**REST**

REST (Representational State Transfer) הוא ארכיטקטורת תקשורת ועקרונות עיצוב לפיתוח שירותי ווב,

Rest הוצג לראשונה בשנת 2000 והוא מתאר סט של הגבלות ארכיטקטוניות שכאשר מיושמות, מאפשרות תקשורת יעילה, גמישה, וחסכונית במשאבים בין לקוחות ושרתים באינטרנט. Restאינו Design Pattern אלא ארכיטקטורה שמתארת איך יישומים ברשת צריכים לתקשר. אפשר ליישם ארכיטקטורת REST גם ביישומים שמשתמשים ב-MVC, אבל ה-View נמצא בדרך כלל בצד לקוח ולא בשרת.

באפליקציות ווב ע"פ עיקרון הREST, המשאבים מיוצגים על ידי טקסט, בדרך כלל בפורמטים כגון XML או JSON, State Transfer מתייחס לעובדה שפעולות על המשאבים (כמו יצירה, קריאה, עדכון, מחיקה) משנות את ה"מצב" של המשאבים אלו בשרת.

**מהו RESTful API?**

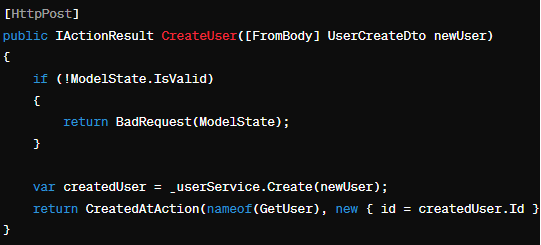
RESTful API הוא ממשק תכנותי לאפליקציות ווב שעוקב אחר העקרונות של REST. הוא מאפשר לאפליקציות לשלוח ולקבל נתונים באמצעות בקשות HTTP, תוך שימוש בסטנדרטים כמו URIs (Uniform Resource Identifiers), מתודות HTTP (כגון GET, POST, PUT, DELETE), וקודי תגובה HTTP.

RESTful APIs נועדו להיות פשוטים להבנה ולשימוש, והם מבוססים על משאבים (Resources) שכל אחד מהם מזוהה באמצעות URI. המידע יכול להישלח ולהתקבל בפורמטים שונים, כמו XML או JSON, אך JSON הוא הנפוץ ביותר בשל פשטותו וקלות השימוש עם JavaScript בצד לקוח.

**עקרונות ה-REST**

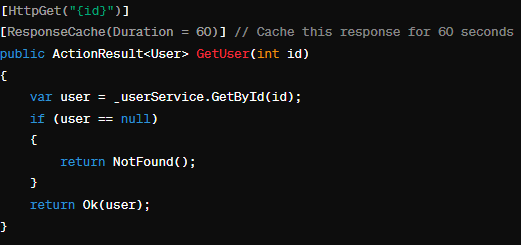
Client-Server**:** הפרדה ברורה בין לקוח (המשתמש הסופי או הדפדפן) לבין שרת, מה שמאפשר פיתוח וקידום עצמאי של שני הצדדים.

**Stateless**: עקרון ה-Stateless ב-REST מתייחס לכך שכל בקשה מהלקוח לשרת צריכה להיות מלאה ועצמאית - כלומר, השרת לא "זוכר" מצב מבקשות קודמות. כל בקשה צריכה לכלול את כל המידע הנדרש לשרת כדי לעבד אותה, כולל פרטי אימות (אם נדרש), פרמטרים, וכל מידע נוסף שנחוץ ללוגיקת העסקים. וזה תורם ליציבות ולקלות התחזוקה של המערכת, מכיוון שהשרת לא צריך לשמור מצב או להניח הנחות על בקשות קודמות.



**Cacheable:** כאשר הלקוח (למשל, דפדפן אינטרנט) מבקש לראשונה את המידע הזה, השרת מחשב או שולף את העלות ממסד הנתונים, מחזיר אותה ללקוח, ושומר את התגובה במטמון. בקשות עתידיות לאותו המידע יכולות להיענות מהמטמון במקום לגרום לשרת לבצע חישוב מחדש או לפנות שוב למסד הנתונים.

בדוגמא הזאת רואים שהתגובה שהלקוח קיבל יכולה להישמר במטמון (בדפדפן של הלקוח או במטמון צד שרת) ולשמש לענות על בקשות עתידיות של אותו משאב במשך 60 שניות ללא צורך לבצע את הבקשה מחדש לשרת.



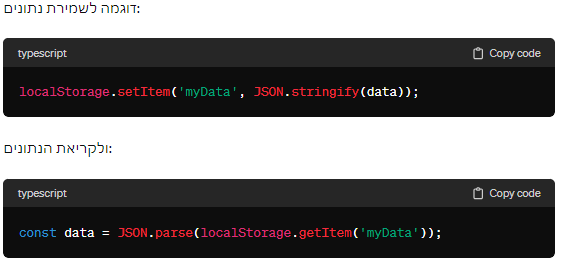
מתי נזדקק למטמון?

1. נתונים שאינם משתנים לעיתים קרובות: אם יש באפליקציה נתונים שאינם משתנים לעיתים קרובות, כמו רשימת קטגוריות או תגיות, אפשר לאחסן את התגובה במטמון לזמן מסוים.
2. שיפור הביצועים: כאשר רוצים לצמצם את זמן התגובה ולהפחית את העומס על השרתים, במיוחד במערכות גדולות עם כמות גבוהה של בקשות.
3. מזעור עומסים בשעות פיק: במידה ואתר מקבל כמות גדולה של תנועה בשעות מסוימות, שמירת תוכן במטמון יכולה לסייע בשמירה על ביצועי האתר.

נקודות לזכור:

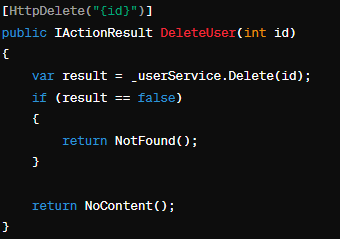
1. עדכון המטמון: אם מה שנשמר במטמון משתנה לאחר פרק הזמן שנכתב באטריביוט או בכל פרק זמן אחר, יש לדאוג לעדכון המטמון בהתאם.
2. שיקולים בביצועים: תכנן בקפידה את מדיניות המטמון שלך כדי למקסם את היתרונות של מטמון בלי להחזיק מידע לא עדכני.
3. תמיכה במטמון: ודא שהתשתית שלך (למשל, דפדפנים, פרוקסי, ושרתים) תומכת ומוגדרת נכון לעבודה עם מטמון לפי הצורך.

ניתן להשתמש בזיכרון מטמון גם בצד שרת כפי הדוגמא למעלה, או בצד לקוח בLocalStorage



ניהול נכון וחכם של מטמון יכול לשפר משמעותית את הביצועים והיעילות של אפליקציות, אך דורש תכנון, ניטור ותחזוקה מתמשכים כדי למנוע פוטנציאל לבעיות.

**Uniform Interface:** ממשק אחיד, REST משתמש בפעולות HTTP סטנדרטיות (כמו GET, POST, PUT, DELETE) לפעולות על משאבים. המתכנת יכול ליישם את הממשק האחיד על ידי כך שישתמש בפעולות ה-HTTP הללו באופן עקבי ב-API שלו, ויבטיח שה-API מחזיר משאבים בפורמטים סטנדרטיים כמו JSON או XML.



בפיתוח אפליקציות ווב, במיוחד בעבודה עם ASP.NET Core, ישנם מספר סוגי בקשות HTTP נפוצים שמשמשים לתקשורת בין הלקוח (למשל, דפדפן או אפליקציה) לשרת. כל סוג בקשה מייצג פעולה שונה שהלקוח רוצה לבצע על המשאב בשרת. הנה הסוגים העיקריים:

GET

תיאור: משמשת לאחזור מידע מהשרת. בקשת GET לא אמורה לשנות מידע בשרת, אלא רק לקבל נתונים.

שימוש נפוץ: לאחזר דף אינטרנט, תמונה, או נתוני JSON מ-API.

POST

תיאור: משמשת ליצירת משאב חדש בשרת. בקשת POST נשלחת עם נתונים בגוף הבקשה, אשר לרוב ישמשו ליצירת משאב חדש.

שימוש נפוץ: ליצור רשומה חדשה במסד נתונים (לדוגמה, משתמש חדש).

PUT

תיאור: משמשת לעדכון משאב קיים בשרת או ליצירת משאב חדש אם הוא לא קיים.

שימוש נפוץ: לעדכן פרטים של משתמש קיים במסד נתונים.

DELETE

תיאור: משמשת למחיקת משאב מהשרת.

שימוש נפוץ: למחוק רשומה מסוימת ממסד נתונים, כמו מחיקת משתמש.

PATCH

תיאור: משמשת לעדכון חלקי של משאב קיים. בניגוד ל-PUT, שעלול לעדכן או להחליף את כל המשאב, PATCH מעדכן רק את החלקים שנשלחו בבקשה.

שימוש נפוץ: לעדכן חלק מהפרטים של משתמש במסד נתונים, ללא הצורך לשלוח את כל המידע מחדש.

לכל אחד מסוגי הבקשות הללו יש תפקיד ומשמעות בתקשורת בין הלקוח לשרת, והם מאפשרים למפתחים לבצע פעולות שונות על משאבים באינטרנט באופן מובנה ותקני.

סוגי הבקשות HTTP כמו GET, POST, PUT, DELETE, ו-PATCH הם חלק מהפרוטוקול HTTP עצמו ולכן הם מוגדרים ותקניים. השימוש בהם אינו נתון לשינוי או תרגום חופשי מצד המפתח.

למה צריך Attributes מסוגים אלה, הרי כמה קשה זה למחוק לקוח או לעדכן ערך מסויים?

נלך על דוגמאת הHttpDelete ונבין למה צריך אותו,

1. הבהרת ותיעוד הקוד - השימוש באטריביוט [HttpDelete] מבהיר מיידית לכל מי שקורא את הקוד שהמתודה מיועדת לטפל בבקשות מחיקה. זה מסייע לתחזוקת הקוד ולשיתוף פעולה בצוותים גדולים, כאשר הכוונות והפעולות של כל מתודה מובנות מאליהן.
2. ניתוב הבקשות - ב-Web API, ניתוב הבקשות למתודות המתאימות בקונטרולר מתבצע בהתבסס על סוג הבקשה (למשל, GET, POST, DELETE) והנתיב (URL). האטריביוט [HttpDelete] מאפשר לפריימוורק לזהות בקלות איזו מתודה צריכה להיקרא כאשר מתקבלת בקשת DELETE.
3. הגנה ובטיחות - הפרדה בין סוגי הבקשות מסייעת למנוע פעולות בלתי מורשות או בלתי רצויות. לדוגמה, תוכנית יישום שמשתמשת בבקשות GET בלבד לא תוכל בטעות למחוק נתונים מהשרת. כך מתודות שצריכות לטפל בעדכון או מחיקה של נתונים מקושרות לבקשות מסוגים מתאימים כדי להגביר את הבטיחות ולמנוע שגיאות.

אמנם לא חייבים להשתמש באטריביוט [HttpDelete] (או בכל אטריביוט HTTP ספציפי אחר) כדי שמתודה תוכל למחוק רשומה מהשרת, אבל השימוש בו מומלץ מאוד מסיבות שקשורות לתקינות, ברורות, ואבטחה. ובמקרה כזה של ניתוב ללא אטריביוט נצטרך לעשות ניתוב ידנית, וגם נסתכן בשגיאות, ויתכן שנפעיל את הלוגיקה של המחיקה בתגובה לסוג בקשה שאינו מתאים.

הערה\הארה: ניתן להקים מתודה בקונטרולר שתבצע מספר פעולות על משאב בתוך טרנזקציה אחת, כולל קריאה (GET), עדכון (UPDATE), ושמירה (PUT או POST בהתאם לסמנטיקה של הפעולה). עם זאת, יש לשים לב למספר נקודות חשובות בעת עיצוב ופיתוח של מתודה – **אך במקרה כזה עוברים על עקרונות הRest.** לכן, פעולה שמשלבת GET ואז PUT/UPDATE באותה מתודה עשויה להיות בעייתית מבחינת הגדרת ה-API לפי עקרונות REST. זה לא אומר שלא ניתן לעשות זאת טכנית, אלא שזו פרקטיקה שכדאי לשקול אותה בזהירות.

**Layered System**: המערכת יכולה להיות מאורגנת בשכבות, כאשר כל שכבה מספקת שירותים לשכבה שמעליה. זה יכול להיות מיושם על ידי הפרדת הלוגיקה לשכבות שונות (למשל, שכבת נתונים, שכבת לוגיקת עסקים, שכבת API). זה מאפשר גמישות בפיתוח ובתחזוקה, וכן מאפשר להכניס שכבות אבטחה, שכבות מטמון, ועוד, ללא השפעה על הלקוחות.

לדוגמא:

1. שכבת הנתונים: מחלקות מודל שמגדירות את הנתונים (למשל, User).
2. שכבת לוגיקת עסקים: שירותים כמו IUserService שמנהלים לוגיקת עסקים.
3. שכבת ה-API: קונטרולרים שמגדירים את ה-API ומשתמשים בשירותים משכבת לוגיקת העסקים.

**Code on Demand** (אופציונלי): האפשרות לשלוח קוד מהשרת ללקוח (לדוגמה, JavaScript) שירוץ בצד הלקוח מאפשרת גמישות גדולה יותר ביצירת ממשקים דינמיים ותגובתיים. זה יכול להיות מיושם על ידי כך שהשרת מחזיר קוד JavaScript כחלק מתגובה לבקשה מהלקוח. הקוד הזה אז יכול להרוץ בדפדפן הלקוח ולשנות את הממשק או לבצע פעולות ללא צורך בטעינה מחדש של הדף.

עם זאת, יש לציין שבאנגולר ובפריימוורקים דומים, פעולה זו אינה נפוצה מאוד מכיוון שהקוד בדרך כלל נטען מראש כחלק מהאפליקציה.