# עיצוב תוכנה

## עיצוב תוכנה

עיצוב זהו תכנון ופיתוח פתרונות בכפוף לסט אילוצים. עיצוב תוכנה זהו מודולציה של הקוד בצורה נכונה. נחלק ל-2 רמות:

מיקרו – שימוש בתבניות עיצוב כמו strategy, observer, וכו'. משמשות כפתרון לבעיות שחוזרות על עצמן פעמים רבות.

מאקרו – שימוש בתבניות ארכיטקטוניות. זהו עיצוב תוכנה במרכיבים הקריטיים שלה.

ההבדל בין ארכיטקטורה לעיצוב זה ששינוי ארכיטקטורה זו החלטת עיצוב מאוד יקרה וארוכה שמשנה את הפרויקט בצורה מהותית, ואילו שינוי בעיצוב זהו שינוי מהיר ולא יקר. החלטות ארכיטקטוניות יש לקבל מראש, לפני כתיבת התוכנה, ועם חשיבה מרובה.

## עקרון S.O.C

לפי עקרון S.O.C (Separation Of Concerns) יש להפריד לחלקים (מודולים) את התוכנית כך שלכל חלק יש תפקיד אחד מוגדר על נושא יחיד. בין מודולים צריך שיהיה צמידות נמוכה ולכידות גבוהה.

צמידות נמוכה – כל מודול משמש כיחידה עצמאית ותלוי כמה שפחות במודולים אחרים במערכת. כלומר, אם משנים אחד אין צורך לשנות את השני. כאשר יש תלות בין מודולים יש לנהל תלות זו באמצעות ממשקים מצומצמים ואירועים.

לכידות גבוהה – בתוך מודול צריך להיות קשר חזק בין הפעולות השונות שנמצאות בו. במידה ואין קשר יש לפצל אותו למספר מודולים. נדע שיש לנו מודול מפוקס אם נוכל לתת לו הגדרה אחת.

## עקרונות S.O.L.I.D

עקרונות SOLID הם חמישה עקרונות בסיסיים בעיצוב מונחה עצמים. המחשבה מאחורי העקרונות היא שכאשר הם מיושמים יחדיו בפיתוח של מערכת תוכנה, היא תהיה יותר קלה לתחזוקה והרחבה לאורך הזמן.

באופן עקרוני, קוד גרוע הוא קוד שבו: כל שינוי משפיע על הרבה חלקים בקוד, שינוי בקוד משפיע על אזורים לא קשורים בקוד, קוד לא פריק - לא ניתן להשתמש בקוד שכבר כתבנו בהקשרים אחרים מאלו שלשמם נכתב הקוד במקור. האופי המרכזי של הבעיות האלו הוא יותר מידי תלות בתוך הקוד. עקרונות SOLID באים לתת קווים מנחים שיגרמו לנו להימנע מלכתוב קוד עם הבעיות הנ"ל.

**Single responsibility**

לכל מחלקה צריך להיות תחום אחריות אחד עליו היא אחראית בצורה מלאה. טעות מאוד נפוצה היא שכאשר בונים מחלקה שמייצגת אובייקט כלשהו, אזי מוסיפים לה שיטות שלא בהכרח קשורות לאובייקט אך במבט ראשוני נראה שנוח אם האובייקט יוכל לבצע אותן. פעמים רבות שיטות אלו מאוד מגבילות ודורשות שינויים מהותיים כאשר מרחיבים או משנים משהו במחלקה. לדוגמא, עבור אובייקט שמייצג משתמש, לא נרצה שתהיה שיטה שמקבלת אימייל ובודקת אם הוא תקין. במקום זה נבנה מחלקה מיועדת שבודקת מידע.

באופן עקרוני, כדאי לפצל מחלקות למחלקות קטנות יותר כאשר זה אפשרי. מספר רב יותר של מחלקות מקנה לתוכנית רמה גבוהה יותר של מודולאריות. חשוב להשתדל לעבוד בכיוון זה.

**Open for Extension, Closed for Modification**

מחלקה צריכה להיות פתוחה להוספות וסגורה לשינויים. כלומר, שהוספת שיטות למחלקה לא ידרוש מאיתנו לשנות קוד שכבר כתבנו. גם הוספה של מחלקות אחרות לתוכנית לא אמור לשנות מחלקות שכבר כתבנו. נעשה זאת בעזרת תכנון נכון של המחלקה ושימוש נכון בממשקים.

**Liskov substitution principle**

פונקציות המשתמשות במשתנים מסוג מחלקת אב, חייבות להיות מסוגלות לפעול בצורה תקינה גם על כל סוגי האובייקטים מסוג הבן, מבלי להיות מודעות לסוג האובייקט בפועל. עצה לכך היא שכאשר כותבים מחלקה שיורשת ממחלקה אחרת או מממשת ממשק, אזי תמיד נסתכל על המחלקה שאנו כותבים כאל אובייקט מסוג מחלקת האב או הממשק עם תוספות קלות. באופן כללי, תמיד כדאי להשתמש בממשקים ולא בירושה.

**Interface segregation principle**

אין להכריח לקוח להיות תלוי בממשק שהוא אינו משתמש בו באופן מלא. יש לדאוג לממשקים מצומצמים, כך שיאלצו מחלקה שמממשת את הממשק לממש רק מה שרלוונטי לה. פתרון לבעיה זו הוא ממשקים שמממשים ממשקים.

**Dependency inversion**

מחלקות high level לא צריכות להשתמש באופן ישיר במחלקות low level, כלומר מחלקות המתארות מקרה כללי לא צריכות להשתמש במחלקות המתארות מקרה פרטי. הסיבה לכך היא שאם בשלב כלשהו נרצה להוסיף עוד מחלקה, המייצגת מקרה פרטי לתוכנית שלנו, נצטרך לשנות גם את המחלקה הכללית. הפתרון לבעיה זו היא ליצור ממשק, המייצג גם כן מקרה כללי, שאותו יממשו המחלקות המתארות מקרה פרטי. כעת במחלקת ה-high level ממנה התחלנו, נוכל להשתמש בממשק זה במקום. כל הוספה של מחלקה המייצגת מקרה פרטי תממש את הממשק.

## תוכנה

אוסף של שירותים דיגיטליים. נחלק לשלושה סוגים:

1. שירותים כללים ובסיסיים. בדרך כלל חלק ממערכת ההפעלה.
2. אפליקציות: אוסף של שירותים יעילים למשימה ספציפית.
3. שירותים המשמשים תשתית לפיתוח אפליקציות.

## ביזור

חלוקת מערכת לתת מערכות. מערכת שרצה או משתמשת בכמה מקומות נקראת מערכת מבוזרת. כל סוג של תת-מערכת נקראת tier. דוגמה נפוצה לביזור היא 3-tier שבה יש שרת מחשב, שרת נתונים ומשתמש. שני סוגים של ביזור:

חישוב מבוזר – חלקים שונים של החישוב מתבצעים במקביל במספר מערכות המקושרים ביניהם באמצעות רשת.

אחסון מבוזר – שמירת נתונים על קבוצת שרתים ברשת.

שני דרכים כיצד לשפר מערכת תוכנה ללא שינוי בקוד:

Scale up – שיפור החומרה של המחשב – זיכרון, מעבד וכו'. שיפור כזה הוא מאוד מוגבל כי יש גבול כמה שיפורים אפשר לעשות במחשב אחד. זהו non-linear scale.

Scale out/over – פיצול התוכנה למספר מחשבים כאשר יש load balancer שמתכנן עומסים. משפר ביצועים בצורה טובה יותר. זהו linear scale.

## CI/CD

CI/CD – (Continuous delivering/ Continuous integrations) פעילות שבה מתקנים באגים או מוסיפים תכונות חדשות באופן מידי. אנו רוצים תמיד את הגרסה הכי מעודכנת, לכן מעדכנים על בסיס יומי. זוהי עבודתם של מהנדסי devOps. הם בונים צינור שמכיל עדכונים מסביבת הפיתוח (development) להפצה (operations).

MLOps או DataOps – אנשים שיודעים להעביר raw data ולנתח ולהעביר אותו לרמה שניתן לתפעל אותו.

## ארגון הקוד - תבנית MVC:

Controller – מקבל את כל הבקשות. הוא מפנה לקוד שנקרא model כדי לקבל ממנו דאטה. את מה שקיבל מהמודל מפנה ל-view שיחזיר לו קוד בצורת HTML שניתן להציג למשתמש. את התשובה הסופית מחזיר חזרה ללקוח.

Data Model – מתפעל את הדאטה. מחזיר את הדאטה הנדרש ממנו.

View – מקבל מידע להציג וממיר אותו ל-HTML מעוצב יפה.

בפרויקט גדול לכל משימה יש תבנית MVC נפרדת. כדי לשלוט בכל המשימות יוצרים front controller שיקבל את כל הבקשות ויפנה ל-controller המתאים.

# שירותי תוכנה בענן

## מחשוב ענן

הגדרה 1: מודל מחשוב המשלב ביזור, עלויות תפעול נמוכות וניצול משאבים עם יכולות חוצות פלטפורמות, המאפשר ללקוחות שירותי מחשוב ישירות ללא פעולות תחזוקה.

הגדרה 2: שם גג לצורת הפצה של שירותי תוכנה המבוססים על רשתות ומרכזי נתונים.

הממשק שאיתו מתקשרים עם תוכנה בענן נקרא web API. שולחים בקשה שמתאימה לדרישות ה-API ומקבלים תשובה, בדרך כלל בשפת Jason.

## סוגי שירותי תוכנה בענן

### IaaS – Infrastructure as a Service

מספקים שירותי מחשוב כמו מכונות וירטואליות, אחסון, נתבים וירטואליים. לדוגמה Azure, AWS.

### Paas – Platform As A Service

מספקים פלטפורמה להרצה או פיתוח אפליקציות. המשתמש מקבל משאבי מחשוב וגם תוכנות תשתית הנצרכות לצורך הרצת או פיתוח המערכת כמו שירותי ענן, אחסון וניהול נתונים, שירותי מסרים ושירותי ניהול משתמשים וזהויות. לדוגמה Google App Engine.

ההבדל בין IaaS ל-PaaS, שב-IaaS יש יותר גמישות לעשות מה שאנו רוצים. אפשר לדמות זאת לכך שב-IaaS מקבלים חומרים לבנות בית וב-PaaS מקבלים בית מוכן וצריך רק לרהט.

### SaaS – Software as a Service

מספקים למשתמש שירותים הניתנים באמצעות אתר הספק. לדוגמה Office 365, Google Docs.

### XaaS - Anything as a Service

כל השירותים דרך חישוב ענן. יכולים להיות עיבוד או אחסון.

## Serverless computing

לתכנת דברים בלי לדעת שפונים לשירותים בענן. משתמשים בענן בלי לדעת ואז גם לא מקלקלים. רק מי שבונה את המערכת הבסיסית הוא יודע איך היא בנויה ובדך כלל זו תהיה קבוצה קטנה יותר של מפתחים. נקרא גם lambda functions.

## Microservice architecture

סגנון של ארכיטקטורת תוכנה שבה אפליקציות מורכבות מיחידות קטנות ועצמאיות שמתקשרות ביניהן באמצעות language agnostic API. אם מחלקה מקבילה לשירות אזי פונקציה מקבילה ל-microservice. ההפך הגמור מאפליקציה מונוליטית שבנויה כיחידה אוטונומית אחת ושבה כל שינוי משפיע על כל התוכנית ועלול להוביל לגרסה חדשה.

### יתרונות:

* מאורגן טוב יותר וקל להבנה.
* קל לעדכן והתמודדות עם שגיאות.
* יכול להיות מפותח ע"י קבוצה קטנה.
* קל לבצע אינטגרציה בין יחידות וצד שלישי.
* ניתן לפתח כל יחידה בשפה שונה.
* נוח יותר בשביל scaling.

### חסרונות:

* יותר שגיאות בשל עיכובים ברשת.
* קשה לבצע בדיקות.
* קושי בניהול.
* צריך להגדיר אינטגרציה בין היחידות שיכולה להיות מורכבת.

## SOA

Service Oriented Architecture היא ארכיטקטורה של מחשוב ארגוני שמתבסס על שירותים. במקום שכל אחד יחשוף את השרות שלו משתמשים ב-ESB (Enterprise service bus) המשתמש מתווך בין הלקוחות לשירותים. בכל פעם שמקבל בקשה מפנה אותה לשירות שייתן תשובה. ESB הוא כמו ממשק ולכן הוא צריך להיות עם צימוד נמוך.

# תקשורת ופרוטוקולים

## REST

ארכיטקטורה למערכות רשת. כל מי שתומך בREST נקרא REST FULL. הרעיון העיקרי הוא הגדרת משאב מערכת שמצבו עובר שינוי כתוצאה מהאינטראקציה בין מספק השירות והצרכן. ארכיטקטורת REST דורשת 5 כללים:

* 1. לכל יחידת קצה חייב להיות מזהה ייחודי ID.
  2. חייבים לחבר את כל היחידות יחד.
  3. התקשורת היא באמצעות סט סטנדרטי של פקודות המחייבים את השרת והלקוח.
  4. המערכת צריכה לדעת לעבוד עם סוגים שונים של מידע (גרף, טבלה, XML וכו').
  5. Stateless, ההקשר (context) של הלקוח בפנייה לשרת, אינו נשמר בשרת. הפרוטוקול מטפל בכל בקשה בנפרד, כך שהתקשורת מורכבת מזוגות נפרדים של בקשות ותגובות. מגבלה זו נועדה לשפר את מדרגיות (Scalability) השרת כיוון שלא צריך לזכור הרבה מידע. כן אפשר לזכור תשובות קודמות במטמון במחשב הלקוח.

יחידות קצה מדברות בעזרת REST API.

## HTTP

פרוטוקול HTTP (Hypertext Transfer Protocol) הוא פרוטוקול תקשורת שנועד להעברת דפי HTML ואובייקטים שהם מכילים (כמו תמונות, אודיו, סרטונים וכו') ברשת האינטרנט וברשתות אינטראנט. הפרוטוקול פועל בשכבת האפליקציה, בתקשורת "לקוח-שרת" (client-server). לקוח HTTP הוא הדפדפן, ושרת HTTP הם שרתי ה-web שהם שרתי התוכן המרכזיים ברשת האינטרנט. הדפדפן מבקש, מקבל ומציג, והשרת שולח אובייקטים בתגובה לבקשות. מבוסס REST.

התקשורת בין הלקוח לשרת היא באמצעות הודעות בפורמט סטנדרטי. לכל הודעת HTTP יש כותרת (header) וגוף (body). צורת העבודה של HTTP היא request and response. כאשר רוצים קשר רציף בין שני נקודות הקצה משתמשים ב-web socket או ביצוע refresh כל פרק זמן קבוע או AJAX או SSE.

מעצם הגדרתו, פרוטוקול HTTP הוא stateless protocol - חסר מצבים, כלומר אינו שומר מידע על בקשות קודמות של הלקוח. על מנת ליצור תקשורת בין הלקוח לשרת שמבוססת על היסטוריית הבקשות-תשובות שלהם, נעשה שימוש בעוגיות (cookies). לדוגמא, שרת יכול לשתול במחשב הלקוח עוגייה עם אישור שהלקוח התחבר לחשבון מסוים עם סיסמה נכונה, על מנת שלא יצטרך להקיש סיסמה בכל התחברות מחדש לאתר שמתארח על השרת.

## פרוטוקול Web Socket

Web Sockets היא דרך לתקשורת ברשתות תקשורת. עבור מפתחי ווב – הכוונה היא לתקשורת בין שרת לבין לקוח (client) כאשר הלקוח יכול להיות דפדפן או שרת אחר, אבל ברוב המקרים יש לנו שרת מרוחק של אתר כלשהו ו-JavaScript שמתקשר עם השרת הזה באמצעות web socket ו-API מיוחד שהוטמע בכל הדפדפנים המודרניים שתומכים ב-HTML 5.

מה ההבדל בין Web Sockets ל-AJAX? עם AJAX אנחנו יוצרים בקשה בצד הלקוח אל צד השרת (צד השרת לא יכול ליזום בקשה). בנוסף, כל בקשה כוללת את ה-headers ועטופה במידע. מה שהופך את ה-AJAX לבזבזני. Web Sockets לאחר שהלקוח מקים את החיבור דרך תהליך Web Socket Handshake, נפתח לנו קשר דו כיווני – כלומר גם השרת יכול לדחוף מסרים משלו אל הלקוח. בנוסף כל מה שעובר דרך ה-Web Socket לא כולל headers ו-junk אלא פשוט עובר. לא מדובר בבקשה-תגובה כמו ב-AJAX אלא בקשר שנשאר פתוח.

## Docker

Docker היא תוכנה בקוד פתוח המספקת שכבת הפשטה להתקנה והרצה של יישומים בתוך קונטיינרים (מכולות) בעזרת וירטואליזציה ברמת מערכת ההפעלה, בניגוד למכונות וירטואליות (VMs), שהן בדרך כלל בעלות תקורה גבוהה יחסית של הקמה ותחזוקה.

הרעיון המרכזי הוא לבדוק אם התוכנה עובדת רק עם הדרישות המינימליות ללא שום תלות במערכות אחרות שקיימות אצל המפתחים. דוקר משתמש במינימום שהוא צריך ממערכת ההפעלה בעוד מכונה וירטואלית לא יודעת שהיא משתמשת במערכת ההפעלה. היתרון על פני מכונה וירטואלית הוא שקונטיינר עולה תוך שניות בעוד שמכונה וירטואלית היא איטית הרבה יותר לפעולה. קונטיינר מוגדר בקלות באמצעות קובץ הגדרות בעוד שמכונה וירטואלית לא.

באמצעות שימוש בקונטיינרים ניתן לפשט יצירה של מערכות מבוזרות ולתת למשתמש אפשרות עבודה עם יישומים מרובים, משימות עובדים ותהליכים, ולנהל את כל סביבת העבודה באופן עצמאי על מכונה פיזית אחת או על פני מספר רב של מכונות וירטואליות.

### רכיבים:

Image – תבנית עם הוראות ליצירת קונטיינר.

Container – מכונה וירטואלית שיש בה את מה שאנחנו רוצים ומוגדרת כבר עם מערכת קבצים שמתממשקת, פורט משל עצמה וכו'. ניתן להריץ, לעצור ולמחוק.

Engine – הכלי שמייצר ומריץ את הקונטיינר.

Registry – מאחסן images. לדוגמא docker hub.

## Middleware

זהו קוד שמופעל בין הלקוח לשרת. אנו מפעילים קוד זה על כל הבקשות שמגיעות מהלקוח לפני שמגיעות לשרת. בחלק מהמקרים על הבקשות מפעילים שרשרת של יחידות עיבוד הנקראת pipes and filters. היתרון שבכל זמן נוכל לבחור איזה יחידת עיבוד להוריד או להוסיף בכל מקום בשרשרת.

כדי ליצור middleware יוצרים פונקציה שמקבלת בקשה, תגובה ו-next. הפונקציה תבצע את הקוד ביניים שאנו רוצים. ותסיים בלקרוא לפונקציה next. לאחר מכן מגדירים לאפליקציה להשתמש בפונקציה זו.

App.use(middlewear)

ניתן לשרשר פונקציות כאלו על ידי use.

# מסדי נתונים

מסד נתונים או בסיס נתונים (Database, או בקיצור DB) הוא אמצעי המשמש לאחסון מסודר של נתונים במחשב, לשם אחזורם ועיבודם. הגישה לבסיס הנתונים נעשית באמצעות תוכנה ייעודית - מערכת לניהול בסיס נתונים (DBMS - DataBase Management System).

## NoSQL

קבוצה של מערכות לניהול מסדי נתונים שאינן מסתמכות בלבד על אחסון הנתונים בטבלאות (רלציות) כמו מסדי נתונים רלציוניים. נובעות מהצורך לטפל בדאטה שהולך וגדל ואינו מסודר במבנה מסוים. מקור השם הוא Non SQL, אך נקרא לעיתים Not Only SQL, כדי להדגיש את העובדה שהן עשויות לתמוך בשאילתות הדומות ל-SQL.

מסדי נתונים מסוג NoSQL יותר גמישים בשמירת עקרונות ה-ACID, ביחס למסדי נתונים רלציוניים. הם מתוכננים עבור מערכות מבוזרות שבהן המידע מתפרש על כמה שרתים, ומשמשים בעיקר למערכות של ביג-דאטה או מערכות זמן אמת.

### יתרונות

1. Scalability - ניתן בקלות להרחיב את כמות המידע המאוחסן. יש שני דרכים לעשות זאת: על ידי הוספת משאבים לשרתים הקיימים (Vertical scaling) או על ידי הוספת שרתים (horizontal scaling).
2. Availability - המידע במסדי נתונים של NoSQL משוכפל למספר שרתים, כך שאם שרת אחד או יותר נופלים נוכל לשחזר את המידע משרת אחר. תכונה זו הופכת אותו לזמין ביותר, מכיוון שבכמעט כל מצב נוכל לגשת למידע.
3. Speed - למערכות מבוזרות יש יותר כוח חישוב מאחר וכל שרת יכול לבצע חישובים בפני עצמו - חישוב מקבילי.
4. Flexibility - ניתן להוסיף תכונות מהר תוך כדי ריצה. לא מקפידים על עקביות.

### חסרונות

1. Narrow focus - מיועד בעיקר לאחסון ולכן מספק מעט מאוד פונקציונליות.
2. No standardization - אין סטנדרט אחיד למסדי נתונים של NoSQL.
3. Management challenge - ניהול נתונים ב-NoSQL הרבה יותר מורכב ממסד נתונים רלציוני. יש לו מוניטין שהוא מאתגר להתקנה ואפילו קדחתני יותר לניהול על בסיס יומי.
4. GUI - לרוב מסדי הנתונים מסוג NoSQL אין GUI נוח ופשוט.

## סוגים של NoSQL

מבני הנתונים בהם משתמשים מסדי הנתונים השונים ב-NoSQL שונים אחד מהשני, אמנם ניתן לחלק אותם למספר קטגוריות. הבחירה של מסד נתונים תלוי בבעיה שאותה אנו רוצים לפתור.

1. **Key-Value Stores** - מסד נתונים פשוט ומהיר, המאחסן נתונים בצורה של מפתח-ערך. עובד כמו Hashtable ענק. השליפה ממסד הנתונים תמיד תהיה לפי המפתח, כלומר נכניס את המפתח ונקבל את הערך או ערכים המתאימים לו. אין סדר על הנתונים, ניתן להכניס כל סוג מפתח שיצביע לכל סוג ערך, כמו XML, Json, תמונות, וידאו, וכו'. מסד נתונים זה הוא פשוט ומהיר. דוגמה: Redis (הנפוץ ביותר), ‏Riak, ‏Voldemort.
2. **Wide-Column Stores** - משתמשים בטבלאות, שורות ועמודות כדי לאחסן את המידע, כמו במסדי נתונים רלציוניים, אלא שכאן לכל שורה יכולים להיות עמודות שונות, אפילו באותה טבלה. משמשים בעיקר לשאילתות על מערכי מידע גדולים במיוחד הניתנים לשינוי מהיר. דוגמה: Cassandra.
3. **Document Databases** - בסיס נתונים המצמיד מפתח למבנה מורכב של מידע הנקרא "מסמך". מסמך יכול להכיל צמדים מורכבים של מפתח-ערך, מפתח-מערך, או אפילו מפתח-מסמך. מסמכים אלו בדרך כלל יהיו קבצי XML או JSON, או קבצים מסוג דומה. בניגוד למסדי נתונים מסוג Key-Value, כאן ניתן לבצע שאילתה לא רק על המפתחות אלא גם על המסמכים ובערכים שבתוך המסמכים. דוגמה: MongoDB, המסד NoSQL הנפוץ ביותר.
4. **Search Engine Databases** - כמו מסדי נתונים מסוג Document שבהם כל מפתח מזווג למסמך. אלא שהם מוסיפים לכל מסמך שמוחזר משאילתה, דירוג (scoring) כמה הוא מתאים למה שחיפשנו בשאילתה. מסדי נתונים אלו, כמו שמציין השם שלהם, מתאימים בעיקר למנועי חיפוש, שבהם אנו רוצים שיופיעו קודם התוצאות הכי רלוונטיות לחיפוש שלנו. דוגמה: Elastic Search.
5. **Graph Stores** - מאחסנים את המידע בצורה של גרף, עם צמתים וקשתות. מסדי נתונים אלו מאפשרים שאילתות מסוג של "מצא את כל החברים" של אובייקט כלשהו, את החברים שלהם, וכן הלאה עד רמת קרבה 10. משמשים בעיקר לשמירת מידע הקשור לרשתות וקשרים חברתיים. דוגמה: Neo4J, Jena.

## Transaction

טרנזקציה היא רצף של בקשות לשינוי נתונים שמבוצעות כולן כיחידה אחת או לא מבוצעות כלל. באמצעות טרנזקציות אנו יכולים להבטיח שאנו שומרים על עקביות ובכך להבטיח שאין שגיאות לא צפויות במסד הנתונים. לדוגמה, עבור רצף של בקשות שמבצעות העברה בנקאית מחשבון אחד לחשבון אחר, אם תהיה שגיאה באמצע יכול להיווצר מצב שבו נמשך הכסף מחשבון אחד אך לא הועבר לאחר. בעיה נוספת אם באותו זמן ירוץ עוד תהליך שמושך כסף מהחשבון וידרוס את השינוי שעשינו. הפתרון לבעיות אלו הוא להצהיר על רצף הבקשות אלו כטרנזקציה, ובכך יבוצעו כולם יחד או לא יבוצעו בכלל. במידה ויש שגיאה באמצע טרנזקציה כל הפעולות שכבר בוצעו יבוטלו באמצעות rollback.

מדד TPS - כמות הטרנזקציות המבוצעות בכל שנייה.

## תאוריית CAP

קובעת כי עבור מאגר נתונים מבוזר אי אפשר להבטיח יותר משתיים מבין שלוש התכונות הבאות:

* **C**onsistency - בכל קריאה מבסיס הנתונים מתקבל המידע העדכני ביותר שנכתב או שגיאה. כל המשתמשים רואים את אותו המידע, ולכן כל פעולה על מסד הנתונים היא על הנתונים העדכניים ביותר.
* **A**vailability - כל בקשה מקבלת תגובה (לא בהכרח העדכנית ביותר).
* **P**artition tolerance - המערכת ממשיכה לתפקד אפילו כשהתקשורת בין השרתים אינה אמינה, כלומר השרתים עשויים להיות מחולקים למספר קבוצות שאינן יכולות לתקשר זו עם זו.

משפט CAP מוכיח שאי אפשר למלא את כל שלושת הדרישות אלא לכל היותר 2. לכן כל מסד נתונים של NoSQL עוקב אחר השילובים השונים של C, A, P מתוך משפט CAP. כאשר יש בעיה בהעברת הנתונים בין מאגרי המידע ישנן שתי אופציות כיצד להגיב לקריאה מהמערכת. או אישור הבקשה, מה שמבטיח זמינות אבל פוגע בעקביות. או דחיית הבקשה על מנת לשמור על עקביות אך גורם לפגיעה בזמינות. מסדי נתונים רבים ב-NoSQL בוחרים להתפשר על העקביות ולהרוויח זמינות וסובלנות לחלוקה.

## עקרונות ACID

המונח ACID הוא ראשי תיבות של: Atomicity, Consistency, Isolation, Durability. תרגום המונחים לעברית הוא אַטוֹמִיוּת, עִקְבִיּוּת, בִידוּד ועֲמִידוּת. תכונות אלה הן אבן הפינה של מסדי נתונים ומערכות לניהול תנועות, ובלעדיהן לא ניתן להבטיח את שלמות הנתונים במערכות אלה. אמנם בפועל, תכונות ה-ACID נאכפות במידה רופפת יותר כדי לשפר את ביצועי המערכת.

* **אטומיות (Atomicity)** - כל רצף פקודות מבוצעות בבת אחת, כך שאם המערכת תקרוס באמצע ביצוע הפעולות לא יהיה איבוד מידע. כלומר, כל רצף פקודות מתבצעות במלואן או לא מתבצעות כלל.
* **עקביות (Consistency)** - הבסיס נתונים תמיד נשאר עקבי, אם מנסים לבצע פעולה שתכניס את הבסיס נתונים למצב לא חוקי מבחינתו (השמה של מחרוזת במקום מספר, השמה של מספר שלא זהה לחוק שהוגדר עבור עמודה), אזי בסיס נתונים לא יאפשר ביצוע של אותה הפעולה.
* **בידוד (Isolation)** - בסיס הנתונים מאפשר לבצע פעולות רבות במקביל, כל עוד התוצאה זהה לביצוע הפעולות באופן טורי. המשמעות היא שלא ניתן לגשת לאותו תא משני מקורות במקביל (לדוגמה, משיכת סכום של שני משתמשים שונים מאותו חשבון בנק במקביל).
* **עמידות (Durability)** - פעולה שמתבצעת תמיד נשמרת בבסיס נתונים, אפילו אם היה תקלה באמצע כמו הפסקת חשמל וכו'. פעולות לא הולכות לאיבוד.

## עקרונות BASE

מסדי נתונים רבים ב-NoSQL בוחרים להתפשר על העקביות ולהרוויח זמינות וסובלנות לחלוקה. בכך הם לא שומרים על עקרונות ה-ACID, ומציעים במקום את עקרונות BASE:

* **B**asically **A**vailable - הנתונים זמינים לרוב.
* soft **S**tate - המצב עשוי להשתנות אפילו ללא עדכונים (מכיוון שעדכונים ישנים עדיין מתפשטים).
* **E**ventual consistency - אם נאפשר לנתונים להתפשט מספיק זמן, הם לבסוף יהפכו עקביים.

# ביג דאטה

## הגדרה

מונח המתייחס למאגר מידע הכולל נתונים שאינם מאורגנים לפי שיטה כלשהי, שמגיע ממקומות רבים, בכמויות גדולות, בפורמטים מגוונים ובאיכויות שונות.

### ההבדל בין Small Data (SD) לעומת Big Data (BD):

* + מטרה – נאסף למטרה מוגדרת אחת (למערכות התפעוליות) לעומת אבולוצית מטרות עם הזמן (למערכות האנליטיות, קודם תשמור את הנתונים, אח"כ נראה מה נעשה איתם. יתכן שישמשו ללמידת מכונה וקבלת החלטות).
  + מקום – אחד בפורמט אחיד לעומת מבוזר על גבי מספר פלטפורמות ופורמטים.
  + מבניות – בד"כ מובנה לעומת מעורב.
  + אורך חיים – נאגר ורלוונטי למספר שנים ואח"כ יורד מבסיס הנתונים לגיבוי, לעומת מצטבר לאורך זמן לא ידוע (לדוגמה ברפואה, כי יתכן שההשלכות של שימוש תרופה/טיפול רפואי אותם מנסים לגלות באנליטיקה, מתרחשות לאחר שנים רבות).
  + יחידות מידה – אחיד ומוגדר לעומת רבגוני.
  + שחזור – ניתן ליצירה מחודשת לעומת לא ניתן.
  + אינטרוספקציה – נתונים המתארים עצמם בבהירות (מיהו האובייקט הנמדד, מה המימד ומהו הערך) לעומת מגוון, חזרות, כפילויות וחוסרים.
  + יכולת ניתוח – בבת אחת לעומת צורך להכין, לצמצם, לפצל, להמיר. התהליך הכי קשה בביג דאטה זה להכין את הנתונים ל-preprocessing.
  + ניתוח Small Data אפשרי באמצעות כלים רגילים, ניתוח Big Data מצריך כלים מיוחדים כמו למידת מכונה.

## מודל הV-ים לתיאור אתגרי ביג דאטה

יש ארבעה בעיות שאם אנו נתקלים בהם אנו יודעים שהבעיה שלנו היא של ביג דאטה:

* Volume – כמות גדולות של מידע לעיבוד וניתוח. צריך הרבה זיכרון.
* Velocity – מידע זורם בקצב מהיר. צריך לספק תוצאות בזמן מהיר.
* Variety – מידע לא מובנה שלא ניתן להכניס לטבלה רגילה. נאסף ממקורות שונים בפורמטים שונים.
* Veracity – איכות ואמינות הנתונים אינה וודאית. יכולים להיות שכפולים או מידע רב חסר.

## מודל DIKW

Data – עובדות ונתונים לא מסודרים מהעולם.

Information – סידור הדאטה באיזשהו מבנה כדי שנוכל למצוא קשרים.

Knowledge – ניתוח המידע ומציאת תבניות.

Wisdom – חיבור כל הקשרים והתבניות יכול להוביל אותנו לתובנות מה נכון ומה לא.

כיצד אוספים ידע? דרך אחת היא בעזרת מומחה ליצור אונטולוגיה שהיא אוסף של מושגים והיחסים ביניהם. דרך אחרת לאסוף נתונים ולעבד אותם באמצעות טכניקות ניתוח ולמידה.

## שלבי kdd (knowledge discovery in databases):

1) data cleaning – ניקוי רעשים מהמידע שאספנו.

2) data integration – איחוד מקורות שונים של מידע.

3) data selection – בחירת המידע הקשור לחיפוש שאנו רוצים לבצע.

4) data transformation – המרת הנתונים לצורה נוחה לשימוש עבור הניתוח.

5) data mining - הפעלת פעולות אשר מפיקות דפוסי נתונים.

6) הערכת המידע - מבצעים הערכה על הדפוסים שקיבלנו.

7) הצגת המידע – אפשר בקלסטרים, עץ החלטה וכו'.

## Data Pipeline

זוהי תוכנה שמטרתה לאפשר זרימה חלקה של מידע בין שתי תחנות. מגדירה מה, איפה ואיך נאספים הנתונים ומבצעת אוטומציה לתהליכים כגון: חילוץ, טרנספורמציה, אימות, וטעינה של המידע להמשך ניתוח וויזואליזציה. מספקת נוחות ומהירות על ידי ביטול שגיאות וטיפול בצווארי בקבוק.

## Hadoop

פלטפורמת קוד פתוח לעיבוד מידע בנפחים גדולים. היתרונות הגדולים שהופכים אותה למאוד פופולרית הם שמאפשרת מספר שיטות לעיבוד מקבילי, יכולה להתמודד עם כמות הולכת וגדלה של נתונים (scalability), גמישה וחינמית. מורכבת משני חלקים: HDFS ו-Map Reduce.

### HDFS - Hadoop Distributed File System

מערכת קבצים המסוגלת לשמור מידע רב באמינות ושרידות גבוהה. מחלקת את המידע לבלוקים ושומרת אותם ב-nodes. לכל בלוק יש כמה עותקים כך שאם node אחד נפל או שאינו זמין יש node אחר שממנו אפשר לשלוף את הבלוק.

### Map Reduce

זוהי שיטת עיבוד שבה מריצים מספר אלגוריתמים במערכת מבוזרת המאפשרת לבצע חישוב מקבילי על כמות גדולה של מידע. כל תבנית מחולקת לשלושה שלבים, שכל שלב הוא אלגוריתם אחר:

1. Map - האלגוריתם הראשון תפקידו לסנן את המידע ולקחת רק את החלקים עליהם צריך להפעיל פעולת חישוב. בנוסף מבצע מיפוי על המידע הנותר ל-key ו-value, כך שנוכל לגשת אליו בצורה נוחה.
2. Sort/Grouping - מפרק את המידע לקבוצות הגיוניות, ומקצה לכל worker קבוצת מידע.
3. Reduce - כל worker מפעיל אלגוריתם זה על המידע שקיבל ומחזיר את התוצאה.

בחלק מהמקרים איחוד המידע שמוחזר מכל worker הוא עצמו התשובה, אמנם ישנם מקרים בהם צריך אלגוריתם נוסף שיעבד את כל המידע שהתקבל ויחזיר את התשובה. שלב זה נקרא finalize.

דוגמה לשימוש ב-Map Reduce. נתון מסד נתונים המחזיק מידע על סטודנטים. נרצה למצוא את הציון הממוצע לכל שכבת גיל. בשלב ה-map נמפה בין הגיל לציון ונסנן את כל שאר המידע. בשלב ה-Grouping נאחד את הזוגות עם אותו גיל ונשלח ל-worker כלשהו. בשלב ה-Reduce כל worker ימצא את הממוצע ציונים בקבוצת הסטודנטים שקיבל ויחזיר את המידע.

## Spark

ספארק הוא מסגרת תוכנה המשמש כדי לבצע חישובים על כמות גדולה של מידע במערכת המבוזרת למחשבים רבים. פותח לתת מענה למגבלות הקיימות בתבנית Map-Reduce אשר מכריח מעבר ליניארי של מידע. הוא יותר מהיר, מנהל זיכרון בצורה מיטבית, גמיש, יותר קל ונוח לפתח בו. בנוסף, מאפשר להפעיל אלגוריתמים שחוזרים על עצמם מספר פעמים, ניתן לעשות בו עיבודים של ML, ומספק עיבוד בזמן אמת.

ספארק חושף ממשק (API) למפתחים המתבסס על מבני נתונים הנקראים RDD ו-Data-frames. ישנם מספר שפות שבהן ניתן להשתמש בממשק זה, כמו Scala, Java, R, Python.

* RDD (resilient distributed dataset), מחזיק קבוצה של פריטים המפוזרים בין מספר מחשבים, באופן שמספק עמידות בנפילות או שגיאות. RDD של ספארק פועל כסט מידע פעיל אשר באופן מכוון יוצר ביזור מידע בתצורה מוגבלת.
* Data-frame - מבנה נתונים המשמש כדי לאחסן טבלאות. בנוי על גבי RDD, אמנם בעוד RDD משמש כדי לאחסן נתונים לא מסודרים ומקוטלגים (כמו קבצי טקסט), Data-frame משמש כדי לאחסן נתונים עם מבנה מאוד מסודר ומקוטלג לרשומות ועמודות. למעשה, פעולות חישוב על Data-frame יותר מהירות מאשר ב-RDD מאחר ומשתמש בהמון אופטימיזציות כדי לייעל את החישובים שמבצע. הוא מאפשר בנוסף להריץ פקודות בשפת SQL ממש על הנתונים שמאחסן, וגם מוגדרות עליו שיטות המזכירות את הפקודות ב-SQL.

## Kafka

מתווך מסרים בפלטפורמת קוד פתוח. מתווך מסרים זהו כלי המשמש להעברת הודעות כאשר השולח אינו מיעד את ההודעה ללקוח ספציפי אלא מסווג את ההודעה לסוג מסוים של לקוחות, והלקוח מציין אילו סוג הודעות הוא מעוניין לקבל. Kafka מספק תפוקה גבוהה וזמני השהיה קצרים בזמן אמת. שכבת האחסון היא תור הודעות גדול ממדים בתבנית עיצוב יצרן/צרכן (pub/sub) עם ארכיטקטורה מבוזרת הניתנת להגדלה.

ב-Kafka שומרים כל הודעה בקטגוריות הנקראות topics. תהליך שמפרסם הודעות ל-topics נקרא יצרן ותהליך שמקבל את ההודעות נקרא צרכן. Kafka מורכבת ממספר שרתים שכל אחד מהם נקרא broker. התקשורת בין הרכיבים מתבצעת באמצעות API ופרוטוקול TCP.

## ארכיטקטורות למבדה וקאפה

למבדה – תבנית לעיצוב מערכות ביג דאטה עם 2 ערוצים של נתונים: עיבוד מהיר שבו רוצים להציג למשתמש נתונים מהר, ועיבוד איטי שבו רוצים להגיע לניתוח מדויק יותר ורק לאחר מכן מציגים למשתמש. מחלקים את הנתונים לפי מה דחוף יותר ומה דחוף פחות.

קאפה – תבנית לעיצוב מערכות ביג דאטה שבה יש רק ערוץ מהיר וישר מוצגים למשתמש. מיועד ל-streams.

# למידת מכונה

תת-תחום במדעי המחשב ובבינה מלאכותית, העוסק בפיתוח אלגוריתמים שבאמצעותם ניתן ליצור תוכנות מחשב הלומדות מתוך דוגמאות כיצד להסיק תכונות לא ידועות על מידע חדש. למידת מכונה פועלת במגוון משימות חישוביות בהן התכנות הקלאסי אינו אפשרי. המקרה הנפוץ ביותר בו נרצה להשתמש בלמידת מכונה הוא בעיית זיהוי שמומחה אנושי מסוגל לפתור, אך לא מסוגל לכתוב את הכללים לזיהוי בצורה מפורשת או שהם משתנים עם הזמן ולא ניתנים לכתיבה מראש. העקרון המרכזי בלמידת מכונה הוא לתת למחשב קבוצה גדולה של מידע מתויג וכלים שבאמצעותם המחשב יוכל ללמוד את המידע ולפתח מודל שידע להכריע בהסתברות גבוהה על כל מידע חדש.

## סוגי בעיות

**Classification** - נתונים k מחלקות. אנו מעוניינים לסווג כל אובייקט לאיזה מחלקה מ-k המחלקות הוא שייך. פעמים רבות מספר המחלקות הוא 2 המייצגים "כן" או "לא", אך גם יכולים להיות מספר רב של מחלקות. דוגמאות:

* בהינתן נתונים על אדם נרצה לדעת האם הוא עובד או מובטל.
* בהינתן תמונה של פרי נרצה לדעת מהו סוג הפרי.

**Regression** - בהינתן אובייקט x נרצה לנבא עבורו איזשהו ערך מספרי רצוף y. דוגמאות:

* בהינתן נתונים על אדם כלשהו נרצה לנבא מהו משקלו או גובהו.
* בהינתן גרסה של פלאפון נרצה לנבא מהו מחיר הפלאפון.

**Clustering** - בהינתן n אובייקטים לקבץ אותם למספר קבוצות, ושיוך דוגמה חדשה לקבוצה המתאימה ביותר. כל האובייקטים הנמצאים באותה קבוצה דומים זה לזה יותר מאשר לאובייקטים השייכים לקבוצות אחרות. לדוגמה, פילוח של לקוחות לפי התנהגות צרכנית ותכונות דמוגרפיות.

**Ranking** - בהינתן n אובייקטים לדרג אותם מהגבוה לנמוך לפי העדפות משתמש. לדוגמה, לדרג אתרי האינטרנט שנמצאו בחיפוש לפי העדפות המחפש כך שהטובים ביותר יופיעו קודם.

## סוגי למידה

נהוג לחלק את אלגוריתמי למידת המכונה למספר סוגים:

* למידה מונחית (supervised learning) - כל דוגמה מגיעה עם תיוג. מטרת האלגוריתם היא ללמוד מהמידע המתויג כיצד לחזות את התיוג של דוגמאות חדשות שאותן לא פגש בתהליך הלמידה. לדוגמה סיווג מיילים ל"חשוב" ו"ספאם" כאשר נתונים מיילים שכבר מסווגים. החסרונות בלמידה כזו שקשה להחליט מהו המידע הטוב ביותר ללמוד עליו ומהו האלגוריתם הטוב ביותר ללמוד איתו.
* למידה בלתי מונחית (unsupervised learning). מטרת האלגוריתמים היא למצוא ייצוג פשוט וקל להבנה של אוסף הנתונים. שיטות נפוצות מסוג זה הן חלוקה לקבוצות (clustering), ושיטות להורדת מימד כגון ניתוח גורמים ראשיים (PCA). החיסרון בלמידה זו שבדרך כלל מביאה לתוצאות פחות טובות מלמידה מונחית.
* למידה מונחית חיזוקים (reinforcement learning) - אלגוריתם הלמידה מקבל משוב חלקי על ביצועיו (רק לאחר סיום ביצוע המטלה) ועליו להסיק אילו מהחלטותיו הביאו להצלחה/כישלון. בסוף תהליך הלמידה האלגוריתם יודע בכל מצב מהי הפעולה הטובה ביותר. למידה זו משמשת כיום בעיקר בפיתוח תוכנה שתדע לשחק משחקים.

## Linear Regression

זהו אלגוריתם שפותר בעיית רגרסיה, כלומר מחזיר ערך מטווח רציף ולא בדיד. רגרסיה ליניארית היא שיטה מתמטית המאפשרת לנו לבנות פונקציה ליניארית, כך שבהינתן אובייקט חדש x, שאנו יודעים את **כל** k התכונות שלו, נוכל לנבא מהו הערך y עבור x.

כל מייצג את המשקל (weight) שאנו נותנים לכל יחידה מהתכונה ה-i, ו-b הוא ערך מינימלי כלשהו (bias), שמטרתו להוסיף כוח חישוב שאינו תלוי בתכונות. הסיבה שאנו רוצים דווקא פונקציה שהיא ליניארית היא מפני שאנו מחפשים חוק פשוט, שלפי משפט Sauer הוא חוק טוב.

הרעיון המרכזי הוא שהפונקציה הליניארית שאותה אנו מחפשים היא הפונקציה שסך הסטיות של כל m האובייקטים בדאטה ממנה הוא מינימלי. ככל שיש לנו יותר מידע, כלומר יש בידינו המון אובייקטים נתונים, כך יכולות הניבוי של המודל יהיו טובים יותר. אמנם יש איזשהו גבול של מספר אובייקטים שכבר לא ישפר את המודל.

## עצי החלטה

עץ החלטה הוא עץ בינארי מלא המורכב מצמתי החלטה שבכל אחד מהם נבדק תנאי מסוים על תכונה ועלים המכילים את הערך החזוי עבור האובייקט המתאים למסלול שמוביל אליהם בעץ. ככל שתכונה נמצאת גבוה יותר בעץ כך היא יותר חשובה לניבוי.