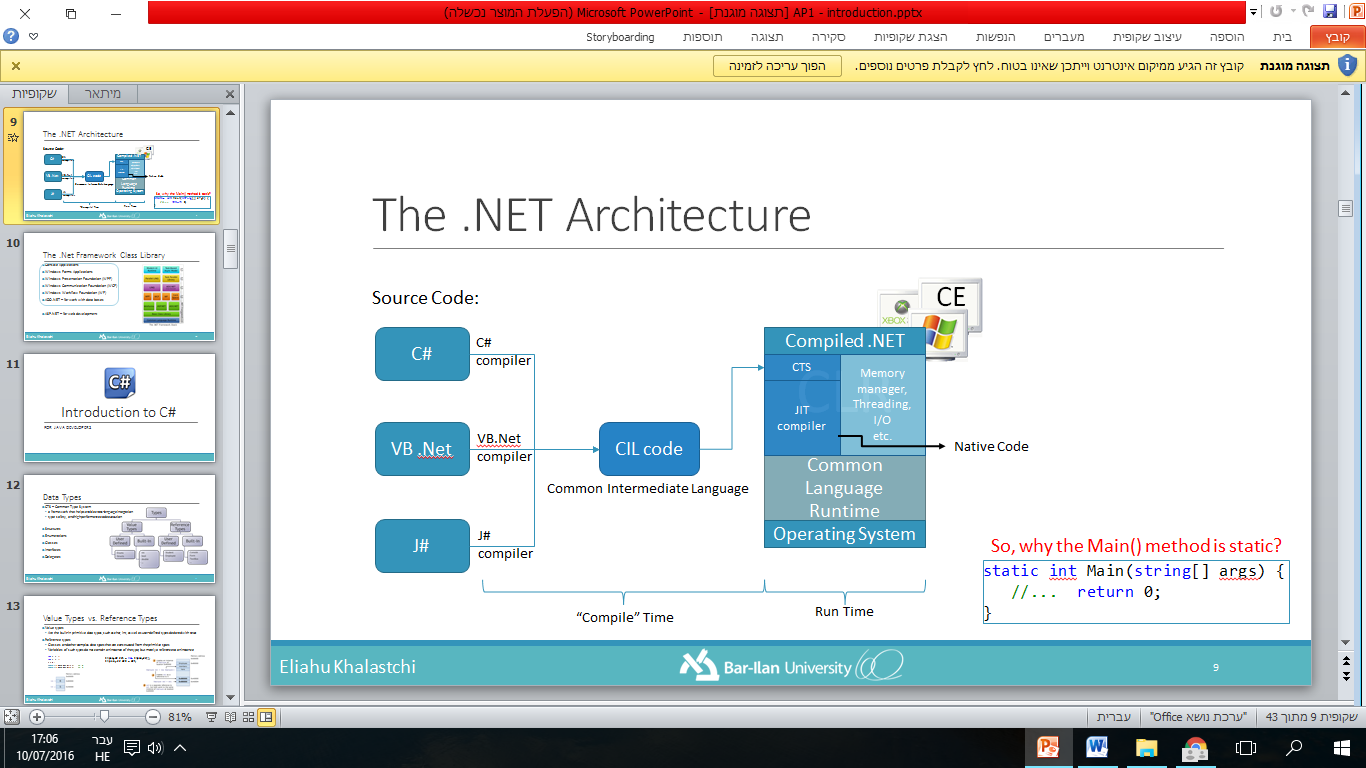
**תכנות מתקדם 2 –סיכום הרצאות**

שיעור 1

ייחודי לC#:

Data Types

Parameter Passing

Properties

String Interns

Operator Overloading

Delegates

Events

Data type

* CTS-common type system מסגרת שמאפשרת לאינטגרציה של המעבר שפות, type safety, and high performance code execution/
* Value types
  + Are the built-in primitive data types, such as char, int, as well as user-defined types declared with struct
* Reference types
  + Classes and other complex data types that are constructed from the primitive types
  + Variables of such types do not contain an instance of the type, but merely a reference to an instance

Primitive types in C# are Objects! int is an alias for System.Int32

double is an alias for System. Double

* Arrays in C# are actually references to **objects** that contain an array (like in Java)
  + Elements of the same type placed continuously in the memory, accessed by an integer index
  + This array object has methods and properties

Parameter Passing

בC# פרמטר הדיפולטיבי עובר by value .

Like in Java, when a reference type is passed by value

The object’s address is passed by value

Therefore, the local variable still points to the passed object

And can manipulate the object’s data

However, calling new will only change the address of the local variable

And will not change the passed object

**Out:**  המשתנה לא צריך להיות מאותחל, אם הוא מאותחל אז הערך מבוטל אך חייב להיות אתחול בתוך המתודה.

**Properties**

String Interns

C# == יעבוד על 2 מחרוזות זהות גם אם שניהם נוצרו ע"י new, כאשר זה אותו REF אז ייקח O(1) אחרת O(n). בג'אווה יש מנגנון של string pool בדומה יש בC# intern pool , בכל יצירת מחרוזת הבריכה נבדקת של אחסון המחרוזת וref נשמר. המתודה insert() היא סטטית של מחלקת הString שמחזיר את ref שנשמר.

Operator Overloading

כל Operator Overloading יהיה מתודה סטטית בניגוד לc++ . נרשום operator -(T a,T b)

Delegates & Events

ניתן להעביר delegate כפרמטרים

new Thread(

delegate() {

while(!stop){

x++;

}

**}**

).Start();

Interface

אפשרי להיות : חתימות של מתודות, properties,events

מצגת 2

Design Principles: אבסטרקציה, כימוס, ירושה ופולימורפיזם.

General Principles:

DRY\* don’t repeat yourself)) - לא להעתיק להדביק בלוקים של קוד, להימנע מכפילויות בסכמות המידע, הדיאגרמות והתיעוד. “A single source of truth”

\*YAGNI(you ain’t gonna need it)-לא לכתוב קוד של ספקולציה, פתור את הבעיה הקיימת.

\*להימנע מcode smell

Code smell

1.שיכפול קוד: קוד דומה או זהה מופיע ביותר ממקום אחד.

2.סיבוכיות מיותרת: שימוש בתבנית עיצוב מסובכת במקום שתבנית פשוטה הייתה מספקת

3.שכבות/מודלים טריוויאליים שנצטרך רק את השכבה אחת מתחת.

4.god class- שכבה שמנסה לעשות הכל , יש לה אחריות רבות שלעיתים אינם קשורות אחת לשניה, יש לה יותר מידי data members,נפוץ אצל מתכנתים הלמדו רק את הסינטקס.

5. Feature Envy- עושה מעט חוץ משימוש בכל המתודות של מחלקה אחרת.

6.אינטמיות לא מתאימה: מחלקה שתלויה בפרטי מימוש של מחלקה אחרת.

7.freeloader – מחלקה שעושה קצת מידי.

8. Conditional complexity-יותר מידי ענפים ולולאות, דבר שאולי מראה על פונקציה שצריך לפרק לפונקציות אחרות או על פוטנציאל לפישוט.

9. Downcasting-cast ששובר את מודל האבסטרקציה למשל java containers before java generics

10.מחלקה עם יותר מידי משתנים:לשקול לחלק אחריות עם מחלקה אחרת

11.דימיון בתת מחלקות- למשל 2 תת מחלקות שמטפלות ב2 קלטים שונים באותה דרך.

12.הורשה מרובה נקרא גם The diamond of death

13.יותר מידי פרמטרים: קשה לקריאה, לקרוא ולבדוק. מראה על אי חשיבה על מטרת הפונקציה, כדאי לעשות חלוקה מחדש של אחריות (refactor)

14.מתודה גדולה: בדר"כ נכתבת ע"י procedural programmer

15.מזהה ארוך/קצר מידי: מזהה צריך לשקף את הפונקציה אלא אם תפקידה ברור.

16. Excessive return of data- הפונקציה מחזירה יותר מידע ממה שהקורא צריך.

17.יותר מידי מתודות non-public : קשה לאבחן

18.data class: יש להימנע ממחלקות שבצורה פסיבית מאחסנות מידע, מחלקה צריכה להכיל מידע **ו**מתודות.

19. Middle man: מחלקה שהמאצילה את כל הסמכויות, wrappers צריכים להפעיל משהו.

20.אותו שם ,משמעות שנה.

21.שמות לא קונסיסטנטיים , לדוגמא open() אז צריך להיות close()

22.הערות חסרות משמעות, הקוד צריך להסביר את עצמו.

23.חוסר הערות כאשר נחוץ.

SOLID -

S – Single-responsibility principle - כל מחלקה בעלת תחום אחריות אחד.

O – Open-closed principle - פתוחה להרחבות סגורה לשינויים.

L – Liskov substitution principle - לא צריך לעשות down casting. כשעושים קוד גנרי עבור מחלקת אב אשתמש במתודות שלו בלבד.

I – Interface segregation principle - אינטרפייס ארוך זה בעייתי. להפריד בין תחומי עניין שונים.

D – Dependency Inversion Principle - קשר בין שתי מחלקות צריך להיות דרך ממשק, אינטרפייס כלשהו כלומר: קשר עקיף.

GRASP - General responsibility assignment software patterns (or principles)

SOLID וGRASP אינם מהווים קונפליקט ביניהם, מכוון זה שם את הדגש על האחריות.

ישנם 9 רעיונות בGRASP:

• Creator - המחלקה שתייצר את האובייקט היא בעלת המידע עליו.

• Controller - להוסיף מחלקה אבסטרקטית בין אינטרפייס והמממשים שלו כדי לממש את הפונק' המשותפות.

• Pure Fabrication - מותר לייצר מחלקה עבור פונ' אחת בלבד. לא לשים אותה במקומות לא קשורים.

• Information Expert - עבור בעיה מסוימת המחלקה שפותרת אותה זה אותה מחלקה שיש לה את המידע עליה.

• High Cohesion, Low Coupling - תלויות מעטות בין מחלקה למחלקה. וכל התלויות יהיו באותה מחלקה.

• Indirection - להוסיף מתווך בין מחלקות שדורשות תלויות.

• Polymorphism - קוד כמה שיותר גנרי

Protected Variations - לזהות חלקים בקוד שעלולים להשתנות ולטפל בהם בהתאם.

עקרונות אלו שלOO מאפשר לנו לכתוב קוד קריא, שניתן לחזק וגמיש.

**Design Patterns**

Static vs. Dynamic Design

Static נעשה בעזרת ירושה אך ניתן לרשת רק אחד לכן נסתכן בקוד כפול

**עקרון חשוב :**כל פתרון שהוא סטטי, משתמש בירושה, שמשתמש ברמות נמוכות של קוד נידון לחזור על עצמו. כלומר, יגרום שכפול קוד. אם נעלה את הפתרון למעלה נקבל פתרון יותר טוב.

Dynamic משתמש ברכבה לדוגמא bridge pattern.

**From pseudo-code to OOP**

רבות מהבעיות ניתן לתרגם לבעיות פשוטות יותר.

כל אחד מהמשחקים במצגת יכולים להיות מוגדרים כגרף , כאשר כל קודקוד זה פריט במשחק ,משקל. הקשת היא עלות הזזה,ובעצם אנחנו רוצים את המסלול הזול ביותר. בשביל זה נשתמש בBFS (best) . אלגוריתם שדומה לBFS המוכר, רק שהוא בוחר את הקשתות ע"פ משקלם, לפי עדיפות.נשתמש בתור עדיפויות.

האלגוריתם:

**Best First Search:**

OPEN = [initial state] // a **priority queue** of states to be evaluated

CLOSED = [] // a **set** of states already evaluated

while OPEN is not empty

do

1. **n** 🡨 dequeue(OPEN) // Remove the best node from OPEN

2. add(n,CLOSED) // so we won’t check n again

3. If **n** is the goal state,

backtrace path to **n** (through recorded parents) and return path.

4. Create **n**'s successors.

5. For each successor **s** do:

a. If **s** is not in CLOSED and **s** is not in OPEN:

update that we came to **s** from **n**

add(**s**,OPEN)

b. Otherwise, if this new path is better than previous one

i. If it is not in OPEN add it to OPEN.

ii. Otherwise, adjust its priority in OPEN done

כעת איך נעשה את זה בOOP,?אנחנו רוצים להפריד(decoupling) את האלגוריתם מהבעיה.

אני ארצה להגדיר אלגוריתם חיפוש ובעיית חיפוש גנרים , כל מה שהאלגוריתם יודע זה שהוא מקבל איזשהו בעית חיפוש. ההבדל היחיד ביניהם הוא הפונקציונליות (כלומר להגדיר את הinterface. ) כך אח"כ כל ישות יכול להיות מימוש שונה בלתי תלוי מהשני.

**מצגת 3-Architectural Patterns**

תבנית אשר כוללת את הארכיטקטורה של התוכנה.

לא נרצה שהכול יבוצע בשכבה אחת מכוון שיהיה שינוי נצטרך לשנות הכל. בחלוקה לשכבות הקוד יהיה מודולרי ויהיה ניתן לעבוד כמה צוותים עליו וכמו כן לאתר ולבודד באגים.

רוצים להפריד בין model וview.

View-. להציג מידע ונתונים, לשאול את המשתמש מה הוא רוצה ולהבין אותו.

Model-. הbusiness logic , מכיל את הdata.

השכבה האמצעית- תפקידה לקשר בינהם , יודעת מי עושה מה.

"מתי,מי ואיך": המודל יודע מתי ואז פונה למי שיודע מה שקורה, וזה יעבור לview שידע איך להציג.

MVC



Controller

View

Model

view: מכיל data member של controller

Controller:מכיל dm של view +model,"מאכילה" מידע את 2 השכבות האחרות. (לפעמים גם פיזית גם משנה את המידע), אחראית גם כל לוגיקת הפעלה שלנו. בהינתן נוטיפיקציה,נשלוף אוביקט מסוג commend בO(1) (hashmap) ואז ניתן להשים אותו איפשהו (כי הוא אובייקט) ,כמו למשל בתור עדיפויות

Model-מכיל dm של controller

**Observer Design Pattern**

**בג'אווה**

* An ***Observable*** can **notify** many ***Observer***s
* An ***Observer*** can **subscribe** to many ***Observable***s

C#

* The ***Observable*** defines an event variable of some known delegate type
* The ***Observer*** registers its own delegates to the observable
* The observable activates all the registered delegates whenever it is needed

MVP- Model, View, Presenter

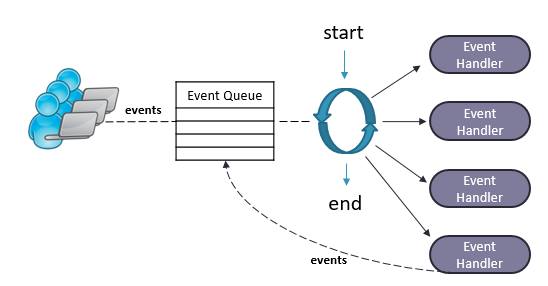
נגדיר delegate void func ();

View-event func ViewChanged

Presenter-יש dm של view +model

שנרשום את הפונקציות הרצויות של המודל וview לevents של הdata member הרצוי

Model- מכיל event func ModelChanged

**מצגת 4-WPF**

Event Driven -

קיימים - מייצרי event-ים , כל event היא ישות.

נסדר בתור עדיפויות(קדימיות)

יש thread ראשי בלולאה ששולף event מהתור ומשלחת לevent handler. זה מונחת אירועים. ראינו את זה ברובוט, ,GUIמערכת הפעלה, שרת, try-catch

Event Driven Programming

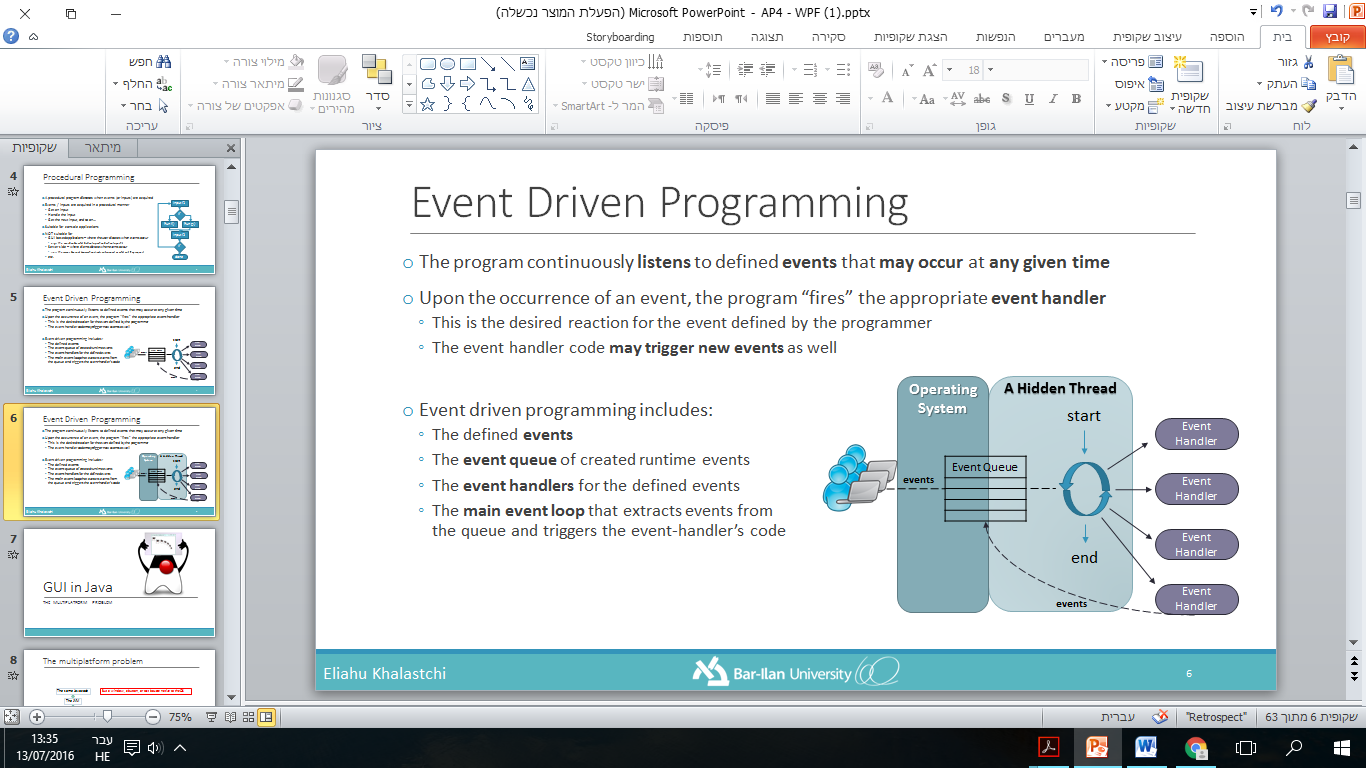
1.Procedural Programming:

Procedural Programming **מכתיב** מתי אירועים (או קלטים) הם נחוצים.

* Events / inputs are acquired in a procedural manner
  + Get an input
  + Handle the input
  + Get the next input, and so on…
* Suitable for console applications
* NOT suitable for
  + GUI based applications – where the user dictates when events occur
    - e.g., the user decides what button to push and when to push it
  + Server side – where clients dictate when events occur
    - i.e., the server does not know when clients will connect or what will they request
  + etc.

2.Event Driven Programming:התוכנית מקשיבה לאירועים מוגדרים ש**עלולים** לקרות ב**כל זמן נתון**.

שevent קורה התוכנית "משגרת" את הevent handler הנכון (התגובה הרצויה שהוגדרה ע"י המתכנת) , הקוד של הevent handler יכול לגרום להפעלת event חדש אחר.

Event driven programming includes:

* The defined **events**
* The **event queue** of created runtime events
* The **event handlers** for the defined events
* The **main event loop** that extracts events from  
  the queue and triggers the event-handler’s code

**GUI in Java**

**רקע**

* AWT - Abstract Windowing Toolkit (1995)

הGUI הראשון של ג'אווה , משתמש ברכיבי מ"ע , Lowest common denominator problem (רכיבי משתמש יש בכל פלטפורמה) . יעיל אך מכוער.

* Swing /JFC (1998):

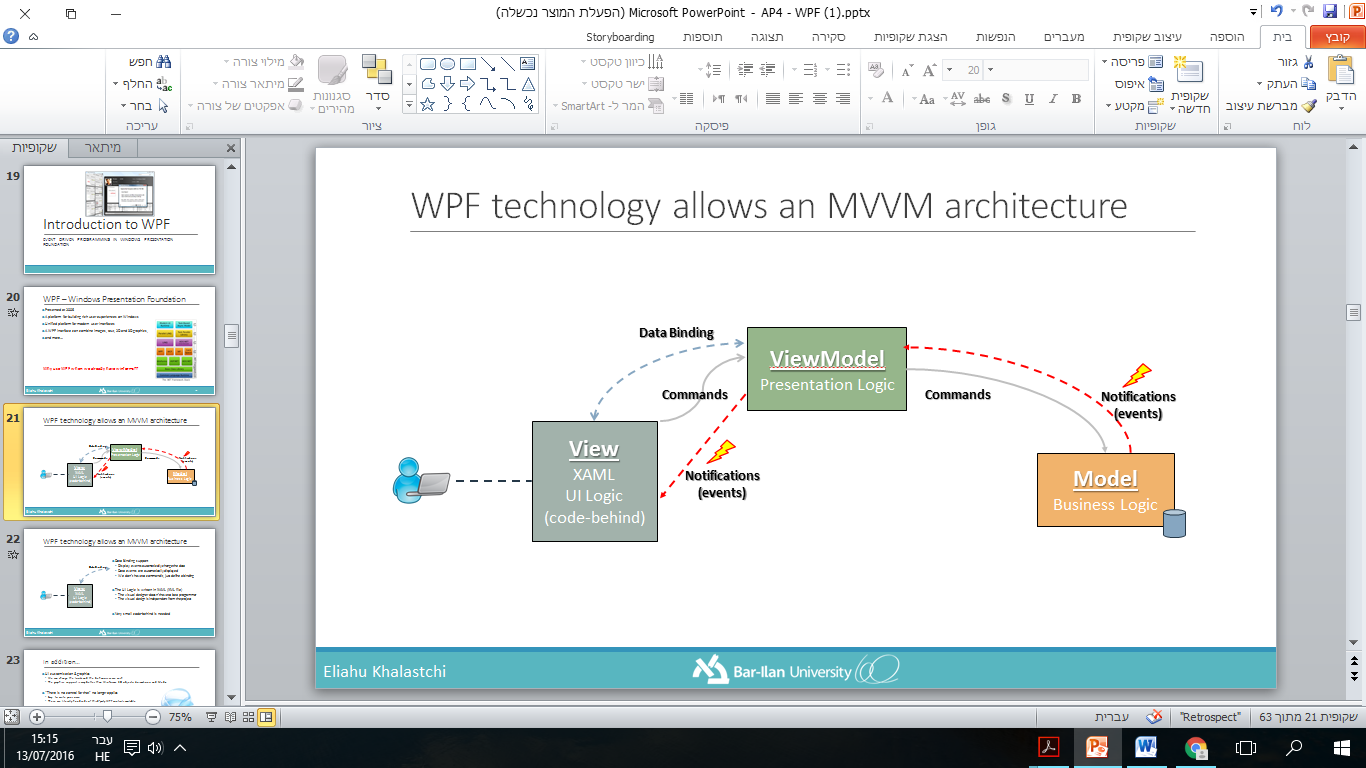
הGUI הרשמי של ג'אווה , כל רכיב נכתב בג'אווה המנסה לחקות את מראה מ"ע , יעבוד על כל פלטפורמה אך המראה לא יהיה אותו דבר.

* SWT – Standard Widget Toolkit (IBM, 2001)

מנסה ליהנות משני העולמות, משתמש ברכיבי המערכת במידה ואפשר,אחרת משתמש במימוש של ג'אווה , יש ראות ותחושה של מ"ע ,יש מימוש לכל פלטפורמה. צריך swt.jar הjar אפשר לשנות וישתנה בכל פלטפורמה, בכך נוכל לשנות את הjar מבלי לשנות את הקוד ועדין לקבל תחושת של מ"ע.

**WPF – Windows Presentation Foundation**

הוצג לראשונה ב2006, פלטפורמה לבניית user experiences בwindows . WPF מאפשר ארכטורה של MVVM.

**MVVM**

WPF \*תומך בdata binding

Data Binding

* + Display events automatically change the data
  + Data events are automatically displayed
  + We don’t have to command it, just define a binding

\*לוגיקת הUI נכתבת בXAML : כלומר העורך הוויזואלי לא צריך להיות מתכנת ובלתי תלוי בפרויקט.

\* קצת code-behind נחוץ.

\*UI customization & graphics: אפשר לשנות את המראה והתנהגות כרצוננו. התמיכה הגרפית יותר טובה מWinForm ( (3D objects, Animations, and Media.

* “There is no control for that” no longer applies
  + Easy to write your own
  + There are literally hundreds of third-party WPF controls available
* Ability to run in a browser
* Microsoft firmly switched its focus on WPF
* Strong ties to Silverlight technology
  + If you know WPF, it is easy to use Silverlight

Partial Classes

-אפשרי לחלק מחלקה, struct,interface,מתודה לשתיים או יותר source files

-בפועל כמה מתכנתים יכולים לעבוד על אותה מחלקה.

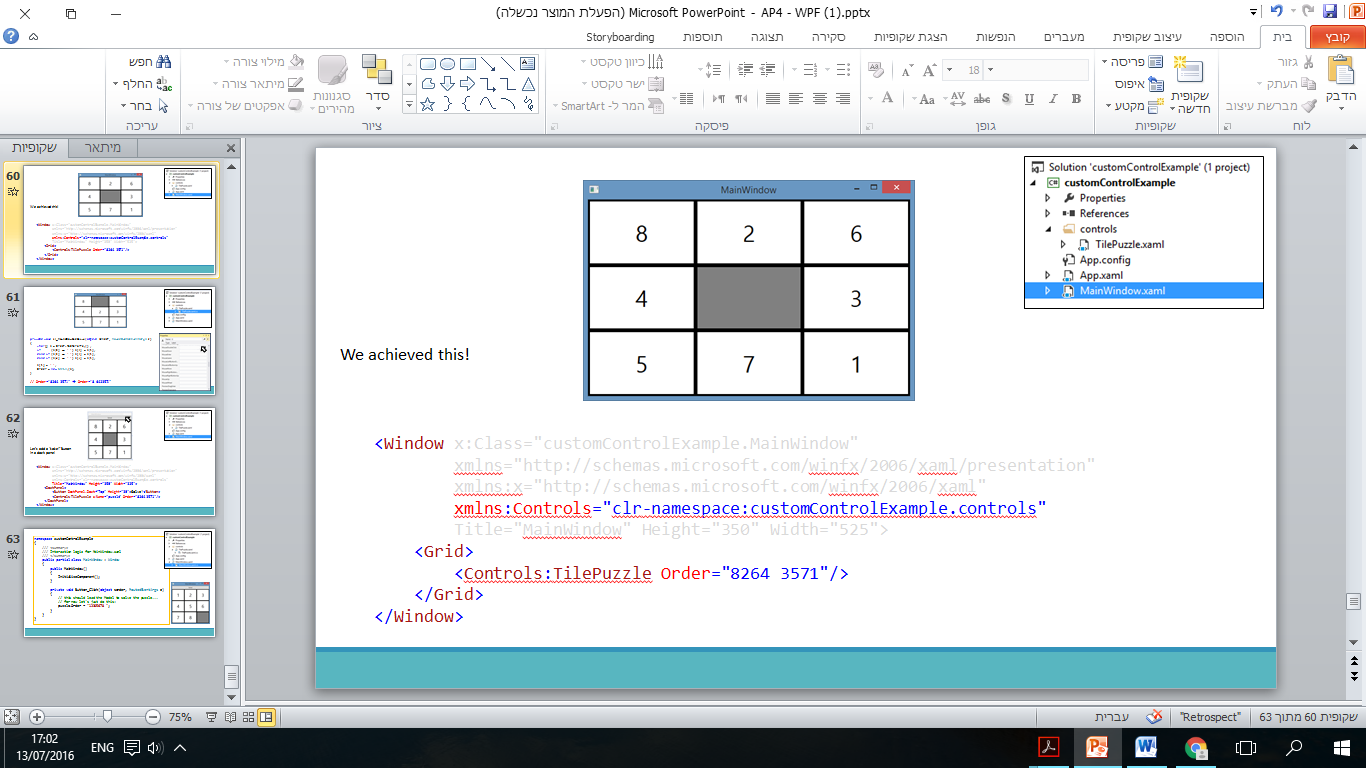
-יוצר בצורה אוטומטית קוד+source של אותו מחלקה.

- MainWindow שלנו בחלקו נערך ע"י visual studio.

XAML- Extensible Application Markup Language

* XAML is a declarative markup language
* XAML simplifies creating a UI for a .NET Framework application
* Separates the UI definition from the run-time logic by using code-behind files
* XAML enables a workflow where
  + separate parties can work on the UI and the logic of an application
  + using potentially different tools
* XAML are XML files with .xaml extenssion
* XAML can create objects, set Properties, and connect to events
* XAML cannot call methods,
  + for this we have the code-behind – to handle events and change items dynamically
* XAML object elements declares an instance of a type
* Use the *Attribute Syntax* to set the properties of an object
* When *Attribute Syntax* is not possible – use *Property Element Syntax*

Custom WPF Controls

* Controls: elements capable of receiving focus and handling input
* Many controls are available “out of the box”
*  Custom controls can be created
  + User controls that **wrap** one or more controls and expose higher level properties
  + Custom controls that **derive** from an existing control and extend its functionality

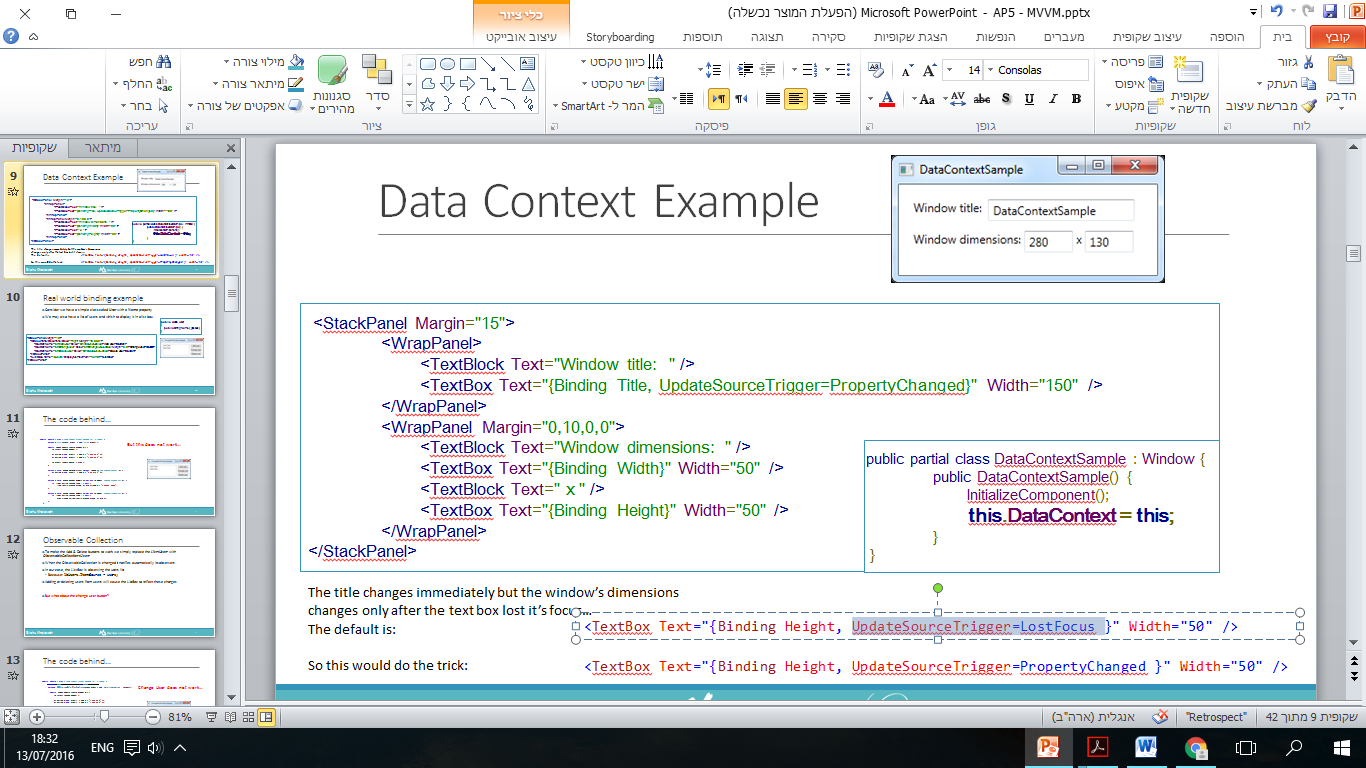
**מצגת 5-MVVM**

ViewModel

\*הוא המודל של הview.בשביל הview זה אבסרקסיה של המודל, זה מעביר פקודות מview למודל.

\*ממיר model information ל view information., משהו שview יכול להבין.

Data Biding

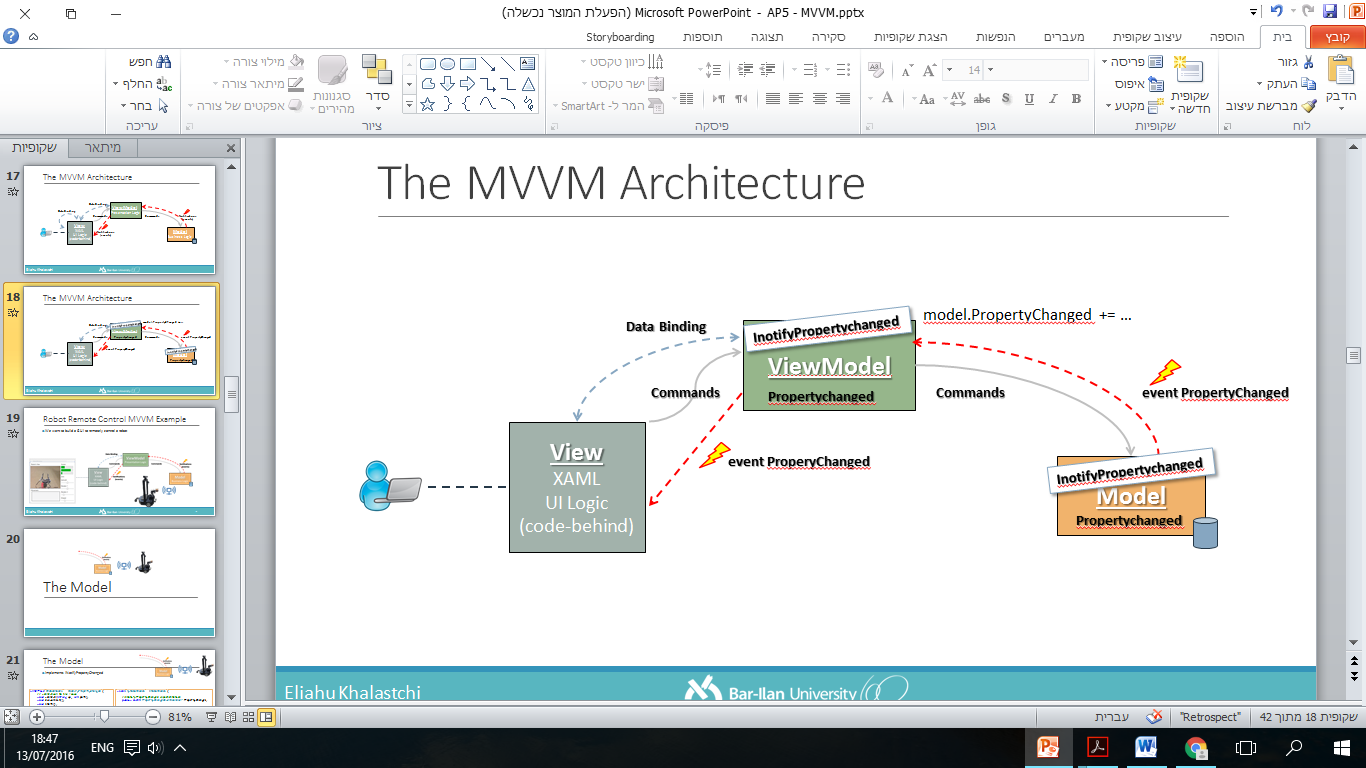
* Data binding is general technique that binds two data/information sources together
* and maintains synchronization of data
* Data binding in WPF is the **preferred way** to bring data from your code to the UI layer
* Sure, you can set properties on a control manually
* but the cleanest and purest WPF way is to add a binding between
  + the source
  + and the destination UI element
* Syntax:

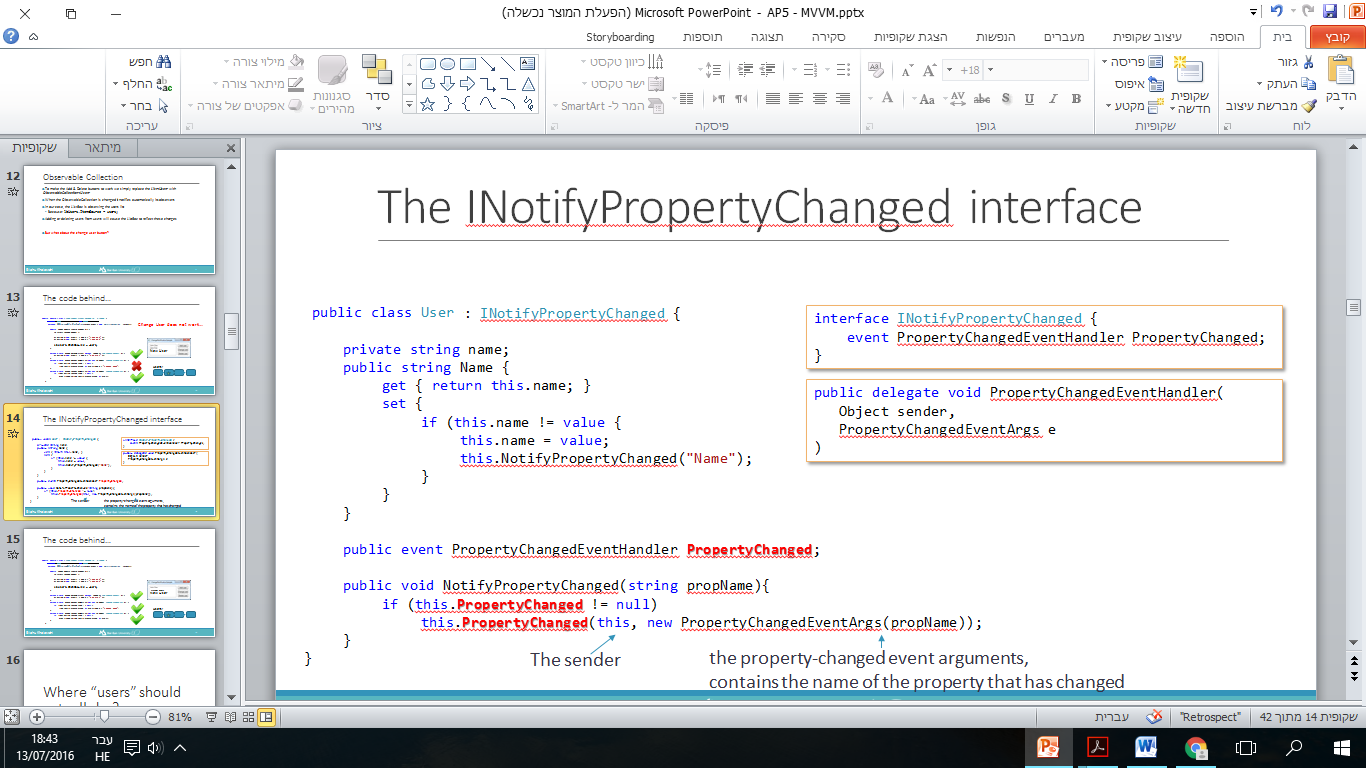
Data Context

ה Data Contextהמקור הדיפולטיבי של הbinding, הוא נירוש דרך ירושת הcontrol,למשל אפשר להגדיר Data Context בשביל החלון עצמו ולהשתמש בו בכל ילדי controls.

Observable Collection

* To make the Add & Delete buttons to work we simply replace the ***List<T>*** with ***ObservableCollection<T>***
* When the ObservableCollection is changed it notifies automatically its observers
* In our case, the *ListBox* is observing the users list
  + Because: lbUsers.ItemsSource = the list<T>;
* Adding or deleting users from *users* will cause the *ListBox* to reflect these changes

 **The INotifyPropertyChanged interface**



The Model

-ממש את INotifyPropertyChanged

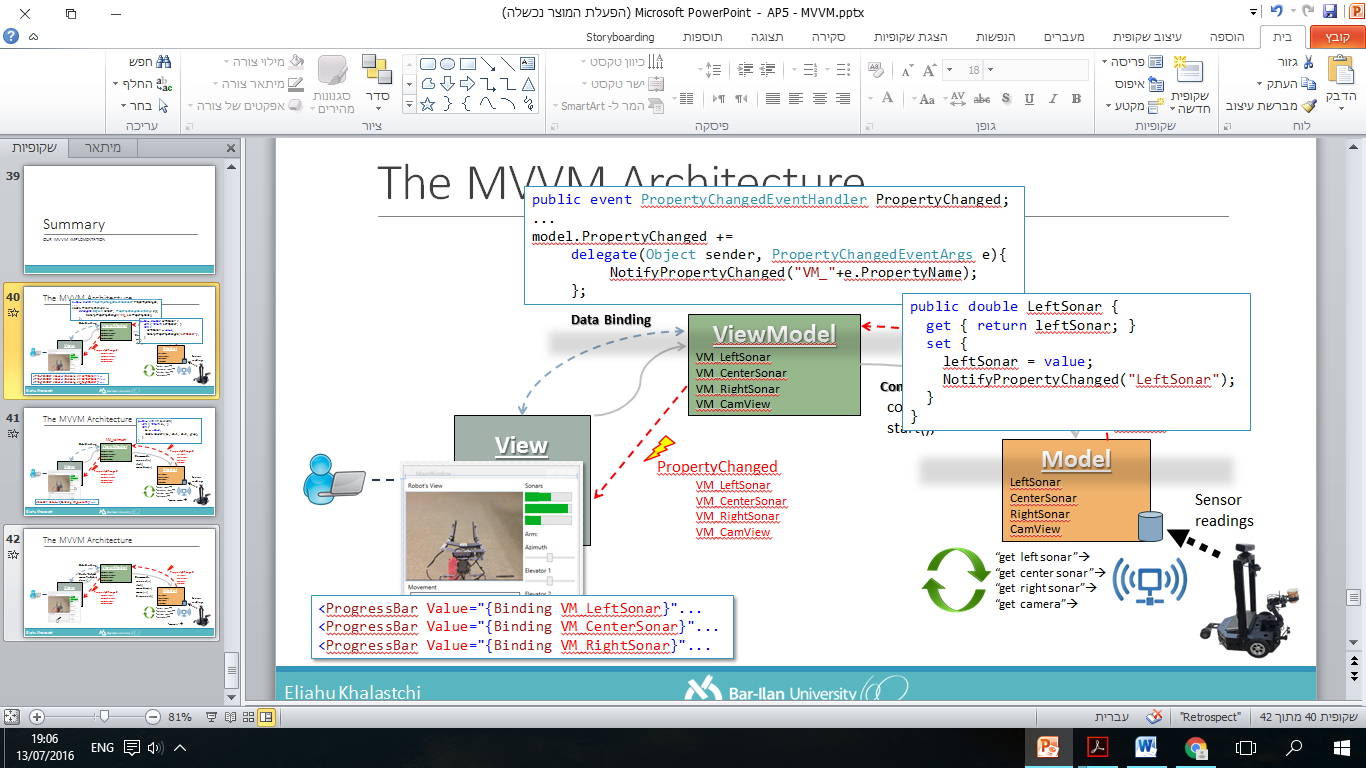
-משתמש במחלקת לקוח במידה ונחוץ תקשורת (לדוגמא עם רובוט)

-בproperty שנרצה לעשות binding נפעיל בset את NotifyPropertyChanged("prop\_name");

The ViewModel

-כדי לפקד על המודל נצטרך ref אליו, נקבל notification ממודל ע"י הוספה לevent של ***PropertyChanged***

***-****כדי להשתמש בdata binding* או notify נצטרך שviewmodel יממש את INotifyPropertyChanged ושיהיה Properties public

 the view-xaml and code behind

-נצטרך להכיר את vm בשביל לתקשר איתו ולעשות binding לprop .

-ה Data Context שלנו יצטרך להיות הvm (ככה נוכל לעשות data binding לpublic properties)

-בXAML פשוט נעשה data-bind לVM public properties

**מצגת 6-Application Servers**

התרשים במצגות הראשונות

נוכל לרשום קוד של ארכיטקטורת MVVM/MVP/MVC אך יהיה לנו בעיה שנצטרך לעשות קומפילציה לכל מ"ע שונה ועקב כך לשנות את הקוד שישתמש בnative libraries .

נוכל לרשום קוד ולהריץ עם interpret אך בחיסרון הוא האיטיות . נוכל להשתמש בvirtual Machine.

אפשר להסיר אחריות מהמודל ונעשה את זה בשרת חיצוני.

שרת היא גם תוכנה המספקת לנו שירותים (services), אנחנו שולחים בקשה (בארכיטקטורה של mvc בשרת הcontroller יפרש את הפקודות שנקבל). לא משנה לנו באיזשהו שפה יהיה השרת. אך desktop application נצטרך התקנה במחשב של הלקוח ובמידה של עדכון התקנה מחדש תהיה נחוצה(הרבה עבודת תחזוקה), והיא לקח עלה הרעיון שהאפליקציות שלנו ירוצו כולם בשרת וצץ הרעיון של אפליקציית web.

נקודה לבלבול בweb app זה שחלקים מהתוכנית **ירוצו אצל הלקוח**.

**Web application**

הקדמה

בצד הלקוח- יש לנו דפדפן, לכל מערכות ההפעלה יש דפדפנים וכולם מדברים באותה שפה, והם מכירים את פרוטוקולי התקשורת השונים, לכן אם נרשום אתר הוא יהיה multi-platform. אז עכשיו במקום אתר נרצה לרשום אפליקציית desktop , השאלה עכשיו אם נרצה לשמור מידע בצד הלקוח (בעזרת ,cookie תישמר ע"י הbrowser ללא תלות במ"ע) או בצד השרת. לקוחות אחרים שיכולים להיות הם הcmd,telnet client.

הדברים החשובים ירוצו בשרת web server נבדיל בין הסוגים שונים של שרתים למשל יש enterprise server שיכיל את המידע או גישה למסד נתונים, למשל השרת שנרשם בש.ב 1. בתוך הweb server יהיה:

web application-יכול לארח יותר מאפליקציית ווב אחת. בעצם מספקת גישה/יוצרת את המשאבים בשרת שנרצה לפי דרישה .

Web Resources-מסמכים של, HTML, PDF, Image XML, JSONחלק מהמשאבים האלה יכולים להיות סטטיים (לא משתנים) חלק דינמיים(נוצרים on the fly).

HTTP-עובד מעל TCIP ממומש ע"פ סוקטים, בדר"כ web app נשתמש בפרוטוקול זה.

למשאב יש URI (Uniform Resource Identifier) הוא המידע הייחודי שלו , url הוא מקרה פרטי וזהו מיקום של הקובץ ובפועל זה אותו דבר.

אם ניצור את עמוד הHTML זה יהיה בצד השרת, אך אם יש java script הדפדפן הוא זה שיריץ את הקוד,כלומר בצד הלקוח .בעצם האפליקציה יש חלק שרץ בצד שרת וחלק בצד לקוח. אם נשחק במשחק זה יהיה בצד הלקוח , אם נרצה לשמור תוצאה למשל אז נפנה לשרת, השרת רק יהיו המשאבים ונפנה במידה ונרצה משאב.

-אפליקציה מורכבת, קוד שרץ חלקו בלקוח אך רובו רץ בשרת.

-הדפדפן משתמש להיות user interface, נקרא לו לקוח כי הוא זה שרוצה לגשת אל האפליקציה

-השרת מספק (מממש) את האפליקציה.

-האפליקציה ווב באה להחליף את אפליקציית הdesktop

**Static** vs. **Dynamic** resources

static

-השרת מעביר קובץ: טקסט, HTML,תמונה וכו'.

-הלקוח מציג את הקובץ.

Dynamic

-השרת מייצר תוכן:

* On the fly
* התוכנית מריצה אותו על הלקוח/השרת

|  |  |
| --- | --- |
| Server side | Client side |
| קוד שרץ על השרת | קוד שרץ על הלקוח |
| הפלט של קוד זה נשלחת ללקוח (להצגה) | בתוך הדפדפן (לדוגמא java script) |
| השרת לא מעבד משאבים דינמיים | יכול להוגש ללקוח כמשאב סטטי  לדוגמאHTML שבתוכו js |

נרצה להבדיל בין web application שהיא stateful לבין אפליקציה שהיא stateless

|  |  |
| --- | --- |
| Stateful | Stateless |
| משאב דינמי יכול להיות stateful | כל משאב סטטי הוא בהכרח state less |
| קוד server-side יכול לשמור מידע על השרת | כל בקשה מקבלת את אותו תוכן, כלומר אם אני אבקש את אותה בקשה כמה פעמים אני אקבל אותה תשובה |
| יש תלויות בין הבקשות | הבקשות בלתי תלויות אחד בשניה |

Web app יש בדר"כ

* גם צד לקוח וגם צד-שרת.
* תוכן דינמי עם אלמנטים סטטים לדוגמא יצירת עמודי HTML עם תמונות ועיצוב סטטי.
* טבע STATEFUL
* יותר מwebserver אחד (מכשירים רבים)

איך נשלב את כל הדברים האלה.

J2EE Application Server Components

Java script- זה שפה דומיית java שהדפדפן יודע להריץ, לכן קוד של java script זה client-side.

JSP-Java server page

קובץ הנראה כמו קובץ HTML לכל דבר עם סיומת jsp אבל המיוחד בו הוא שניתן להכניס בו קטעי קובץ בjava (לא java script) שהשרת הולך להריץ , בעצם הקובץ jsp הוא **משאב** ושהדפדפן יבקש אותו בזמן ריצה אנחנו הולכים להחזיר את הפלט של ריצה של הדף jsp ,צד השרת מייצר לי את הhtml בצורה דינמית, לפי בקשה.(בדף HTML נקבל משאבים סטטים ובJSP דינמים)

אם נעשה view source נראה את הפלט ולא את הjsp כי הוא רץ כבר אצל השרת לעומת זאת אם היינו עושים עם html וjava script היינו רואים גם את הhtml ואת קוד הjava script .

מה נרשום בjs? דברים שאנחנו לא צריכים לערב את השרת (דברי פרזנטציה לדוגמא).

מה נעשה בjsp? ממש לוגיקה של דברים שנרצה לקבל אותם.

Java Servlet

בפועל כל jsp בזמן ריצה מתורגם לservlet

מחלקה שמקבל request and response ובהתאם למידע שהיא מקבלת מrequest היא מייצרת איזשהו response היכול להיות xml,HTML שמכיל js. בדר"כ תגובות HTTP של post/get.

חתימות

**protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {}

**protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {}

מה יותר נוח servlet או jsp? תלוי, יש לשניהם תפקידים שונים.

Enterprise JavaBeans

javaBean- זה מחלקה שניתן לעשות לה serlize , יש לה default cons,set get לכל הdata member שלה , כלומר מחזיקה מידע ורק מידע בלי פונקציונליות מיוחדת.

שנעביר מידע בין המודל לview נרצה שזה יהיה JavaBeans .

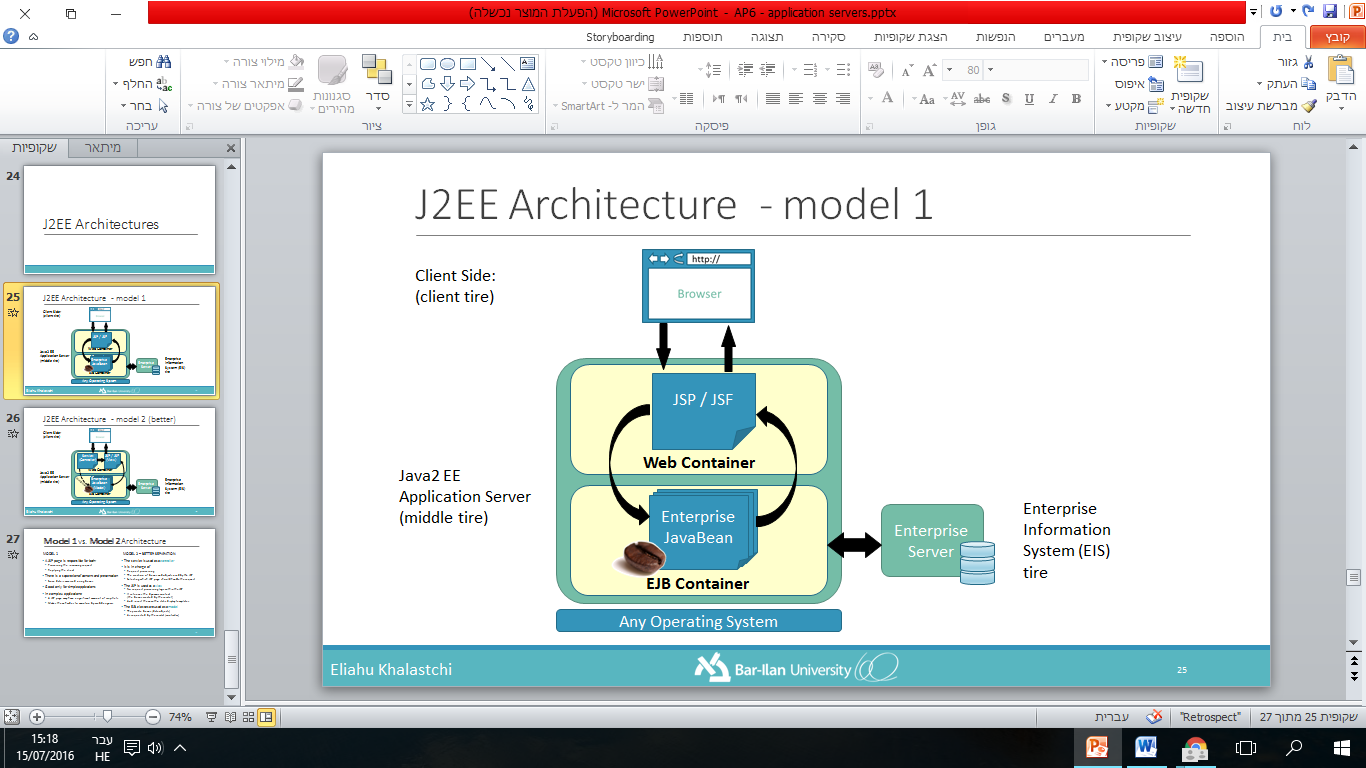
Enterprise JavaBeans זה סדרה של ספציפיקציות שעוזרות לנו לכתוב את המודל העסקי בשרת והסדרה של ה ספציפיקציות האלה מגדירות לנו את כל הרשימה זאת

* Transaction processing: Response and request
* Concurrency control
* Event-driven programming (using Java Message Service)
* Asynchronous method invocation
* Job scheduling
* Interprocess communication using RMI (Remote Method Invocation) and web services
* Security
* Etc.

J2EE Architectures

container זה סביבת ריצה,שיודע להריץ את הדברים הבאים:

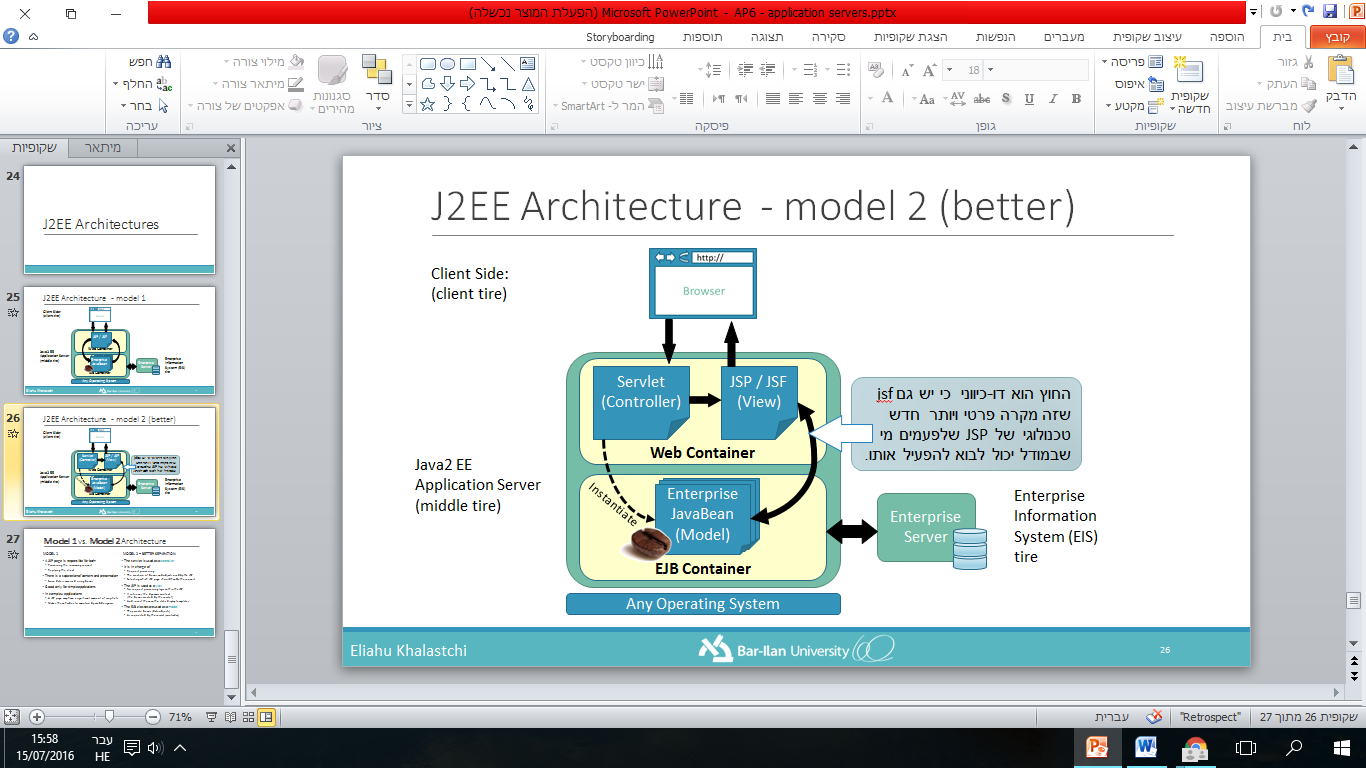
בweb container יש לנו את דפי הJSP ובתוך הEJB יש לנו את הלוגיקה העסקית,זה המודל שלנו.

(ניתן לדלג ישר לטבלה לסיכום שלהם)

בשיטה הראשונה הנקראת **model 1:** האפליקציה נראית כך

הדפדפן מבקש משאב כלשהו, איזשהו JSP ,בדף הזה יהיה HTML וקוד ג'אווה זה יהיה אחראי לכול. גם לפענח את הבקשה גם לטפל בבקשה זאת. כדי לייצר את התגובה שיוצר קוד הג'אווה בjsp נצטרך להשתמש בJavaBeans,לייצר אותם. הם בתורם יכולים לפנות לdata base. למשל פתחתי דף יוטוב אז נתפתח לי דף jsp והקוד אומר "תשים לי סרטונים" אזל איזה סרטונים? צריך לבדוק בdb של YouTube. מידע ששלפנו מDB חוזר בחזרה לJSP ונוצר לי דף HTML עם סרטונים ונחזיר את התגובה לדפדפן.

במודל זה JSP ישמש גם כcontroller וגם כview לכן מודל זה פחות טוב כי הפרדה ביניהם פחות טובה .זה טוב לאפליקציות מאוד קטנות אך ברגע שהאפליקציה גדלה הדף jsp מתחיל לגלוש לתחום הסקריפלטים, המון קטעי ג'אווה . מי שמתחזק את זה הם לא המתכנתים אלא הdesign-ים הכותבים צד הלקוח. תפקידים מתחילים להתבלבל ויהיו סתירות שניתנות לפתרון בקלות אחרת.

**במודל 2** יתאים יותר לאפליקציות גדולות.

נשתמש גם בservlet וגם בJSP,אנחנו מאצילים את האחריות שהייתה במודל 1 לדף הjsp לservlet. כאשר הדפדפן מבקש משאב מסוים הוא מפעיל servlet והוא עושה כמה דברים:

- הוא בעצם יוצר לי java beans ,הוא מפעיל עכשיו את המודל שלנו ומחלקות ג'אווה המייצרות לי את המידע הרלוונטי לתצוגה (אם הן צריכות הם יפנו לdb וישלפו ממנו מידע) .

-מפעיל את הjsp ,למעשה יש לנו כמה דפי JSP והservlet בוחר למי להעביר את הבקשה. בכך הJSP הופך להיות רק view ואחראי רק על הפרזנטציה שלנו,הוא אומר "הינה הJavaBeans כבר מוכנים רק תיקח אותם". הJSP כבר לא אחראי על הrequest כי הservlet טיפל בו.

**האופן שבו אנחנו מייצרים את המידע הוא מנותק לגמרי מהjsp** ,פה הדף jsp שולף את הJavaBeans כי הcontroller הפעיל אותו (הוא מפעיל אותו שהמידע מוכן, אך שאפשר להפעיל אותו לפני שכל המידע מוכן)

|  |  |
| --- | --- |
| Model 2 | Model 1 |
| הservlet משמש כcontroller | JSP אחראי על לעבד בקשות ולענות ללקוח |
| הוא אחראי על :  עיבוד הבקשות,  יצירת של beans ואובייקטים ש הjsp ישתמש  בחירת הJSP שיטפל בבקשה | אין הפרדה של תוכן והצגה: מכיוון שהמידע נלקח ע"י שימוש בbeans |
| JSP משמש כview  אין לוגיקה של עיבוד בקשות בו  לוקח את התוכן הדינמי (הbeans שנוצרו)  שם אותו בתוך התצוגה הסטטית | טוב לאפליקציות פשטות |
| הEJB משמש כmodel :  יוצרים את הbeans,כמו שבוקשו ע"י הservlet שהוא הcontroller | באפליקציות מורכבות דף הjsp יכול להכיל כמות רבה של scriplets מה שגורם לתחזוקה יותר קשה ע"י הweb designers |

**מצגת 7** **Scalability-**

בעולם האמיתי יש לנו הרבה לקוחות, ולקוחות האלה שולחים בקשה העוברת דרך הענן של האינטרנט ואחרי זה תגיע לאיזשהו שרת , שיהיה בדר"כ load balancer,אחד כזה שמחלק את העבודה בין השרתים שווה בשווה (בהקשר הזה הוא סוג של controller) ,הוא יכול להעביר את הבקשה עכשיו לapplication server,אלה שמכילים את הלוגיקה העסקית של אפליקציית הווב (יהיה יותר מאחד כזה), והם בתורם ישתמשו ב:

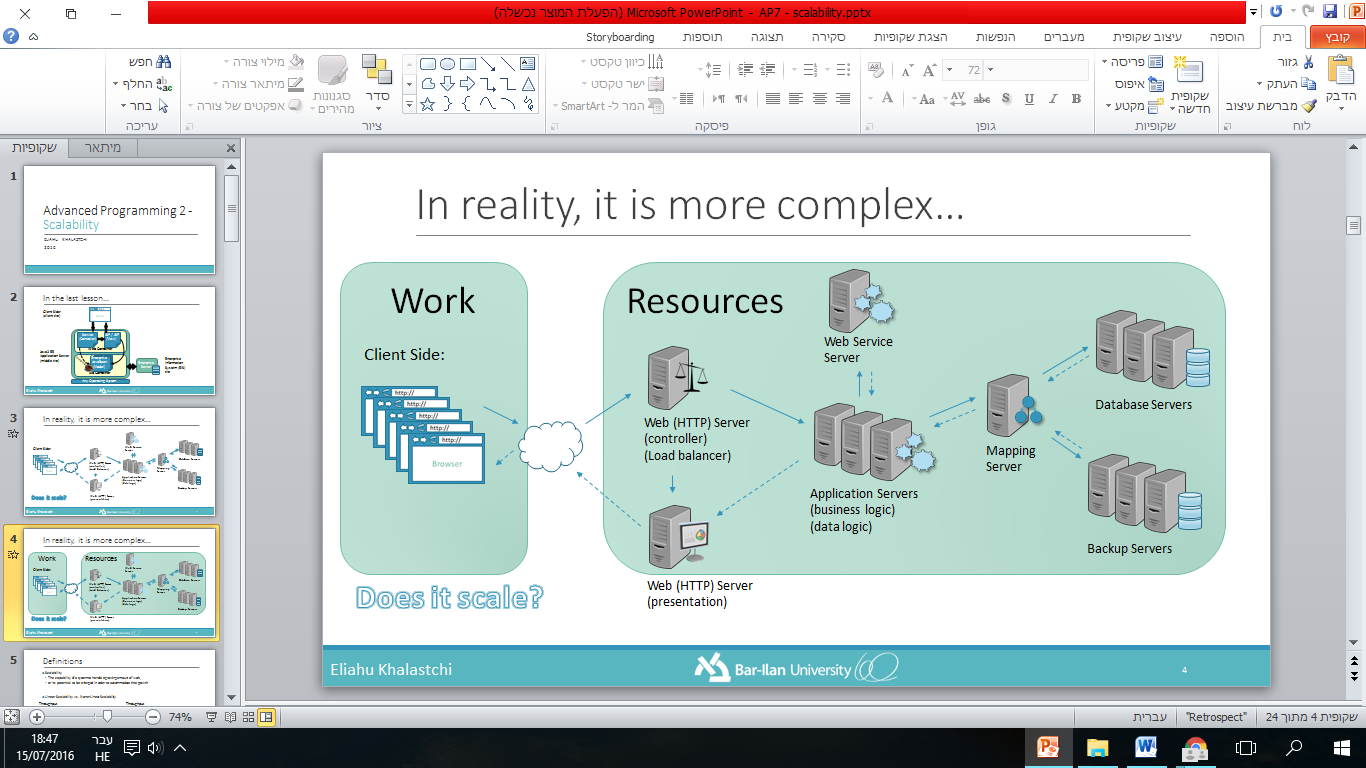
-web services, שנמצאים על שרת אחר , והם יחזירו איזשהו תשובה .

-איזשהו data base, אבל בשביל לדעת איזה לגשת יש לנו עוד שרת שעושה את תפקיד המיפוי,יש איזשהו hash table וניתן לו איזשהו key והוא לוקח מdb ונותן לי את המידע בחזרה, יכול להיות מאיזשהו שרתי גיבוי למיניהם.

ואז הload balancer יכול להפעיל איזשהוweb server ששם יש את הפרזנטרציה, והוא ימשוך את המידע שהweb service יצרו לי וישלח את המידע למשתמש הרלוונטי.

השאלה המרכזית שאנחנו רוצים לשאול את עצמו זה האם הארכיטקטורה שרשמנו האם היא scale-בילית (בעברית scale-בילית זה סולמיות) והרעיון של מה שאנחנו רוצים לשאול את עצמו זה .

יש לנו מי שמייצר לנו עבודה והעבודה יכולה לגדול ,פתאום יש יותר אנשים יותר טרנזקציות יותר בקשות , וכול שהערך הזה גודל (כך ערך שאני בוחר) אז יש לי את המשאבים שבעזרתם אני נותן את הפתרון. כמה אני צריך להגדיל את המשאבים כדי לטפל בכמות הולכת וגדלה של הערך שלי (נגיד בקשות)?

הרצוי הוא Linear Scalability המצוי זה None-Linear Scalability,למה? כי ככול שיש יותר משאבים ,צריך לנהל יותר מאמץ לנהל את המשאבים האלו ולפעמים באיזשהו שלב זה כבר לא משתלם, כלומר הוספתי משאבים אבל זה לא עוזר לי להגדיל את התפוקה שלי.

**הגדרות (בהקשר של web app)**

**Scalability-**היכולת שלנו, כמה מאמץ נצטרך כדי להשקיע במערכת שלנו כך שהיא תוכל לטפל בכמות הולכת וגדלה של עבודה. כלומר כמה דברים נצטרך להוסיף (בהקשר של web, האם נגדיל את הtraffic שלנו),עד כמה קל להוסיף אחסון, כמה טרנדקציות נוכל לטפל במקביל? וכו.

**Performance**- ביצועים. איזה אופטימיזציה של המשאבים כדי לצמצם את זמן תגובה ולטפל בבקשות.

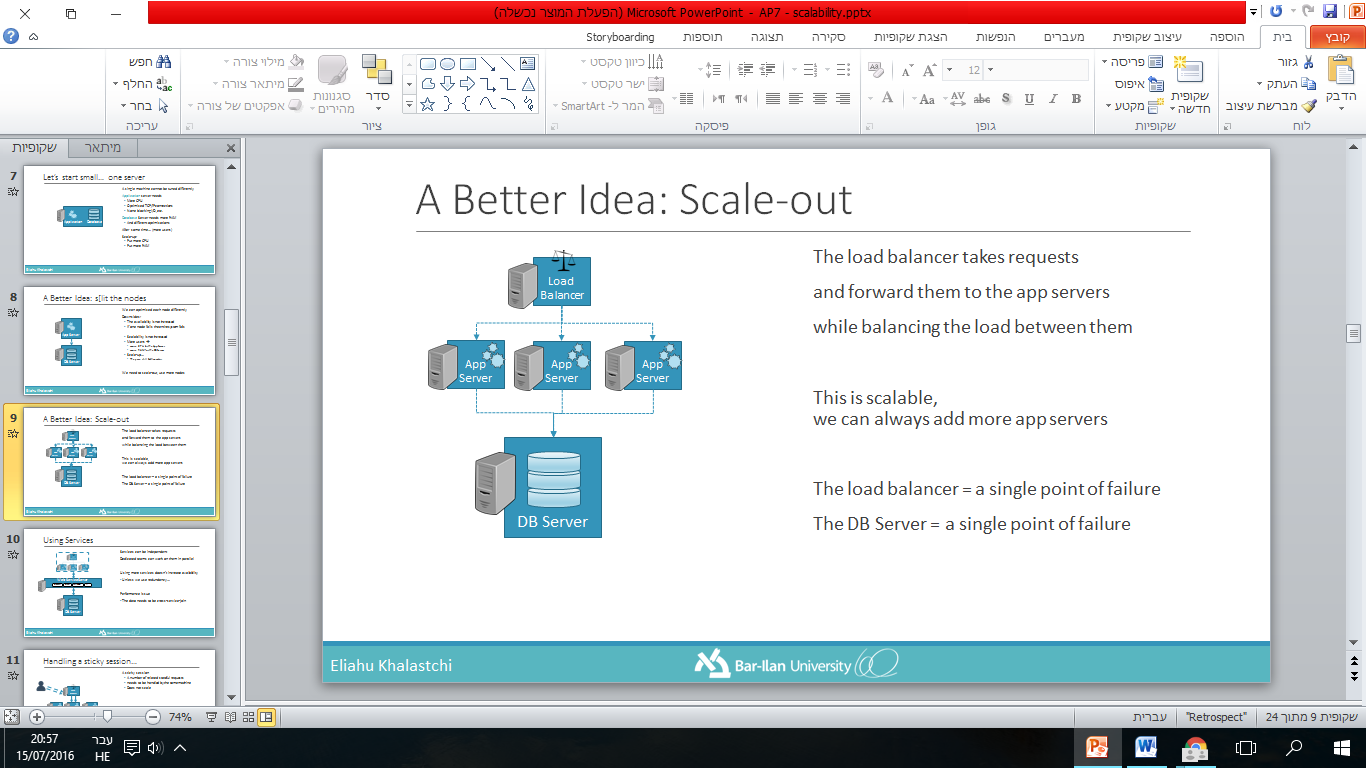
**Availability**-עד כמה אני נמצא שם בשביל הלקוחות שלי, זה קצת שונה מאמינות.

* יתירות למרכיבי המפתח
* זמן התאוששות קצר לקריסה חלקית של המערכת
* Graceful degradation (יציאה מסודרת) שבעיה מתרחשת כלומר אנחנו נשמור את המידע, נשדר ללוג כדי שנוכל לדעת מה בעיה, להראות אולי הודעה כישלון.

פרמטרים נוספים: relatability, manageability, cost.

נתחיל מקטן: שרת אחד מחשב אחד שמרכב הכול, איך יש חסרונות לדבר. כי יש לנו Single point of failure(דבר אחד קרס, הכול קרס) כמו כן הישות הם שונות ,הapplication server צריך יותר CPU,בעוד הdata base צריך יותר RAM. ולכן גם אופטימזציות שונות.

אחד הדברים שיכולים לעשות זה scale-up,להוסיף משאבים למשאבים הקיימים אצלנו. אך זה ניתן לעשות עד גבול מסוים.

אז בוא נפריד פשוט בין הapp server לdb server (עדיין לא scaling-out).עכשיו שמדובר בין 2 מכונות שונות אפשר לעשות אופטימיזציות שונות, אבל לא הגדלנו את הavailability שלנו (אם הdb יקרוס, גם האפליקציה תקרוס) ואת Scalability (אם הגדלת משתמשים עדיין כולם יעברו בapp server ואז ייווצר צוואר בקבוק). מה שאנחנו צריכים לעשות זה scaling-out.

Scale –out:זה להוסיף עוד מכונות, יש לנו כמה שרתים וכולם יודעים לטפל באותו סוג בקשות.

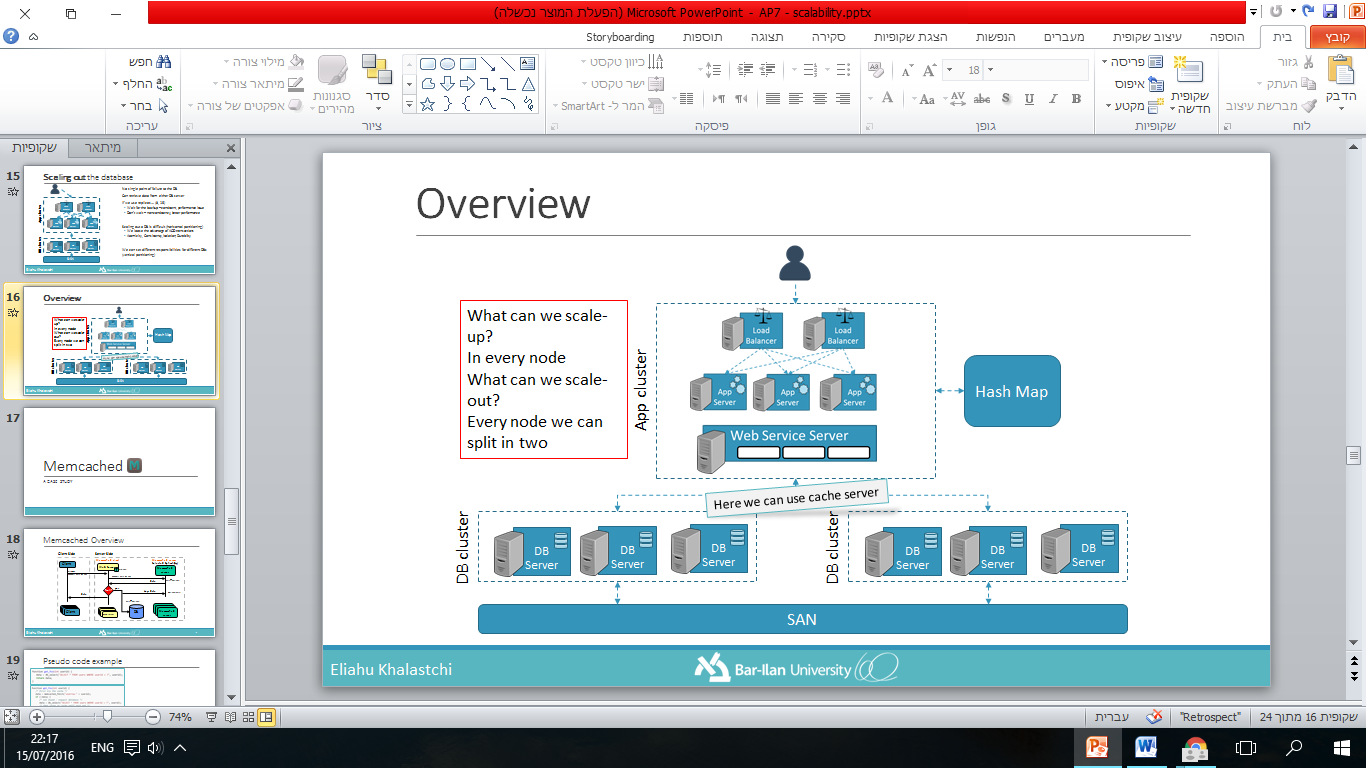
\*כדי שלא כל הבקשות יגיעו לשרת אחד שמנו Load-balancer. נתכנת אותו כדי שידע לחלק את הבקשות בצורה שווה על פני כולם שרק אפשר וכל השרתים האלה יפנו לdata base אחד. זה scalability.אך בפתרון זה שוב יש צוואר בקבוק: בload-balancer ובdata base.

אחד הפתרונות שאנשים מציעים זה והם לא רואים את החיסרון בזה הוא" יש גבול לכמה web service אפשר להשתמש אז בוא נוסיף עוד שרת שיודע לעשות שירותים. שירות אחד יודע לעשות פקודה א ,שירות אחר פקודה ב ואפליקציה רק תשלח לשירות המתאים". אך לא שיפרנו כלום רק הוספנו עבודה יש לטפל בעוד בקשה, במקום שאני אטפל בבקשה במקום אחד העברתי את זה לweb service וחיכיתי שהוא יעביר לי את התגובה בתקשורת ואז יטפל. המהות פה היא יותר קשה, יש לזה יתרונות .למשל אם השירותים בלתי תלויים זה הזה אפשר לחלק את זה לצוותים שונים שכל אחד ישב על שירות אחר במקביל. אך הבעיה המרכזית נמצאת בכך שאם שלחתי הודעה לשירות product,הוא צריך להחזיר לי תשובה שאני מכניסה לשירות costumer. יש קשר בין השירותים האלה ,פלט של אחד זה קלט של אחד . כלומר איפשהו בחוות השרתים צריך לעשות cross-service-join לקחת את כל המידע שהשרתים הביאו לי ולעשות join או איזשהו חישוב. אז הדבר הזה לא scalability כי אם נוסיף עוד שירותים זה לא בהכרח משפר את העבודה, זה אולי מחלק את העבודה יותר טוב אבל לא בהכרח תרם ל scalability .

נפתח איזשהו session עם השרת והגענו לload balancer והוא הטיל את אחריות על app server מסוים, אז שתבוא השאילתה הבא של המשתמש נרצה שהוא יגיע לאותו אחד כי הוא כבר טיפל בו והוא מחזיק את הstate של השרת ויש עוד יתרונות אחרים אך אם נשלח לשרת אחר לא ננצל את כל היתרונות האלה. אך במצב כזה ייווצר בעיה כי אנחנו לא מנצלים את כל המשאבים ולא מחולק ה"נטל" בצורה שווה. למצב כזה נקרא "Sticky Session" כי נידבק לapp server אחד ונעבוד רק אתו כי זה הגיוני שהוא ידבר אתו. הפתרון זה להוסיף עכשיו Central Session Store עכשיו כך פעם שמישהו טיפל בבקשה נשמרו את כל הstate-ים של הsession ישמרו שם והוא מאגד את כל השירות. אך שוב יש לנו צוואר בקבוק ויש לנו מקום של join והפתרון הוא scaling-out אך אם יש לי עכשיו כמה כאלה מישהו צריך למפות לאיפה אני אלך וגם לעשות איזשהו join בסוף והניהול נהיה מסובך.

עכשיו בוא נטפל בload-balancer ,אפשר לעשות scale out להוסיף עוד אחד. אפשר לעשות active/passive אם אחד יקרוס השני יעבוד או שיעבדו יחדיו (יהיה טבלת hash קטנה שתמפה) וכך אין לנו single point of failure .הפתרון הזה הוא scalability.

יש לנו עכשיו חוות שרתים שהיא מדברת עם הdata base שלנו ,אך יש בעיה ששוב זה יהוו צוואר בקבוק שלא ניתן יהיה לפתרון בscale out אפשר לעשות scale up . בסופו של הdata base הוא רק אבסטרקציה והמידע נשמר בעצם בקבצים , בכמה hard-disc . יש לנו איזשהו file system של SAN,זה רשת מחשבים בשבילנו זה מערך שלם של דיסקים שמרושתים באיזשהו רשת מהירה וכל הרעיון הוא שהם מספקים לי מערכת קבצים אחת. לא אכפת לי לאיזה hard disc זה ייפול. זה scalable ודבר שיותר קל לעשות אותו מאשר מספר שרתים. בוא ננסה לעשות scaling –out לזה, יש לנו מספר db שכל אחד מחזיק חלק **אחר** של אותו הdata base שמדברים מעל איזשהו san ששם כל הקבצים שלנו נשמרים. נצטרך לדאוג למיפוי שלהם כמו כן נעשה select \* From Table לא יהיה לי משנה שחלק מהמידע יושב פה וחלק שם. בעצם יש לנו db אחד שנמצא על פני כמה מחשבים יש לנו single point of failure במיפוי וגם מחשבים האלה יכולים להיות כגיבוי. במקרה של גיבוי שיש לנו master אחד וכל השאר גיבוי יהיה בעיה, כי אם נחכה שהמידע ישמר בכולם זה יהיה עקבי consistent אבל יהיה בעיות ביצוע מצד שני אם לא נחכה ונחזיר תשובה ישר אחרי כתיבה למאסטר זה יהיה non-consistent אך ביצוע טוב יותר. אפשרות אחד זה לחלק את הטבלות באופן שונה לפי חשיבותם, היתרון הוא שלא מגבים ממש את הכול אלא מגבה את מה שחשוב ושם נחכה ואיפה שפחות לא נחכה לגיבוי (נגבה אבל). עוד דבר שעושים בחברות זה שאומרים שdata base מסוים יטפל במיליון המשתמשים הראשונים אחרי מיליון נוסיף עוד data base חדש וזה פתרון scalability נוסיף שרתים לפי הצורך. אך מה קורה במידע ונופל לי שרת אחד לאיפה אני אמפה את כל האחרים.



Cache ישמר נגיד אם אני ביקשתי בקשה ומשתמש ביקש אותו בקשה אז ניתן לשמור שם את המידע .

**Memcached**

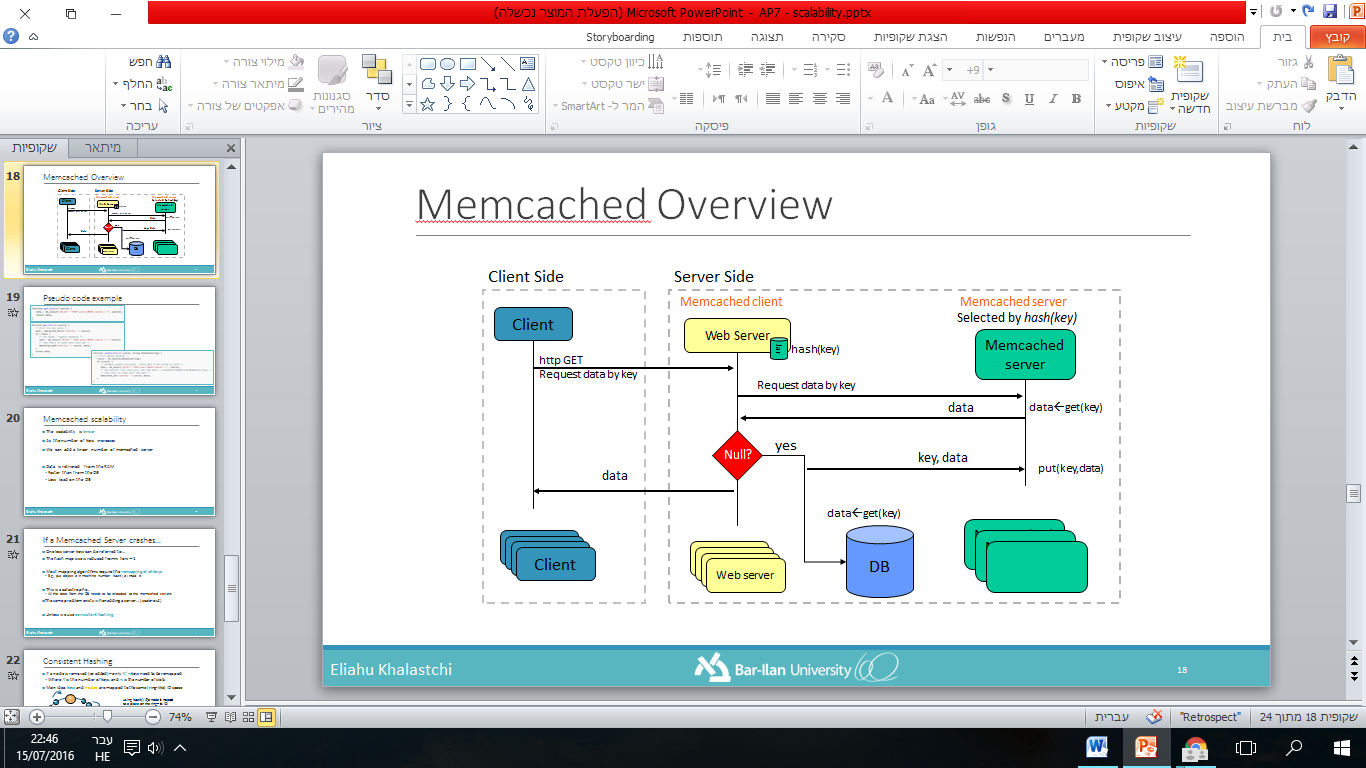
הרעיון הוא מאוד פשוט, Memcached server כל תפקידם זה לשמור אם היו שאילתות שחזרו על עצמם בdb אז אני מאחסן אותם בRAM של השרתים האלו, הם שרתי RAM . שמגיע שאילתה אני אבדוק האם היא קיימת שם ואם לא אני אשלם את המחיר הכבד ואשלוף את זה מהdata base. למשל הלקוח שלח בקשה בget אז מפעיל איזשהו web server שצריך לקחת משהו מdata base אך לפני כן הוא לוקח את הkey ונפעיל אליו איזשהו פונקציית hash שתגידי לי לאיזה Memcached  
server צריך לפנות, מעביר אליו את הבקשה, בעצם הweb server יהיה client לmemcached server והוא מחזיר לי את המידע ומעבירים אותו ללקוח אך יכול להיות שאין וזה Null,אם כן נצטרך לשלוף מהdatabase במחיר יקר יותר וגם נצטרך להשים אותו בmemcached server שאליו נגיע דרך פונקציית הhash. אך מה קורה אם אחד מהם קורס? עכשיו מיפוי לn שרתים יהפוך למיפוי n-1 ונצטרך לעשות מיפוי מחדש. כמו כן פחות מפתחות יכולים לגשת אליהם.

למה זה טוב?

-scalability היא לינארית כי כול שמספר המפתחות גדל אנחנו מוסיפים מספר לינארי של שרתים.

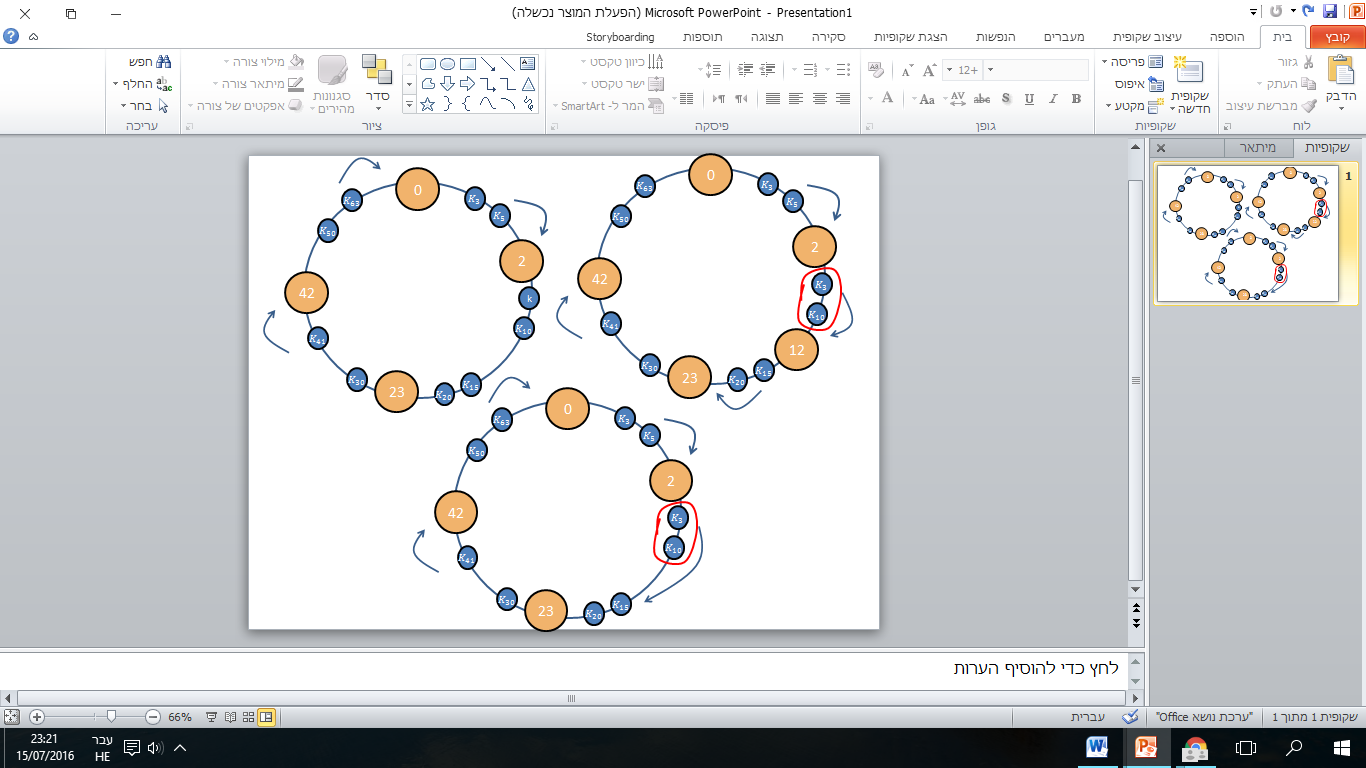
-היתרון של לשלוף מRAM,מהיר יותר מdb וכך יש פחות עומס על הdb.

נחזור לקריסה. רוב האלגוריתמים של mapping ידרשו מיפוי מחדש של כל המפתחות כולל גישה לdatabase ולהשים את זה memcached server מחדש. אותה בעיה תהייה אם נוסיף שרת (scale-out).



**Consistent Hashing**

-אם שרת אחד נפל (נוסף) רק k/n מפתחות צריכים לremapped. גם לשרתים יש איזשהו ID שמתקבל מפונקציית hash וגם למפתחות יש id ששוב מושג ע"י hash . כל שרת מגיע לאיזשהו טבעת כתומה במעגל, על אותו מרחב של המפתחות אנחנו גם ממפים את המפתחות (כחול), כל מידע. מפתח יושב לו באיזשהו מקום אחר על המעגל. הרעיון הוא שברגע שנרצה לדעת איפה נמצא המפתח של k3 אנחנו נלך לחפש על הקשת. לכן שיפול/יתווסף שרת רק נפל בין מה שיש בקשת שבין השרת החדש לשכנו החדש (נגד כיוון השעון) או בין השכנים של השרת שנפל (אם כיוון השעון) .



מה קורה אם 2 שרתים נופלים קרובים אחד לשני (נגיד 2 ו3 )ואז החלוקה תהיה לא שווה?

פשוט נוסיף את הid של 2 כמה פעמים על המעגל. כך העומס על השרתים יהיה יותר שווה כי באמת אם אחד מהם יקרוס זה יותר קרוב לk/n מפתחות שצריך להעביר אותם.

דוגמא ללא consistnet hashing: מודולו n

דוגמא לכן: hash גם לשרתים וגם למפתחות, נציב את השרתים ואז נציב את המפתחות ולפי כיוון השעון נחליט למי הוא שייך .

**מצגת 8-databases –מהסיכום של ליאל**

\*בטבלה השדות זה ממברים וכל שורה זה מופע של האובייקט

**ADO – Active Data Objects**

זה סט מונחה עצמים של ספריות שמאפשרות תקשורת עם מקור מידע ,בדר"כ מסד נתונים. הספריות הללו קרויות data providers.

קודם כל אנחנו צריכים לצור אוביקט חיבור האומר לקוד הado.net עם איזה מסד שנדבר

גישה למסד נתונים: אפשר להתחבר ולשאול שאילתות!

אוביקט מסוג connection: כל הזמן שהconnection פתוח זה גוזל משאבים. מתקבל בבנאי string עם כל הפרמטרים הדרושים להתחברות.

Reader: התשובה נמצאת בו, סוג של מילון.

מה זה בדיוק הreader? נותן stream של מידע שחזר ממסד הנתונים. אי אפשר לחזור אחורה.

Disconnected:

מה שנעשה זה נשלוף את המידע ונשים אותו אצלנו בram . שנרצה לשמור את המידע נפתח את הconnection נשמור ונסגור אותו.

יש לנו data set שיכיל את הטבלאות. הם סוג של connection.

Table adapter :יפתח את התקשורת רק כאשר נרצה לשלוח את השאילתא ויסגור את התקשורת. המידע יישמר בdata set עד שנעדכן דרך הtable adapter נפעיל את הפונקציה update.

נעשה את העבודה לוקלית ואז שנרצה לעדכן את המסד נפתח את החיבור,נוסיף ונסגור.

ORM-מיפוי אובייקטים לסכימות רלציוניות .

Hibernate- נותנת להגדיר את המיפוי בין טבלה לבין מחלקה , נותנת שפת שאילתות מכוונת עצמים.

אפשר דרך קובץ xml להגדיר מיפוי בין אובייקט לעמודה.

חסרונות: שיותר מידי אנשים משתמשים זה יכול להיות צוואר בקבוק שכולם משתמשים בזה עם כמויות גדולות.

**Acid**

* **Atomicity**- כל פעולה (קריאה וכתיבה) צריכה להיות אטומטית כלומר אם החלק נכשל אז הכל נכשל .
* **Consistency-** עקביות המצב של הטבלה תמיד יהיה אותו דבר במידה והמידע לא שונה , כלומר תמיד שנבקש את המידע נקבל אותה תשובה .
* **Isolation**- בידוד-אם נריץ סדרה של פעולה אז תשובתם צריכה להיות זהה לאם היינו מפעילים אותה לבד.
* **Durability**- ברגע שפעולה נעשה ,היא תישאר גם אם נקרוס.

**Big data**

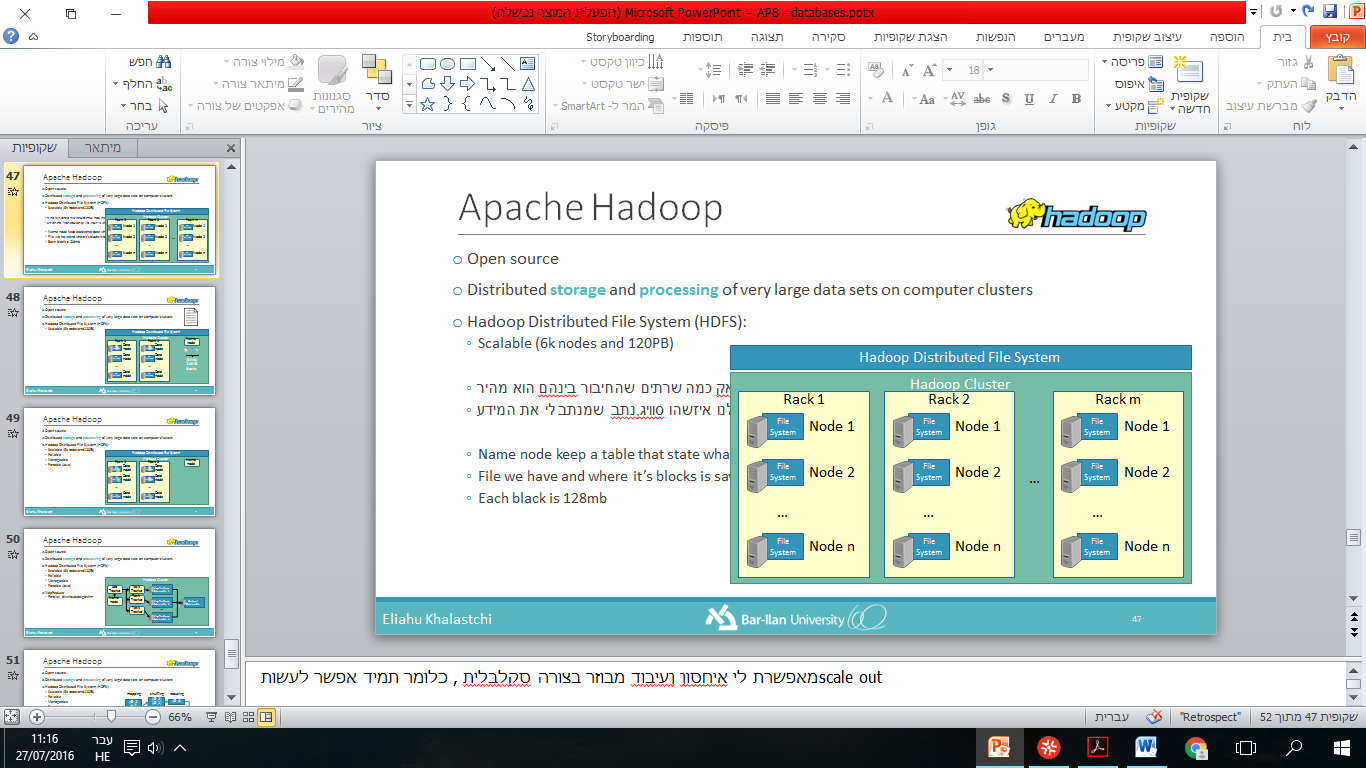
Big data מאופיין בחמשת הV:

* **Volume**- נפח/כמות עצומה של מידע
* **Velocity**-המהירות שבא הוא נוצר או המהירות שאנחנו נדרשים לעבד כמות גדולה של מידע.
* **Variety** -שונות ,לא כל המידע מאותו סוג
* **Value**-כמה נוכל לחלץ מידע יעיל מהdata
* **Variability / Veracity-**מימנות, יש לנו פיסת מידע פה ופיסת מידע שם והם סותרות אחד את השניה והם לא עקביות שזה עוד אתגר בbig data.

לכן נרצה לוותר על sql ולמצוא מערכת שקל לעשות לה scaling-out.

**Hadoop**

Hadoop Distributed File System (HDFS)

מערכת קבצים מבוזרת, מאפשרת עיבוד מבוזר של כל המידע בצורה scalability בצורה של scale-out.

איך זה עובד?

יש לנו node (שרת). וכל כמה שרתים מקובצים לrack מסוים. יותר מהיר יהיה להעביר מידע מאותו הrack. כל אוסף של racks יקרא cluster. יכולים להיות כמה כאלו ומעל כל זה יושבת הhadoop file system.

נותנת תחושה של מערכת קבצים פשוטה.

נגיד שיש לנו קובץ ענק, איך נשמור אותו? יש לנו שרת שמחזיק אותו –name node מחזיק טבלה של כל הקבצים ושבה רשום איפה כל בלוק נשמר (גודל של 128MB).

יתרונות:

* אפשר למנות למקומות שונים לאו דווקא ברציפות.
* לכולם יש אותו גודל,אין בעיה למלא חללים.

הבלוקים צריכים להשמר –שומרים בלוק בnode מסוים ושמורים 2 גיבוים בrack אחר. כל בלוק נשמר על 3 מחשבים ובrack2.

היתרונות:

* אם node נפל אין צורך לטפל מידי יש גיבוי
* ניתן לניהול
* כתוב בjava

איך לנהל? בעזרת map reduce

לעבד בצורה מבוזרת איזשהו מידע על קבצים ענקיים. פיצול של המשימה לחלקים שונים

כלומר המשימה מגיעה לjob tracker המטיל משימות על tasks tracker שונים והם מטילים את המשימות על node data שונים. כל המשימות קוראות במקביל.

הtask tracker מטיל משימה על שרת מסוים ע"פ המידע שנמצא שם,הבלוק שאנו עובדים עליו כדאי שיהיה באותו rack.כלומר המשימה מגיעה לשרת בהנחה שהמידע נמצא שם.

כל בלוק מבצע את המשימה אצלו והכל קורה במקביל ואז נאחד את הכל.

**מצגת 9-Web Services**

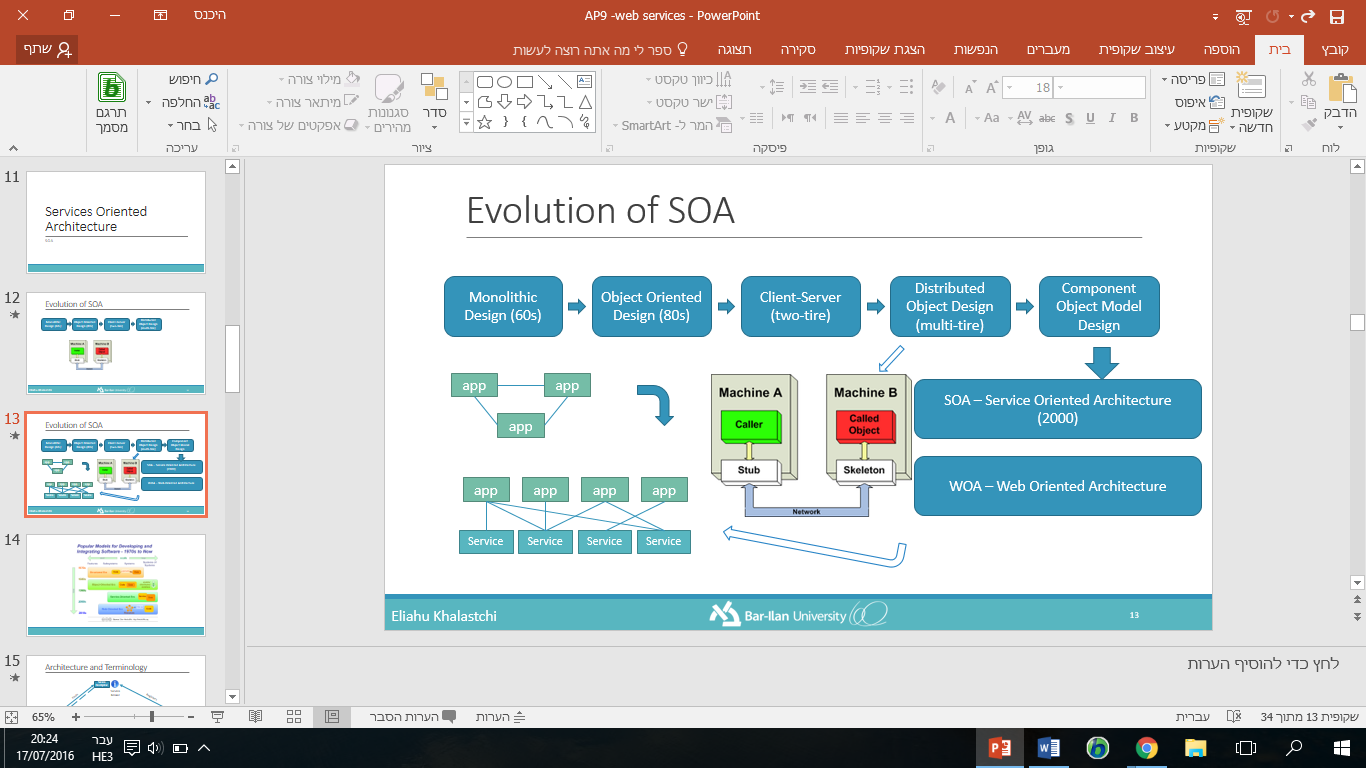
אז נדבר על web services בתור איזשהו שרת שמספק לי איזשהו שירותים ואנחנו רוצים לראות 2 דוגמאות קוד אחת בארכיקטורה של Services Oriented Architecture ואחת Representational State Transfer (REST).

לכן קשה לטפל בbig data בארכיקטורה שהצענו וכל הסיפור עם השרתים אפשר להחליף עם טכנולוגיות כגון hadoop. זה דבר שקל מאוד לעשות לו scaling out.

הפעם נרצה להתרכז בweb services, לפי הסכמה פה האפליקציה משתמשת בweb services אך זה יכול להיות גם אפליקציה שיושבת בצד של הלקוח יכול להיות גם user שבא ומתמש בservice אבל ללרוב נדבר על מערכת שמדברת אם מערכת אחרת.

דבר ראשון שנכיר זה

**Services Oriented Architecture**



**חץ לבן זה דוגמאות**

**רקע**

בעבר , שנות ה60,כל מחשב היה stand alone ולא היו מחוברים אחד לשני וקראנו להם מערכות מודלריות ואז בא הרעיון לרשת אותם לניצול משאבים אך בשביל זה נצטרך פירוק והרכבה ,oop נוצר ונחלק את הקוד שלנו ויש לנו מחלקות קטנות שמחזיקות גם מידע וגם פונקציונליות וכל אחד אחראית לפונקציונליות של עצמה. אך צריך לדאוג לתקשורת (לקוח-שרת) ועכשיו הרבה לקוחות יכולים להשתמש באותו שרת והשירותים שהוא נותן.לאט לאט ההדגש עבר לצד השרת.

Distributed Object Design (multi-tire)

כמו למשל מכונה וירטואלית בג'אווה אוRMI . דיאגרמה מצגת tier זה שכבה פיזית (נמצאת על מחשב שונה) בעוד layer זה שכבת קוד (model לדוגמא) .

Component obj design-טכנולוגיה של מיקרוסופט שדי נזנחה פרט לקטעי קוד שעדיין יש בהם שימוש. מזה נולד דברים רבים (אפשר לטעון .net הגיע מזה)יש לנו אובייקט ,לא חשוב באיזה שפה הוא נכתב . הוא נותן לנו כל מיני שירותים. בוא תעטפו אותו עם אובייקט שחושף פרוטוקול תקשורת לפי סטדנרטים קבועים אז אנחנו מתקשרים עכשיו באמצעות תקשורת עם האוביקט הגדול ובזה מפעילים את האובייקט המוכל ואז אנחנו לא תלוים בפלפורמה ,לא בשפה ולא בשום דבר כי אנחנו שלחנו מחרוזת לצורך העניין וקיבלנו שירות בתמורה.

הדבר הזה הוליד לקראת שנות ה2000 דבר שנקרא SOA-Services Oriented Architecture.

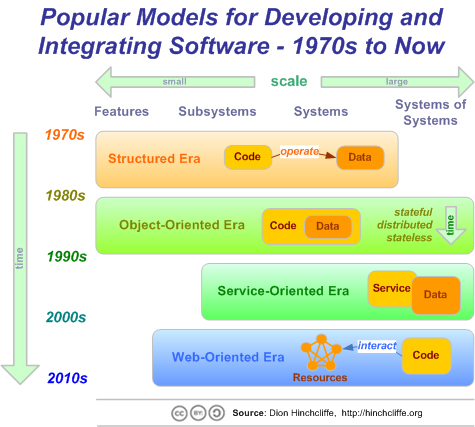
מה הרעיון?

אם תשימו לב ,אנחנו עברנו מסתם אפליקציות מבודדות שאולי יש קשר בינהם ומנסות להפעיל שירותים בודדים הגענו למצב שיש לנו את האפליקציות ויש לנו כל מיני שירותים,על שרתים ומחשבים שונים,וכל אפלקצית משתמשת בשירותים שמתאימים לה. זה **SOA** .

יתרונות

* זה חוסך כפילויות אך יותר עמוק מזה שהרבה יותר קל לי לפתח אפליקציות ,אם יש לי שירותים קיימיים אז אני כמפתח מרכיב "לגו". אני משתמשת במה שצריך ושהצורך עולה גם נכתוב שירות חדש לאפליקציות.
* האפליקציות לא מחוברות אחת לשניה הם מחוברות לשירות
* אין תלות של פלטפורמה
* בארגונים גדולים יש לכל אחד את התשתית מחשוב שלו במקרה של איחוד צריך לחבר בין יחידות המחשוב האלו. ברגע שבאנו וחילקנו את כל הקוד של הארגונים לשירותים קטנים הרבה יותר קל למזג/לחלק חברות ככה כי רק נצטרך לאחד או לקחת את שירות הנחוץ או אפילו להחליף שירות. כל ה enterprise שלנו הופך להיות הרבה יותר מודולרי.

בשנות העשרה (2010s) אנחנו עוברים לwoa-web Oriented Architecture . לא כולם עושים הבדל בין SOA לwoa. גם למרצה קשה לנסח את הבדל בין שניהם. בדר"כ בעולם האמיתי מייחסים אליהם כאחד אך קיימים כאלה המנסים בדקויות להפריד בינהם, גם אנחנו ננסה לעשות ניסיון להסביר מזה מה.

****

**מושגים**

Server requester: לא בהכרח משתמש יכול להיות גם תוכנה אחרת ,זה משהו שמבקש שירות

Web service provider-הוא סיפק שירות מסוים או מספר שירותים, מישהו צריך לדעת איזה שירותים הוא מספק אז יש לנו שרת נוסף שנקרא **service broker** (מין דפי זהב של שירותים) ושם הוא חושף איפה הוא נמצא(iP,port),איך לפנות אליו,איזה שרותים הוא מספק,מה התוצאה של כל שירות, ממש תיאור של השירותים שהוא מספק.

הserver requester מכיר את הserivce broker שם הוא מחפש את השירות שמתאים לו ומקבל את המידע שהוא צריך וכעת הוא יכול לפנות עכשיו אל הweb service והוא יכול לפנות בבקשה אליו ומקבל ממנו תשובה (response).

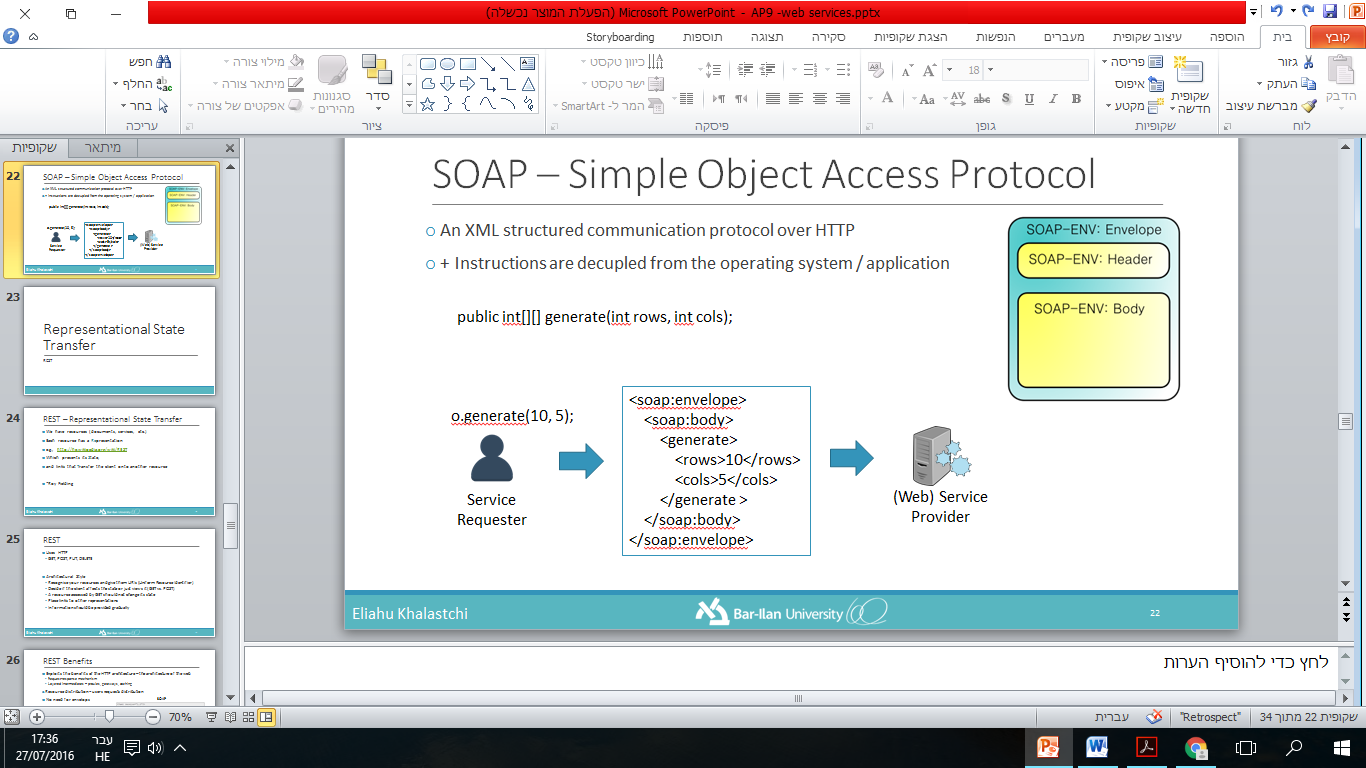
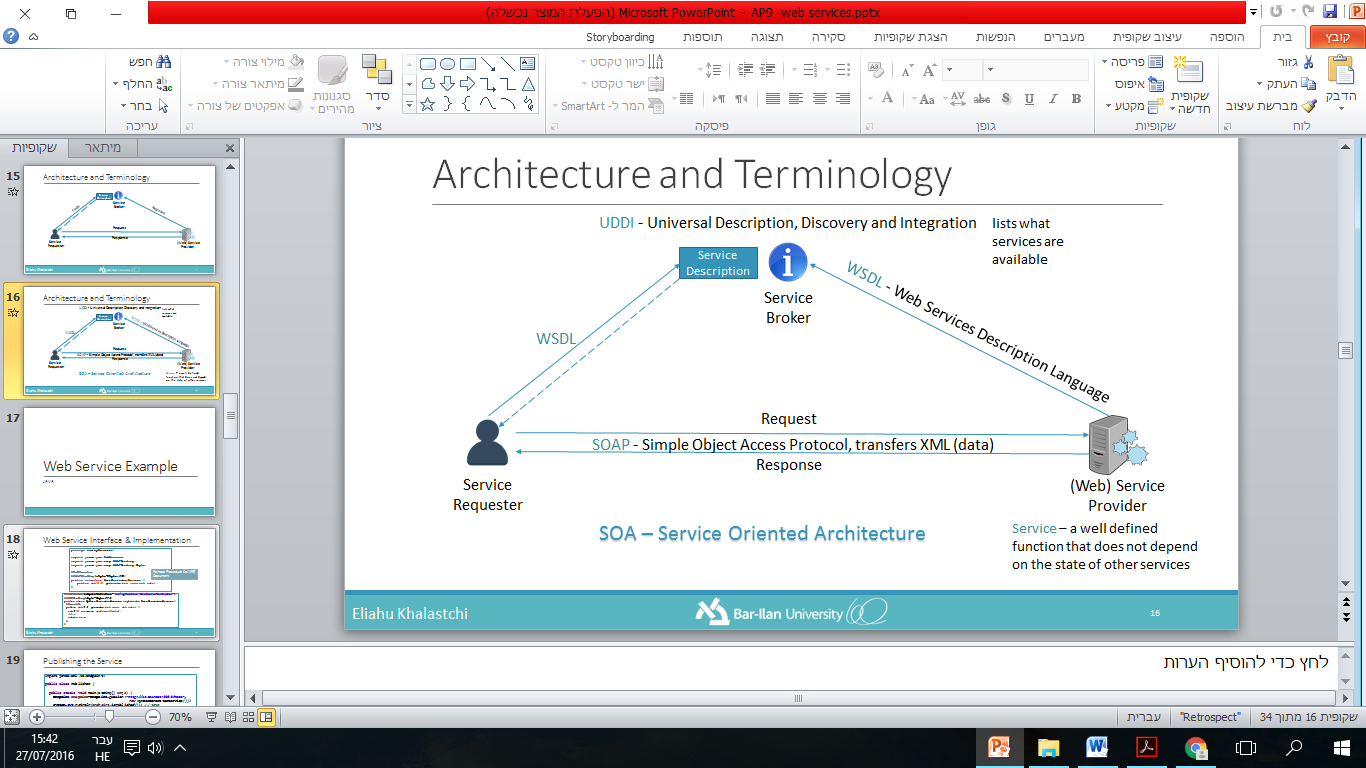
עכשיו ראשי תיבות:

**UDDI - Universal Description, Discovery and Integration** -השרת הזה שמחזיק את כל ה"דפי זהב" הוא רושם את כל השירותים שיש לנו ,מה כל אחד עושה .

**WSDL - Web Services Description Language** השפה שאיתה אנחנו עושים את כל השירותים האלה, web service מתאר את השירותים בשפה הזאת ,server req יודע באיזה שירותים מדובר גם לפי שפה זאת. בגדול אנחנו מדברים על איזשהו XML עם תגיות ידועות וככה אנחנו מתאמים בדיוק איזה שירותים השרת הזה נותן.

servier provider יעביר את המידע למבקש השירות דרך SOAP

**SOAP**-(Simple Object Access Protocol, transfers XML (data גם איזשהו סטייל של XML איתו אני מעביר את המידע ממקום למקום, הweb שולף מתוך את הפרמטרים ומפעיל את השירות הרצוי.

**SOAP**

זה איזשהו XML מגדיר 2 קטעים אחד זה הheader שלו והשני זה הbody. צורת XML פרוטוקול תקשורת מעל HTTP. ההוראות decupled ממערכת ההפעלה/אפליקציה .

עכשיו לסטייל שונה לגמרי, עד עכשיו דיברנו על ארכיקטורת של שרת.

**REST – Representational State Transfer**

עכשיו בשנים האחרונות יש מעבר חד מSOAP לrest.

* יש לנו משאבים רבים (מסמכים,שירותים וכו').
* כל משאב יש Representation (הכתובת שלו URI) לדוגמא <http://he.wikipedia.org/wiki/REST> זה לא באמת דף אלא ה/rest הוא בעצם פרמטר שהפנה אותו למקום המאים,שם פעל איזשהו web service והוא בתורו שלח את המידע של הערך לאיזשהו מסד נתונים ונוצר איזשהו HTML שיחזור
* המיצג מצב state מה שהשירות הזה נתן לנו/מידע שסיפק/הפונקציונליות שהוא סיפק לנו
* וקשרים שtransfer את הלקוח למשאב אחר

ארכיטקטורה שיושבת מעל HTTP יש לנו PUT,POST,GET,DELETE.הוא לא הסטנדרט שמגדיר מה הם התגיות . זה איזשהו ארכיטקטורה/סגנון.

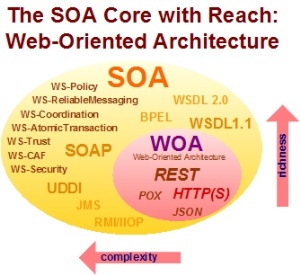
Architectural Style

1. כל דבר שנחוץ למישהו נתייחס אליו בתור משאב וניתן לו כתובות מיוחד URI
2. להחליט אם הלקוח יוכל לשנות את המשאב (post) או רק צופה בו (get לא משנה את state).
3. כל המידע לא מגיע בבת אחת, מביאים בהדרגה באמצעות לינקים.

**יתרונות**

* בין הלקוח לשרת יש שירותים שאותם אנו מנצלים (כמו Cache)
* יש מידע שהוא קל יותר-כתובות.
* מפזרים את הבקשות של המשתמשים השונים.

|  |  |
| --- | --- |
| SOAP | REST |
| פרוטוקול תקשורת | אריכטקטורה |
| חושפים מתודת,מתאים לשירותים | חושף משאבים שמייצגים DATA |
| משתמש בPOST | אפשר לבחור בין GET/POST/DELETE |
| המטרה ליצור הבדלה בין מי משתמש בשירות לבין מי שמספק את השירות | תקשורת יותר פשוטה |
| תמיד XML | אפשר לשלוח כל דבר |
| גם STATEFUL וגם STATELESS | חשוב שיהיה STATELESS מכוון שאם הייתה תלות בין הבקשות נצטרך תמיד להפנות ליעד האחרון ולא ננצל את השימוש ב Cache ואם אין תלות נוכל לשמור את המשאב |

SOA vs WOA

WOA זה סוג של SOA

SOA נותן שירותים יותר גדולים-הפלט יכול להיות כל דבר

WOA נותן חישובים יותר קטנים-הפלט הוא לפי הפרוטוקול.