים לפנות לממשקים הקיימים בתוך הרשת (בסיס נתונים, אחסון, מערכות ניהול תורים וכו').

איור שממחיש את ה API Mediation layer ניתן למצוא בנספח א'.

- יש לפתח שירותים מבוססי REST ולתמוך ב JSON או XML. יש להעדיף JSON.
- יש לחשוף תיעוד מסודר באמצעות Swagger בהתאם ל OpenAPI 3 Specification.
- שירותים חשופים לשימוש גורם חיצוני: שדרת המידע הממשלתי היא ה- API Gateway לפניות למערכת המיועדת לשיתוף מידע בין משרדי ממשלה, גופים רגולטוריים (בנקים וחברות ביטוח) וגופים חיצוניים (חברות, מערכות בענן, אזרחים). יש להשתמש בתשתית של שדרת המידע הממשלתית.
- שירותים לשימוש מערכות פנים ארגונית: יש להטמיע פתרון של API Gateway לשימוש המערכות הפנימיות. מערכת זו תשמש לניהול הפניות בין שירותים ותנהל את פניות מצד הלקוח לצד השרת.

6.3.3 שפות הפיתוח API

כחלק משלב תכנון מערכת חדשה יש לבחון את צרכי המערכת ובהתאם לקבל החלטה על סביבת הפיתוח/ שפת הפיתוח המתאימה ביותר. בבחירת פלטפורמות פיתוח חדשות יש להתחשב במחיר ובזמן תהליך הכשרת מפתחים לסביבות חדשות. כמו-כן, ככל שקיימים תהליכי אוטומציה ו- CI/CD יהיה צורך לפתוח תהליכים מותאמים.

יש לפתח בשפות פיתוח פתוחות הנתמכות בענן כגון, net core, Java, Node.js. ולהימנע מפיתוח מערכות בשפות שאינם נתמכות בענן.

6.4 ארכיטקטורה

קיימות ארכיטקטורות שונות המתאימות לצרכים שונים של מערכות. על מנת לבצע בחירה נכונה המתאימה למערכת, כדאי להתעמק ולהכיר את הסוגים השונים. בפרק זה נסקור בקצרה מספר ארכיטקטורות.

קיים טווח רחב בין מערכות הבנויות כמערכות שלמות, מערכות monoliths (מכילות הכל בתוך מערכת אחת גדולה), לבין מיקרוסרוויסים microservices. טווח זה מייצג מערכת יחסים של פשרה בין יכולת שינוי מהיר לבין מידת המורכבות של המערכת. ככל שהמערכת גמישה יותר לשינויים כך תעלה עלות התחזוקה והתפעול שלה.

המטרה אינה microservices, אלא בחירת ארכיטקטורה שמתאימה לצרכי המערכת. הארכיטקטורה הנכונה בנויה משילוב של שירותים מסוגים שונים העונים על הצורך המשתנה. המפתח לפתרון נכון הינו איזון נכון בין הסוגים השונים ולא להתמקד בארכיטקטורה מסוימת.

ראש סדר עדיפויות בתכנון ארכיטקטורה צריך להיות זמינות גבוהה של המערכת.

שלב תכנון הארכיטקטורה קריטי למניעת נקודות כשל יחידות (single point of failure) יש לזהות את הרכיבים הפוטנציאליים, שתקלה בהם תשבית את המערכת. לדוגמא, שרת יחיד, מסד נתונים יחיד, או נקודת כניסה יחידה לרשת. יש לשקול הטמעת יתירות (Redundancy) שמשמעה הפעלת מספר מופעים של כל רכיב קריטי ופריסת המערכת במספר אזורים (availability zones).

יש לשלב בבחינת ארכיטקטורת הפתרון ייעוץ של ארכיטקט מערכות (software architect), שיעזור בבחינת צרכי המערכת ויתאים את הפתרון לצורך.

בחירת ארכיטקטורה נכונה מבטיחה שהמערכת מותאמת להרחבה, לחוסן וליעילות בתוך הסביבה המקורית של הענן. ההחלטה על בחירת הארכיטקטורה מבוססת על שיטת הפריסה שלה. פיתוח ופריסת מערכת מסורתית מתבצעת על מכונות ווירטואליות, והענן מאפשר סוגים שונים של מכונות לפי דרישת המערכת.

יחד עם זאת, קיימות גם שיטות אחרות, כגון קונטיינריזציה של המערכת או מיחשוב ללא שרת (serverless).

- קונטיינרים ו serveless הופכים את המחשוב ליעיל וגמיש יותר על ידי שינוי רמת ההפשטה.
- שיטת serverless מאפשרת את השימוש היעיל ביותר במשאבי הענן ובמהירות הפריסה. שיטת זו מבטלת חלק גדול מתקורת התפעול והתחזוקה שקיימת בשיטות וירטואליזציה אחרות (VM, containers). הקצאת המשאבים לביצוע משימה מתבצעת אוטומטית ומאפשרת צריכה רק לפי דרישה מכיוון שאין משאבים פעילים, ואין VMs או קונטיינרים שאינם בשימוש (idle).

לארכיטקטורות Serverless יש מספר יתרונות היוצרות תאימות למגוון רחב של משימות. להלן כמה מהמקרים השכיחים ביותר לשימוש ב- Serverless:

- יישומי טריגרים ע"י אירועים (Event-driven applications): פונקציות Serverless מצטיינות בתגובה לאירועים. כל פעילות משתמש שמפעילה אירוע, כגון רישום משתמש או העלאת קובץ, יכולה להתאים היטב ל- Serverless. לאחר מכן ניתן להשתמש בשרשרת של פונקציות Serverless כדי לטפל במשימות backend המופעלות על ידי טריגרים אלה.
- Microservices: פונקציות Serverless נהדרות לבניית מיקרו-שירותים, שהם שירותים קטנים ועצמאיים העובדים יחד כדי ליצור יישום גדול יותר. ארכיטקטורות Serverless מקדמות קישור רופף (loose coupling) בין שירותים, מה שהופך אותם לקלים יותר לפיתוח, פריסה ושימוש.

- ממשקי API: ניתן להשתמש בפונקציות Serverless כדי ליצור ממשקי API שניתן לגשת אליהם על ידי יישומים אחרים. פונקציה זו מאפשרת למפתחים ליצור שימוש חוזר של הקוד.
- משימות מתוזמנות (Scheduled tasks): ניתן להפעיל פונקציות Serverless על ידי תיזמון זמן (schedules). בניית אוטומציה של משימות שצריכות לרוץ בזמנים ספציפיים, כגון שליחת דוחות יומיים או יצירת חשבוניות.

מתוך גרטנר:

The right architecture is multigrained: a mix of fine-grained microservices, coarsely grained miniservices, embedded macroservices and even monolithic applications in areas where the rate of change is slow and the business justification for refactoring does not exist.

6.4.1 סקירת סוגי שירותים

- Monolith

מערכות אלו הינם מערכות גדולות הארוזות לתוך חבילה אחת. היתרון במערכות אלו הינו הפשטות בהעברת הגרסאות וזיהוי תקלות, אך החסרון הגדול הינו שעבור כל שינוי במערכת יש לבצע בדיקה של כלל המערכת, מה שמקשה על העברת גרסאות תכופות. מערכות שנבנו בתצורה זו מתאימות יותר לתקופה שבה הפצות לסביבות ייצור לא היו תכופות ולא נזקקו לשינוים רבים.

- Macroservice

חשיפת שירות מתוך מערכת מונוליטית. שירותים אלו אינם ניתנים להפצה באופן עצמאי ובסיס הנתונים שהם מעדכנים אינו שייך רק להם.

- Miniservice

חלוקה של הצורך העסקי לשירותים בדומה למיקרוסרוויס, אבל היקף האחריות של השירות רחב יותר והוא יכול להיות בתקשורת עם יותר מבסיס נתונים אחד. בתצורה זו יש גם את יכולת ה scalabity ויכולת ההפצה ללא תלויות. בפתרון זה יש להטמיע תהליכי DevSecOps על מנת להתמודד עם כמות ההפצות הצפויות, היות שיש יותר רכיבים הקשורים למערכת תפעולית ולכן יש עלייה ברמת התחזוקה.

בבחירת ארכיטקטורה זו מומלץ לייצר תבנית של שירות שתכיל את ההגדרות הבסיסיות שממנה ניתן לשכפל את השירותים הבאים. תבנית כזו יכולה לחסוך זמן שמושקע בכל פעם מחדש בהגדרות שיש לבצע בכל שירות.

- Microservice

משמעות המילה "מיקרו" מתייחסת להיקף האחריות של השירות וליכולות השירות להיות עצמאי. הוא יכול להיות מופץ באופן עצמאי ללא תלויות, יכולת scalabilty, עובד מול בסיס נתונים אחד ויש לו אחריות מאד מוגדרת. היתרון במבנה זה של מערכת הוא, שנפילה של שירות אחד אינה משביתה את שאר המערכת מפעילות תקינה. חשוב להבין שאין קשר בין ה"מיקרו" למספר שורות הקוד שקיימות בתוך השירות.

ארכיטקטורה זו מחייבת תמיכה ארגונית בתהליכי DevSecOps. קשה עד בלתי אפשרי לתמוך בארכיטקטורה זו כאשר הארגון נשאר בתהליכים ידניים של הפצת גרסאות לסביבת הבדיקות, אינטגרציה וייצור.

בבחירת ארכיטקטורה זו מומלץ לייצר תבנית של שירות שיכיל את ההגדרות הבסיסיות שממנו ניתן לשכפל את השירותים הבאים. תבנית כזו יכולה לחסוך זמן שמושקע בכל פעם מחדש בהגדרות שיש

לבצע בכל שירות. דוגמא לקוד שניתן להכניס לתבנית: פניה לשירות ה logging, התנהגות בזמן הרצה, התחברות לבסיס נתונים, שירות זיהוי וכו'.

6.4.2 דוגמא - מערכת תפעולית שמיועדת לנהל בית ספר

במערכת יש מידע על מורים, מגמות לימוד, תלמידים, ציונים, טיולים וכו'. המערכת מטפלת בכל הנושאים הקיימים במערכת מידע כגון הצגת מידע, עדכון ומחיקה ומבצעת עדכון בין טבלאות לפי הצורך. המימוש באמצעות:

- מערכת אחת גדולה (monoliths) - נראה מערכת שמחולקת ליחידות עסקיות קטנות שמהוות את צד הממשק למשתמש (UX), הפניה לבסיס הנתונים לכל הצרכים ושכבת הלוגיקה העסקית. מבחינת תפעול מערכת כזאת, יהיה קשה יותר להכניס שינויים בשל גודל המערכת. מפתח אחד לא יוכל להעביר שינוי במערכת ללא תיאום עם מפתח אחר שגם ביצע שינוי. השפעות השינוי במערכת כזאת יכולות להיות רבות היות שיש תלות רבה בין רכיבי הפרויקט. למרות זאת תהליך העברה לייצור פשוט יחסית: יש מערכת אחת, יש בסיס נתונים אחד, מעבירים לייצור כשהמערכת מוכנה. ניהול המערכת בהיבט הזה קל יותר.
- מערכת הבנויה בארכיטקטורה של מיקרוסרוויס - המערכת מחולקת לשירותים קטנים כאשר כל שירות מטפל בנושא אחד. מערכת כזו יכולה להגיע לעשרות שירותים שכל שירות מנוהל באופן נפרד. אם נחבר נקודה זו לתהליכי ה- DevSecOps נמצא בבסיס הקוד הרבה מאד שירותים עצמאיים, קובץ קונפיגורציה נפרד לכל שירות, תהליך קומפילציה (build) ותהליך הפצה לסביבות יעד נפרדות עבור כל שירות. יחד עם זאת, כל שירות הינו אוטונומי ומנהל בסיס נתונים נפרד. בנוסף יש לבחון את נושא ההרשאות והגישה בין שירותים, הצורך בהעמסה של השירות וכו'. ניהול שירותים הינו נושא שיש להכיר ולהבין. ניהול מערכת שכזו יהיה קשה יותר.

  איור הממחיש את הארכיטקטורות השונות ניתן למצוא בנספח א'.

6.4.3 Event driven/ Reactive programming

בארכיטקטורה זו יש נתק מוחלט בין השירותים השונים. אין בכלל קשר ישיר בין השירות שמפרסם אירוע לבין השירות שצורך את האירוע.

על מנת ליישם ארכיטקטורה זו בין שירותים, יש להשתמש במוצר שמשמש "מקשר" (event-driven middleware) לתקשורת בין השירותים – כגון Apache Kafka ו- Pivotal's RabbitMQ. ארכיטקטורה זו מותאמת במיוחד ליישומים שיש להם צורך בהגדלה ולכן מאד מתאימה ליישומים שמיועדים להיות בענן.

יישום ארכיטקטורה זו אינו פשוט ומחייב למידה והטמעה נכונה. מסמך זה אינו מתרכז בהעמקה בנושא זה אך מומלץ ללמוד אותה לפני שמתחילים ביישום הפתרון.

דוגמה לשימוש בארכיטקטורה של אירועים יכולה להיות משיכה של כסף מכספומט. הלקוח מגיע לכספומט ומבקש למשוך כסף. השירות בצד הלקוח (הכספומט) רושם את אירוע המשיכה לתור המקבל ורושם את האירוע. שירות אחר שרשום לאירוע זה מזהה שהגיע אירוע של משיכת כסף מכספומט ומעדכן את בסיס הנתונים על היתרה בחשבון. במקרה כזה, אין תלות בין השירותים. השירות הראשון רשם את האירוע ומאפשר ללקוח לסיים את פעולת משיכת הכסף. במקרה שבסיס הנתונים לא היה זמין, הלקוח אינו מתעכב ואינו מודע שיש תקלה. השירות השני מסיים את משיכת האירוע ועדכון בסיס הנתונים ללא קשר לפעולה הקשורה ללקוח.

איור הממחיש את זרימת המידע בין השירותים בארכיטקטורה מונחת אירועים ניתן למצוא בנספח ג'.

6.5 ניהול STATE

בפיתוח של Microservices ובהמשך פיתוח של containers, או בכניסה לפיתוח בענן בארכיטקטורת Serverless, יהיה צורך לשקול את השימוש ב- state במערכות. יש לבחון מערכות חדשות ומערכות הנמצאות בפיתוח ולקבל החלטה באם המערכת תפותח בארכיטקטורת Stateless או Statefull.

- מערכות שבהן לא ידועה מידת השימוש ושיש להיערך לגידול או צמצום (scale up / scale down), יש לפתח בארכיטקטורה stateless. דוגמה טובה למערכות כאלו הינה מערכות שפונות לאזרחים. לדוגמה, בתקופת הקורונה הייתה עליה משמעותית בפניה לשירותים ממשלתיים, דבר שלא ניתן היה לצפות מראש. יש לתכנן מערכות שמיועדות לשימוש של אזרחים ליכולת גדילה וצמצום לפי הצורך. משמעות הדבר, שאין לשמור בתוך השירות מידע שאם ייעלם לא ניתן יהיה לאתחל את השירות ולהמשיך את העבודה מאותה נקודה שבה הופסקה.

  חשוב להדגיש שצמצום השירות (scale down) בשימוש ב- state מהווה בעיה גדולה יותר מאשר הגדלת השימוש (scale up). קיים קושי לצמצם שירות כאשר יש שימוש ב- state היות ויש חשש מאיבוד מידע.

- במערכות שידועה מראש כמות השימוש ואין צורך בהגדלה או בצמצום השימוש במערכת, ניתן להמשיך לעבוד עם משתני state. לדוגמא, במערכות פנים משרדיות שמשמשות את עובדי המשרד, ידוע מראש שעות העבודה ומספר העובדים שפונים למערכת. אין מניעה מפיתוח המערכת בארכיטקטורה stateless אך אין זה חובה.

ישנם מצבים שיש לשמר מידע של ה- state בדרכים חלופיות. ניתן ליישם stateless במספר דרכים:

- אחסון המידע הנדרש בבסיס נתונים (מומלץ לבחון שימוש בבסיס נתונים קוד פתוח).
- אחסון המידע הנדרש ב- file server.
- עבור containers, ניתן להשתמש ב container-native storage (CNS).
- שכבת cache.
- ניהול בצד הקיינט ניתן לממש בעזרת JWT (JSON Web Token).

7. ממשקים

7.1 בסיס נתונים

המעבר לענן יאפשר להתנתק מניהול בסיס הנתונים כפי שמתבצע במערכות ה -onprem. יכולת זו פותחת בפנינו אפשרויות של גמישות ומהירות במתן פתרונות שונים, יכולת הרחבה מהירה וכן אפשרות להתנסויות מהירות (כגון POC או פיילוט). יחד עם זאת, חשוב להבין שמודל התשלום לספק הענן הינו על בסיס פניות לבסיס הנתונים. יש לבנות את המערכת בחשיבה מראש על מספר הפניות, כך שלא יהיו פניות מרובות ו/או מיותרות לבסיס הנתונים, דבר שיעלה את עלות השימוש. נקודה זו חשובה מאד, היות שבפיתוח מערכות עד היום לא התייחסנו למספר הפניות לבסיס הנתונים בהיבט הכלכלי. נקודה נוספת שכדאי לציין, אבל אינה נקודה חדשה, הינו מבט על מורכבות השאילתות מול בסיס הנתונים. על מנת לייעל את הפניות יש לבנות את השאילתות ללא מורכבות גבוהה שמעלה את זמן השליפה.

ביישומים חדשים, כשניתן לבחור בסיס נתונים חדש, מומלץ לבחון שימוש בבסיס נתונים מבוסס קוד פתוח כגון Sqlite, PostgreSQL (נמצא בשימוש רחב באפליקציות מובייל), MySQL. רצוי לבחון שימוש בבסיסי נתונים לא רלציוניים המסוגלים לעמוד בעומסים ונפחים גבוהים תוך שמירת עקביות, שרידות וזמינות גבוהה ותמיכה בתצורה מבוזרת, כאשר בסיס נתונים גרפי (כגון: Neo4j) מאפשר יצירת שאילתות מורכבות בצורה פשוטה, בסיס נתונים מבוסס עמודות (כגון: Cassandra) מאפשר יכולות אחזור ועיבוד גבוהות לצרכים אנליטיים, בסיסי נתונים מבוססי מסמכים (כגון: MongoDB, Elastic) מאפשרים יכולות שמירה גבוהות של קצבים גבוהים של מסרים ובסיס נתונים מסוג In memory key value (כגון: Redis) לניהול cache והעברת מסרים. ניתן לקרוא את הנחיית ראש רשות בנושא מדיניות שימוש בפתרונות מתקדמים לניהול דאטה.

בסיסי נתונים - הנחיה לביצוע

בפיתוח מערכות חדשות יש לבחון מחדש את השימוש בבסיסי נתונים ולהעדיף שימוש בבסיסי נתונים קוד פתוח ככל שזה אפשרי.

יש לבחון את הצורך העסקי של המערכת ולהתאים את הבסיס נתונים לפי הצורך.

7.2 STORAGE

כפי שהוסבר בסעיף של בסיסי הנתונים, גם בבחינה של אחסון (storage) יש לבחון את אופן השימוש המתאים ביותר. השיקולים במעבר לענן משתנים ויש לבחון את העלויות השונות כפי שלא נעשה בשימוש באחסון הקיים במשאבי המשרד onprem.

לדוגמה, אחסון מידע שאין צורך בו יעלה כסף ולכן יש לתכנן מחיקה של תוכן מאוחסן כאשר אין כבר צורך בו. לדוגמא, בשלב הגשת בקשה על ידי לקוח נשמר מידע ב- file server. בסיום הפעולה עם עדכון בסיס הנתונים מידע זה כבר לא רלוונטי, יש לבצע מחיקה של המידע על מנת לשחרר מקום.

בענן, השימוש באחסון של object storage או בסיס נתונים noSql תופס את מקום השימוש ב- file system מסוג NFS.

- אגפי טד"ם שהתחילו להטמיע עבודה עם containers התומכים ביישומים ששומרים State, משתמשים בשטח אחסון חיצוני משותף (כגון NFS) על מנת שהמידע לא יימחק כשה -container יורד. ניתן להמשיך לעבוד בשיטה זו עד לקביעת מדיניות עדכנית.

7.3 שכבת הזדהות

בארכיטקטורת מערכת מבוססת שירותים הדורשת הזדהות, יש לבצע את תהליך הזיהוי בשכבת ה –API (שכבת ה-API הינה השכבה הפונה לביצוע פעולות אשר מחייבות זיהוי). בפניה ל- API תתבצע בדיקת הרשאות גישה. יש לוודא כי תתמוך בפרוטוקולים OpenID +OAuth 2.0 או SAML 2 לצורך מימוש הזדהות המבוססת Service Provider (SP) ו- Identity Provider (IDP) לצורך ביצוע SSO. הצורך ב- SSO הינו מעבר בין שירותים ללא צורך בהזדהות נוספת.

יש להבדיל בין מערכות המיועדות לשימוש בתוך המשרד לבין מערכות המיועדת לשימוש חיצוני, כגון שיתוף מידע בין משרדי ממשלה ובין גופים רגולטוריים (בנקים וחברות ביטוח) וגופים חיצוניים (חברות, מערכות בענן, אזרחים).

יש לוודא תמיכה בפרוטוקולים **OAuth** 2.0 +OpenID או SAML 2.

7.3.1 מערכת לשימוש פנים משרדי

במערכות פנים משרדיות יש להתייחס גם לתהליך זיהוי פנימי והרשאות של משתמשים מזוהים של המשרד. בשלב ה"כניסה" לשירות מתבצעת בדיקה שאכן יש הרשאות לגשת לשירות. שלב הזיהוי הראשוני, שהינו זיהוי ואימות המשתמש (תהליך Authentication), מתבצע מול Token שמועבר ממערכת ניהול ההרשאות החיצונית לשירות. לאחר שאושרה כניסה לשירות יכולה להתקיים, בהתאם לצורך, בדיקה נוספת, בדיקה אפליקטיבית של הרשאות נוספות (תהליך Authorization), הרשאות אישיות ספציפיות לביצוע ברמת האפליקציה.

לדוגמא, כל עובדי משרד מורשים לצפות בנתונים ממאגר מסוים אבל רק לדרגת הניהול הבכירה יש הרשאות עדכון. במקרה כזה, בהנחה שיש שירות אחד בארגון שפונה למאגר הנתונים, בדיקת רמת ההרשאות הפנימית תתבצע בתוך השירות, כאשר זיהוי העובד שנגיש לנתונים מתקיים מול מוצר חיצוני לניהול הרשאות.

שלבי הגישה:

1. הזדהות מול מוצר חיצוני לקבלת Token.
2. אישור כניסה לשירות.

3. בשירות תתבצע בדיקת הרשאות העובד שנכנס מול הפעולות שמעוניין לבצע. רמת ההרשאה מתקבלת מגישה למוצר לניהול הרשאות חיצונית למערכת. מערכת מקבלת את רמת ההרשאה ומאפשרת גישה/פעילות לפי ההרשאה שהתקבלה.

7.3.2 מערכות לשימוש חיצוני

מערכת שמיועדת לשימוש באינטרנט תחת האזור האישי – יש להתחבר לשימוש במערכת ההזדהות הלאומית.

במערכת שמיועדת לשימוש במערכות פנים משרדיות או ללקוחות אחרים, יש לוודא שהמערכת תומכת ב saml, Oauth + openid לביצוע SSO (Single Sign On). יש לוודא כי תתמוך בפרוטוקולים OAuth 2.0+ OpenID או SAML 2.

7.4 LOGGING

בכל הנוגע למערכות ברשת המשרד, במסגרת מיפוי המקורות במשרד שמתבצע עם הסוק הממשלתי, יש לוודא העברת לוגים, כתיבת חוקים וקבלת התראות המועברות למערכות SIEM ולסוק הממשלתי.

על מנת לאפשר איתור של מידע מתוך מאגרי הנתונים שמצטברים, יש להקפיד על ביצוע רישום של המידע באופן אחיד ולציין את תוכן המידע וחותמת הזמן.

לוגים צריכים להכתב על כל אירוע משמעותי ולבצע את שמירת לוג במקום נגיש כגון syslog או טבלה בבסיס נתונים.

יש מספר סוגים של מידע שמנוטר:

- security software logs
- operating system logs (System Events, Audit Records)
- application and database logs

Logging - הנחיה לביצוע

- יש לפעול על פי הנחיות יה"ב לרישום וניהול לוגים במערכת SOC ממשלתי או מערכת SIEM ממשלתי.
- על המשרד לשמור את הלוגים בהתאם למדיניות המשרדית.
- בשירותים הכפופים לתקנות הגנת הפרטיות, יש לפעול במסגרת תקנות להגנת הפרטיות המתייחסות לבקרה ותיעוד הגישה לנתונים. ניתן לצפות ב[הנחיות החוק].
- משך הזמן בו הלוגים נשמרים בסוק הממשלתי הינו חצי שנה. שמירת הלוגים בסוק הממשלתי אינה מהווה גיבוי או תחליף לשמירתם על ידי המשרד.

8. סיכום

הנחיה זו מתייחסת לתחומים שונים החל מתכנון ארכיטקטורת המערכת, תהליך הפיתוח והפצה ועד לתהליך איסוף הלוגים כשהמערכת כבר מותקנת בייצור. ניתן להבחין שפרקים רבים מתוך ההנחיה מתייחסים ל-Best Practices שיש ליישם בתהליכי פיתוח מערכות ללא קשר לסביבת הפיתוח, ענן או onprem. לדוגמא, השימוש במערכת לניהול תצורה הינו בסיס לפיתוח קוד בכל הארגונים וגם בממשלה. פרק בנושא תהליכי תומכי פיתוח הינו הבסיס לכל תהליכי ה- devOps שמהווה שלב מאד משמעותי למעבר ארגון לענן. זאת הסיבה שבהנחייה זו יש התייחסות למכלול של נושאים שביחד מרכיבים את הנקודות לבחינה בשלב פיתוח מערכות, גם לענן וגם לפיתוח onprem. הפרק המתייחס לארכיטקטורת המערכות מהווה בסיס ליכולת לארוז את השירות ל- container במעבר לענן וכולל התייחסות לסוג האחסון שמומלץ להתחיל לבחון, אך גם יכול לשמש משרדים שמתחילים להטמיע תהליכי devOps ו-containers ברשת המשרדית onprem.

כל החלקים יחד מרכיבים את הפאזל של פיתוח מערכת והכנת המערכות ליום בו הממשלה תעביר את המערכות לענן. קשה להתייחס לחלק מהמרכיבים מבלי להבין את החיבור לחלק האחר. קיימת תלות בין כל הרכיבים על מנת לייצר אקוסיסטם ארגוני מוכן לענן.

באיור זה ניתן לראות הממשקים שיש בפיתוח מערכת:

9. מסמכים ישימים

9.1 הפניה למסמך פיתוח מאובטח של יה"ב 5.13

9.2 הנחיית ראש רשות בנושא מדיניות שימוש בפתרונות מתקדמים לניהול דאטה

9.3 המדריך המלא ליישום תקנות הגנת הפרטיות (אבטחת מידע)

9.4 FACTOR APP 12

9.5 OPENAPI 3 SPECIFICATION

9.6 LIST OF NOSQL DATABASE MANAGEMENT SYSTEMS

10. נספחים

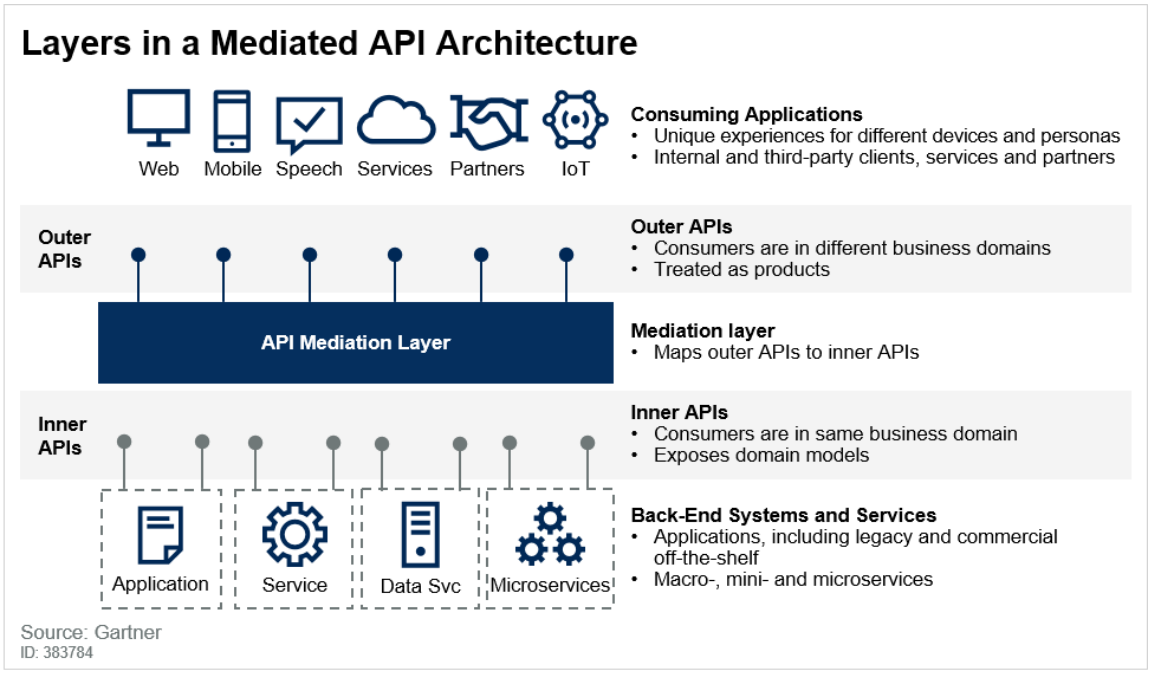
10.1 נספח א' – ארכיטקטורה של שכבות API

10.2 נספח ב' - ארכיטקטורה של שכבות API במבט של מספר מערכות ומספר סוגי שירותים

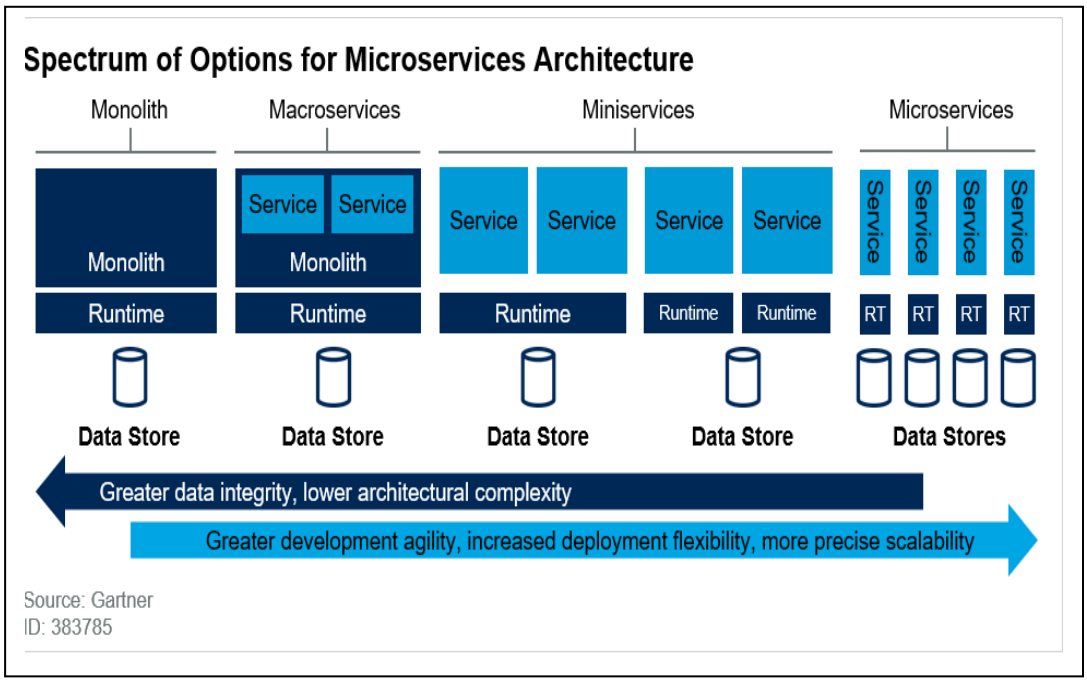
10.3 נספח ג' - ארכיטקטורה מונחית אירועיםEVENT DRIVEN ARCHTECTURE

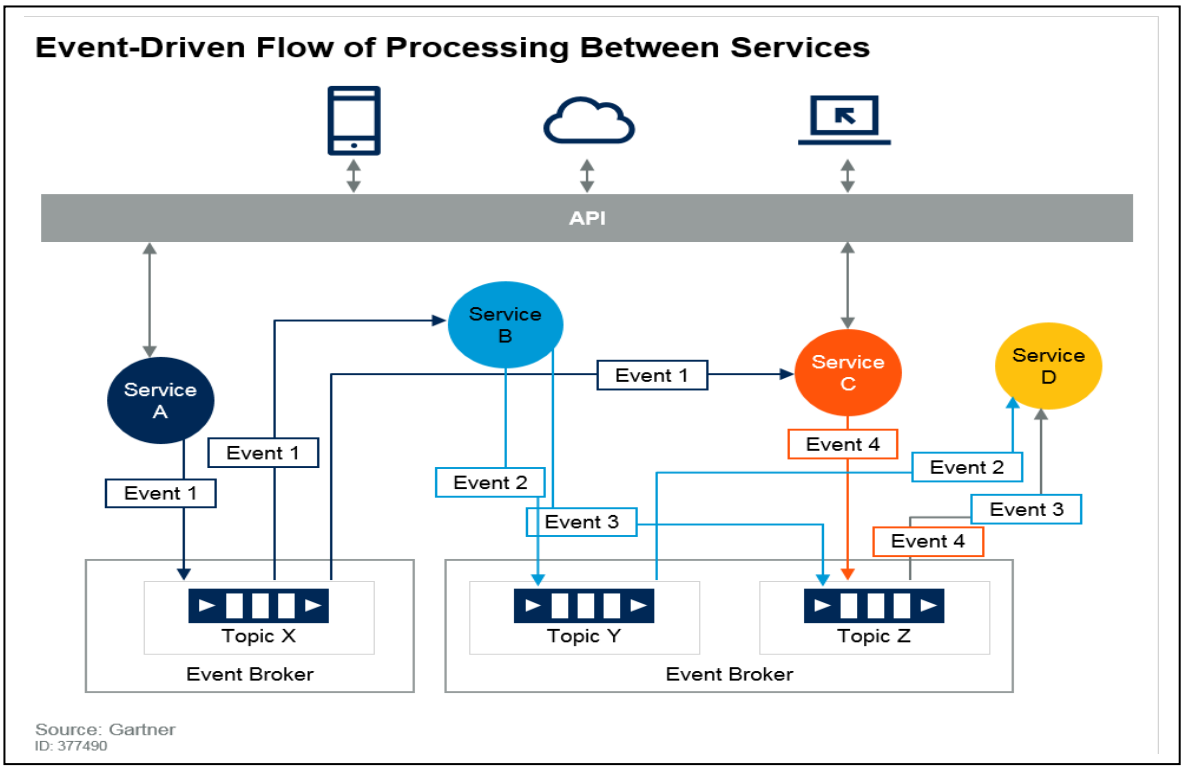
11. גרסאות ההנחיה

מס'	סטאטוס	מהות שינוי	סעיפים שהושפעו	בתוקף מ-	נכתב ע"י	אושר ע"י
1.0	בוטל	גרסה ראשונה		10.11.2020	קרן בר-לב	שחר ברכה
2.1	הופץ	עדכון המסמך	בעיקר סעיפים: 1, 5, 6.	22.5.2024	קרן בר לב	קרן בר לב


Layers in a Mediated API Architecture
Web
Mobile
Speech
Services
Partners
IoT
Consuming Applications
• Unique experiences for different devices and personas
• Internal and third-party clients, services and partners
Outer APIs
Outer APIs
• Consumers are in different business domains
• Treated as products
API Mediation Layer
Mediation layer
• Maps outer APIs to inner APIs
Inner APIs
Inner APIs
• Consumers are in same business domain
• Exposes domain models
Application
Service
Data Svc
Microservices
Back-End Systems and Services
• Applications, including legacy and commercial off-the-shelf
• Macro-, mini- and microservices
Source: Gartner
ID: 383784

נספח ב' – ארכיטקטורה של שכבות API במבט של מספר מערכות ומספר סוגי שירותים




Event-Driven Flow of Processing Between Services
API
Service A
Service B
Service C
Service D
Event 1
Event 1
Event 1
Event 2
Event 2
Event 3
Event 3
Event 4
Event 4
Topic X
Topic Y
Topic Z
Event Broker
Event Broker
Source: Gartner
ID: 377490