DOTNEXT
.NET CONFERENCE

ALFONSO GARCÍA-CARO

UI: FROM IMPERATIVE TO FUNCTIONAL

WHO AM I?

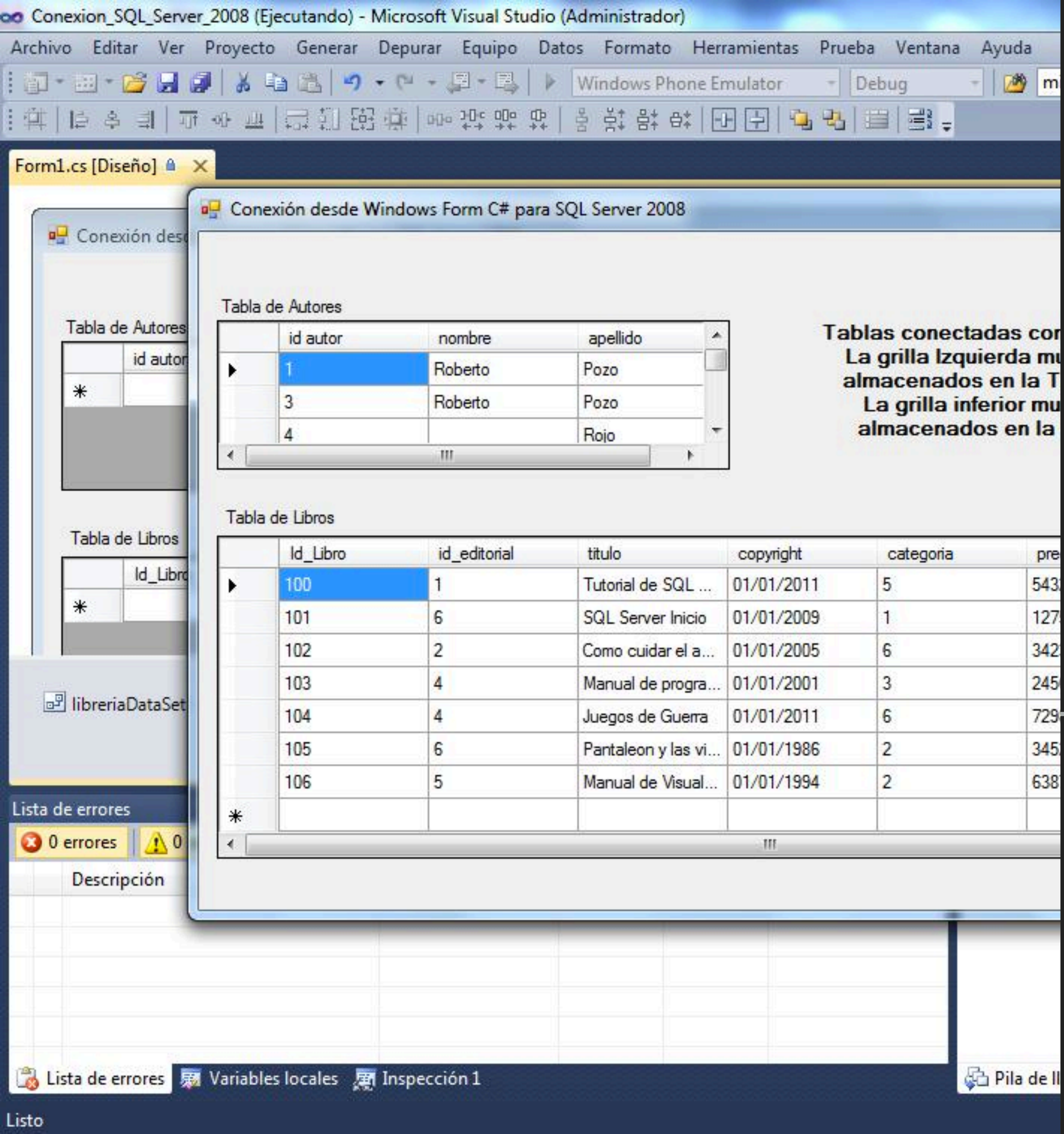
- Alfonso García-Caro
- Degree in Linguistics, self-taught programmer
- Most experience in desktop and web applications
- Experience in multiple sectors: Videogames, Education, Green Energy, Commerce, Genetics
- Creator of Fable
- Coauthor of *Mastering F#*, Packt Publishing

WHAT DOES A UI: LOW LEVEL

- Retrieve data from a source
- Render pixels on screen
- Interpret signals from computer peripherals
- Update data
- Goto 1

WHAT DOES A UI: HIGH LEVEL

- Retrieve data from a source
- Display **familiar controls** (button, text input…) on screen
- React to **events** from controls
- Update data
- Goto 1


Conexion_SQL_Server_2008 (Ejecutando) - Microsoft Visual Studio (Administrador)

Archivo Editar Ver Proyecto Generar Depurar Equipo Datos Formato Herramientas Prueba Ventana Ayuda
Windows Phone Emulator
Debug
Form1.cs [Diseño]

Conexión desde Windows Form C# para SQL Server 2008

Tabla de Autores

id autor	nombre	apellido
1	Roberto	Pozo
3	Roberto	Pozo
4		Rojo

Tablas conectadas cor
La grilla Izquierda mu
almacenados en la T
La grilla inferior mu
almacenados en la

Tabla de Libros

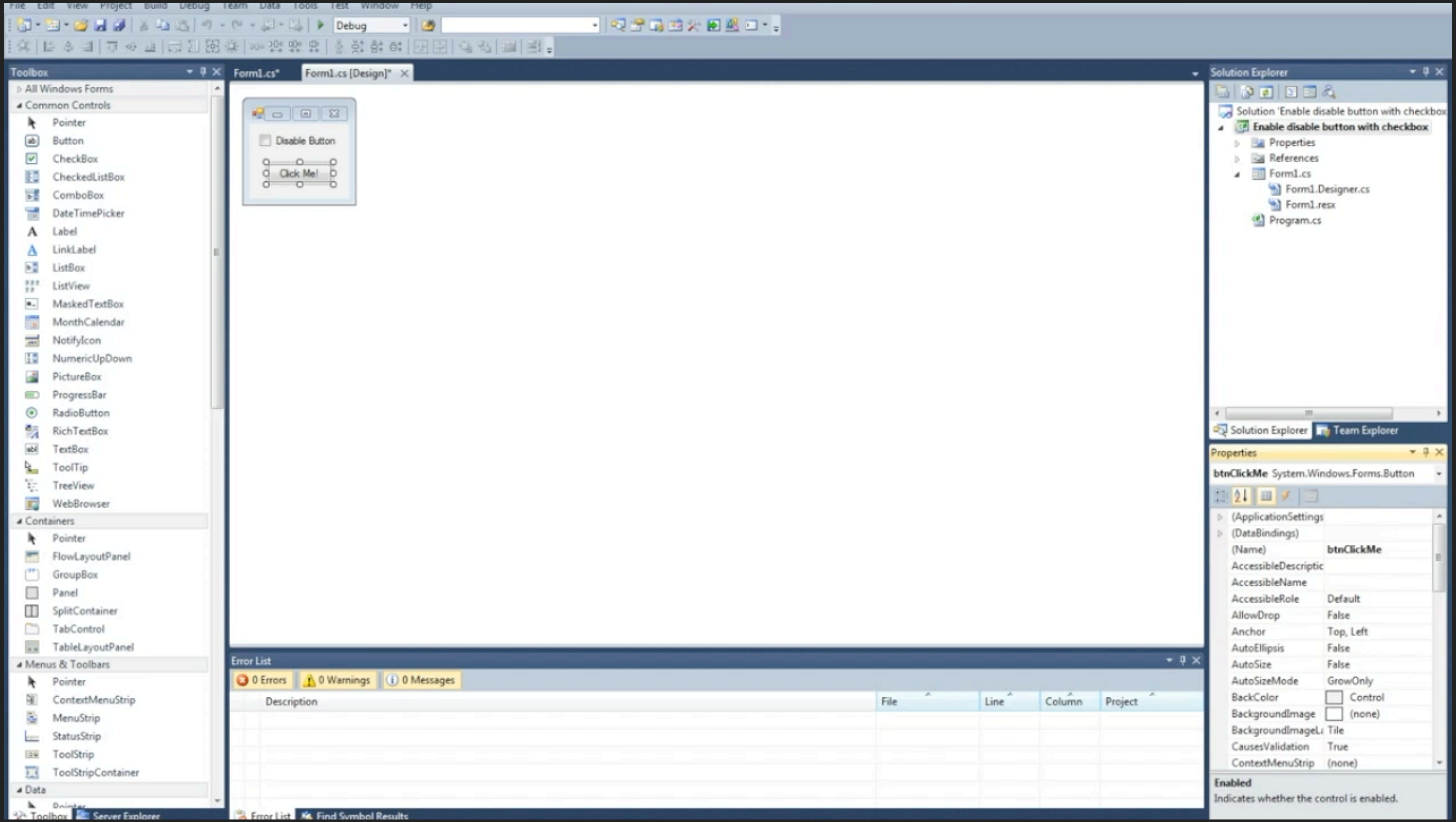
Id_Libro	id_editorial	titulo	copyright	categoria	pre
100	1	Tutorial de SQL ...	01/01/2011	5	543
101	6	SQL Server Inicio	01/01/2009	1	127
102	2	Como cuidar el a...	01/01/2005	6	342
103	4	Manual de progra...	01/01/2001	3	245
104	4	Juegos de Guerra	01/01/2011	6	729
105	6	Pantaleon y las vi...	01/01/1986	2	345
106	5	Manual de Visual...	01/01/1994	2	638

AN IMPERATIVE APPROACH

WINFORMS

BUILDING A UI: AN IMPERATIVE APPROACH

- Model the controls as objects
- UI is built by instantiating and editing properties of those objects
- A **designer** can help significantly
- Most logic goes into the **event hooks**
- Hooks can modify **both data and UI** controls



btnClickMe System.Windows.Forms.Button

(ApplicationSettings)	
(DataBindings)	
(Name)	btnClickMe
AccessibleDescriptic	
AccessibleName	
AccessibleRole	Default
AllowDrop	False
Anchor	Top, Left
AutoEllipsis	False
AutoSize	False
AutoSizeMode	GrowOnly
BackColor	Control
BackgroundImage	(none)
BackgroundImageL	Tile
CausesValidation	True
ContextMenuStrip	(none)

Enabled
Indicates whether the control is enabled.

ADVANTAGES

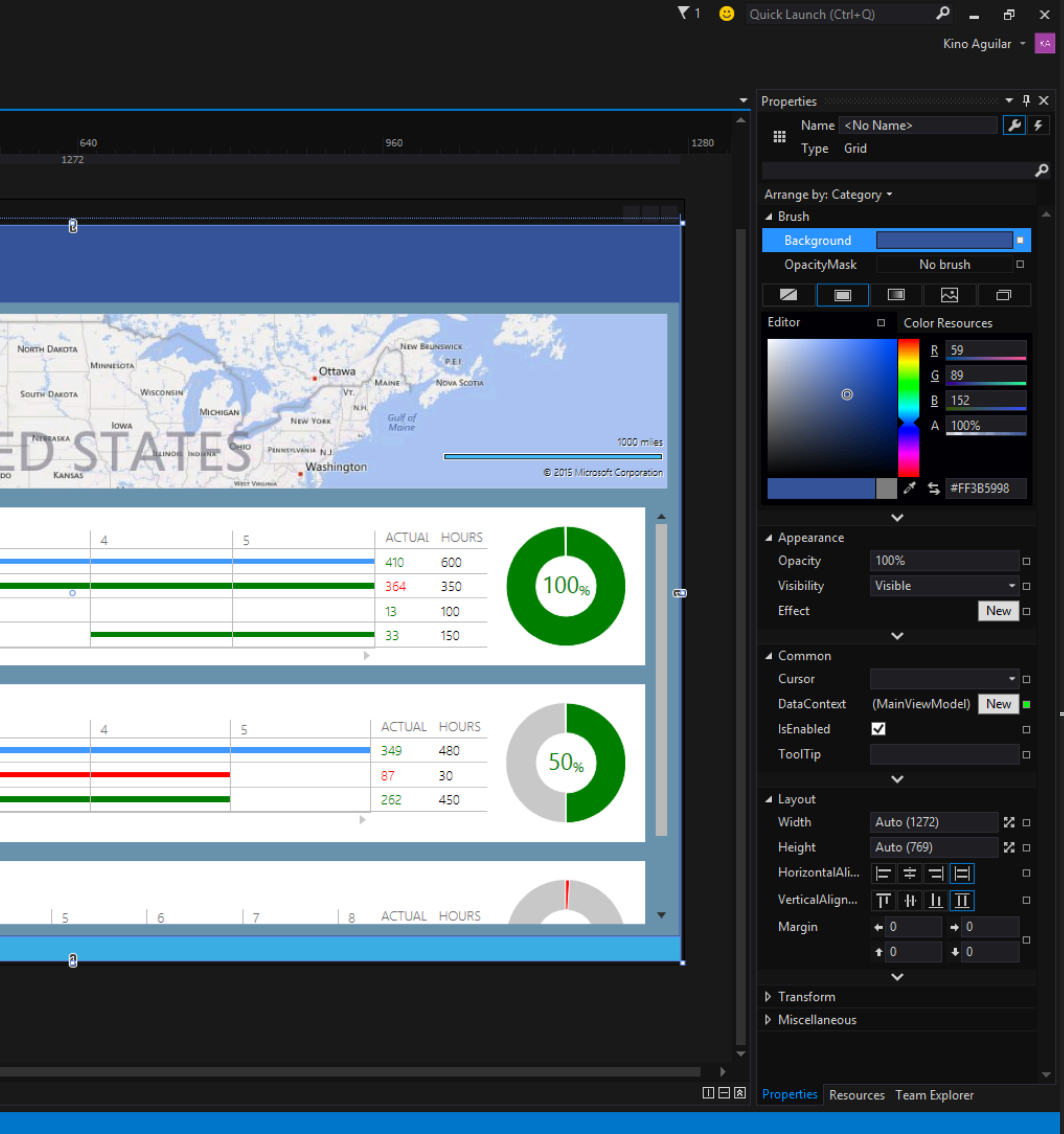
- Rapid prototyping thanks to designer
- Hierarchy of controls and layouts fits well in OOP paradigm
- MVC: Separation of concerns, move logic to controller

PROBLEMS

- Code generated by designer cannot be touched
- UI is not very dynamic
- Difficult to create custom components

IN THE WEB

- DOM: Document Object Model
- jQuery makes it more tractable, still imperative
- Very basic native controls, no styling
- Some designers available



SEPARATE THE VIEW FROM THE LOGIC

WPF

BUILDING A UI: SEPARATING VIEW FROM LOGIC

- WPF: MVVM & XAML
- Cannot fit more acronyms in a shorter space
- XAML: **Declarative language** for the UI
- MVVM: Link the UI and model through “magic” **bindings**
- Lot of logic still happening in the events

ADVANTAGES

- Can use both designer and edit UI code
- Designer and programmer can work separately
- Easier to write components
- Custom styling is easier too

BikeSharing.Xamarin - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Quick Launch (Ctrl+Q)
James Montemagno
Debug
iPhoneSimulator
BikeSharing.Clients.iOS
iPhone 5 iOS 10.1
LoginPage.xaml*
Label
Label
Document Outline
Properties
Test Explorer
Toolbox
Forms Previewer
Device: Phone Tablet
Platform: Android iOS
Username
Password
Stay logged In
YES
NO
Forgot Password?
Sign in
Don't have an account? Sign up
Solution Explorer
Team Explorer
Notifications
100 %
0 changes
0 authors, 0 changes
Error List
Task List...
Output
Find Results 1
Breakpoints
Web Publish Activity
Ready
Ln 80
Col 43
Ch 43
INS
0
41
BikeSharing360
master

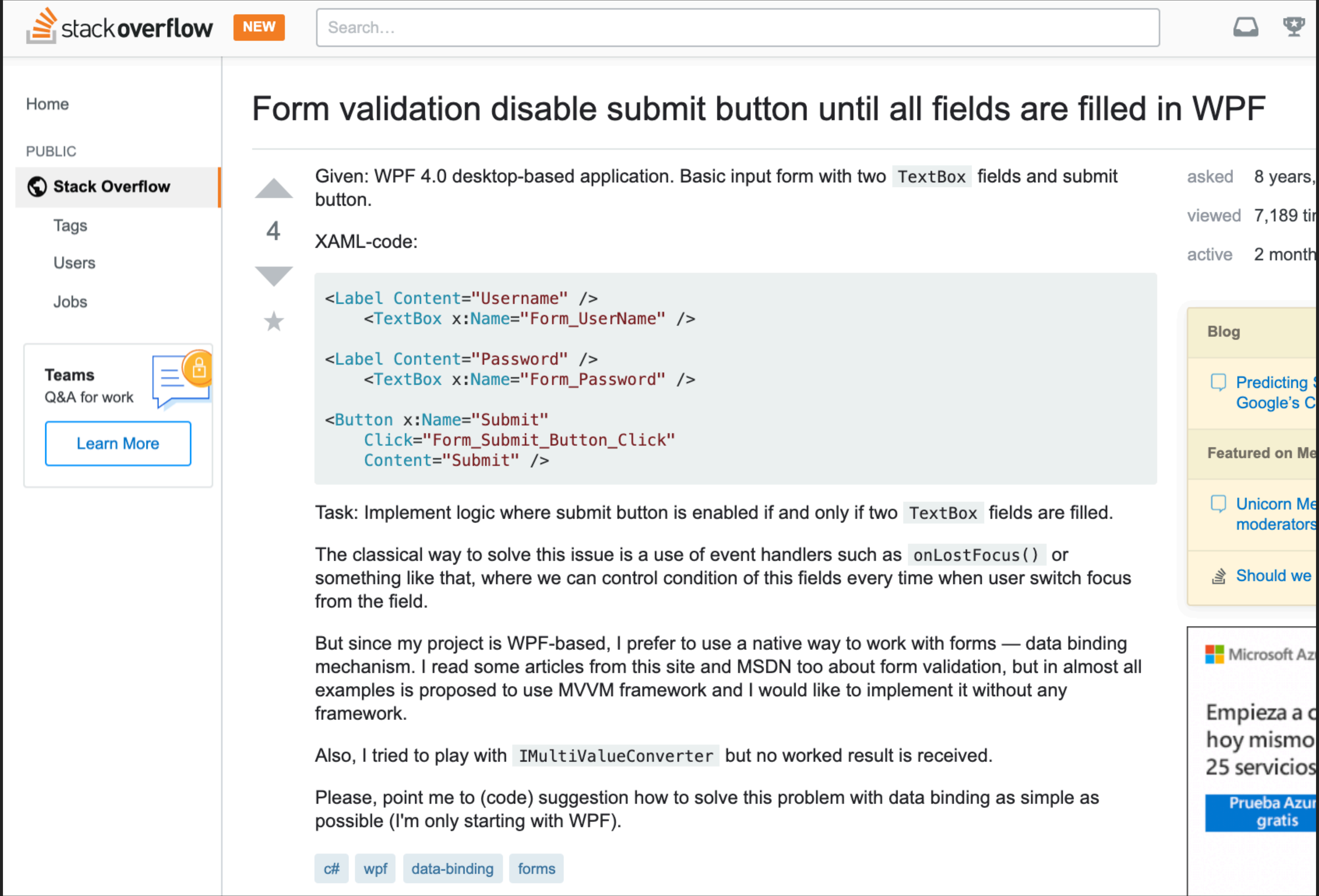
```
        </RowDefinition.Height>
      </RowDefinition>
    </Grid.RowDefinitions>
    <StackLayout Grid.Row="0">
      <Entry x:Name="EntryUserName"
             Text="{Binding UserName.Value, Mode=TwoWay}"
             Placeholder="Username"/>
      <Entry x:Name="EntryPassword"
             Text="{Binding Password.Value, Mode=TwoWay}"
             Placeholder="Password"
             IsPassword="true">
      </Entry>

       <StackLayout Margin="0,20" HorizontalOptions="Center" Orientation="Hor
        <Label Text="Stay logged In" TextColor="White" VerticalOptions="Cente
          <myiOS:SegmentedControl
              SelectedIndex="{Binding Remember, Mode=TwoWay}">
              <x:Arguments>
                  <x:String>YES</x:String>
                  <x:String>NO</x:String>
              </x:Arguments>
          </myiOS:SegmentedControl>
      </StackLayout>

      <Label Text="Forgot Password?"
             TextColor="White"
             FontSize="20"
             HorizontalOptions="End"/>
      <Button Command="{Binding SignInCommand}"
              Text="Sign in"
              BorderColor="White"
              TextColor="White"
              BorderWidth="1"/>
    </StackLayout>
    <StackLayout Grid.Row="1"
                 Orientation="Horizontal"
                 HorizontalOptions="Center">
```

IN THE WEB

- Golden age of data binding libraries: Backbone, Knockout, Ember
- Introduction of template system to make HTML dynamic
- Vue.js is very popular nowadays

PROBLEMS

- Need to learn another language
- Bindings can get complex (one-way, two-way, triggers)
- XAML is limited, some operations require many “tricks” from framework



stackoverflow NEW Search…

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

Teams
Q&A for work

Learn More

Form validation disable submit button until all fields are filled in WPF

4

Given: WPF 4.0 desktop-based application. Basic input form with two `TextBox` fields and submit button.

XAML-code:

```
<Label Content="Username" />
    <TextBox x:Name="Form_UserName" />

<Label Content="Password" />
    <TextBox x:Name="Form_Password" />

<Button x:Name="Submit"
    Click="Form_Submit_Button_Click"
    Content="Submit" />
```

Task: Implement logic where submit button is enabled if and only if two `TextBox` fields are filled.

The classical way to solve this issue is a use of event handlers such as `onLostFocus()` or something like that, where we can control condition of this fields every time when user switch focus from the field.

But since my project is WPF-based, I prefer to use a native way to work with forms — data binding mechanism. I read some articles from this site and MSDN too about form validation, but in almost all examples is proposed to use MVVM framework and I would like to implement it without any framework.

Also, I tried to play with `IMultiValueConverter` but no worked result is received.

Please, point me to (code) suggestion how to solve this problem with data binding as simple as possible (I'm only starting with WPF).

c# wpf data-binding forms

asked	8 years,
viewed	7,189 ti
active	2 month

Blog

Predicting
Google's C

Featured on Me

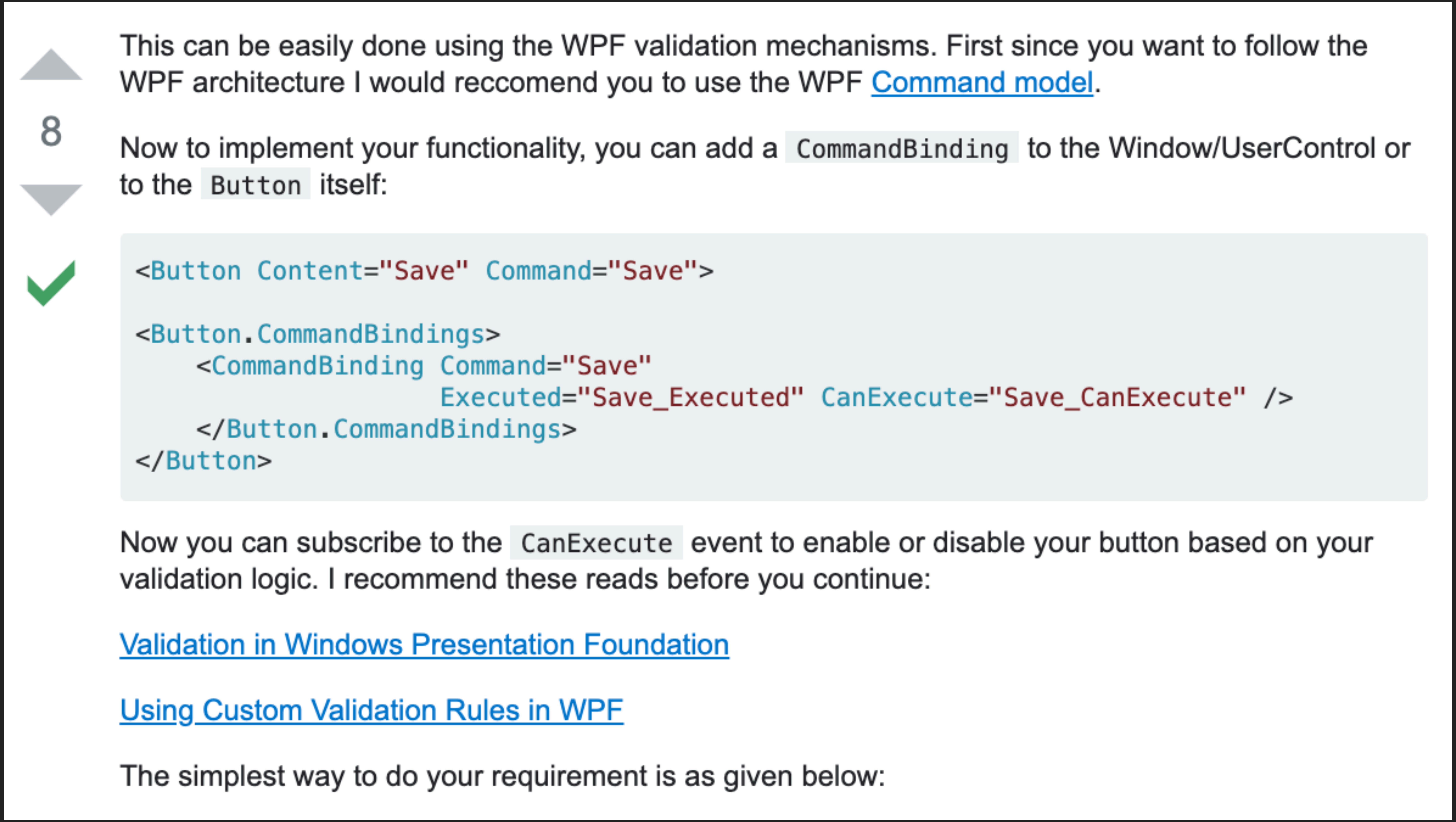
Unicorn Me
moderators

Should we

Microsoft Az

Empieza a c
hoy mismo
25 servicios

Prueba Azur
gratis



8

This can be easily done using the WPF validation mechanisms. First since you want to follow the WPF architecture I would reccomend you to use the WPF Command model.

Now to implement your functionality, you can add a `CommandBinding` to the Window/UserControl or to the `Button` itself:

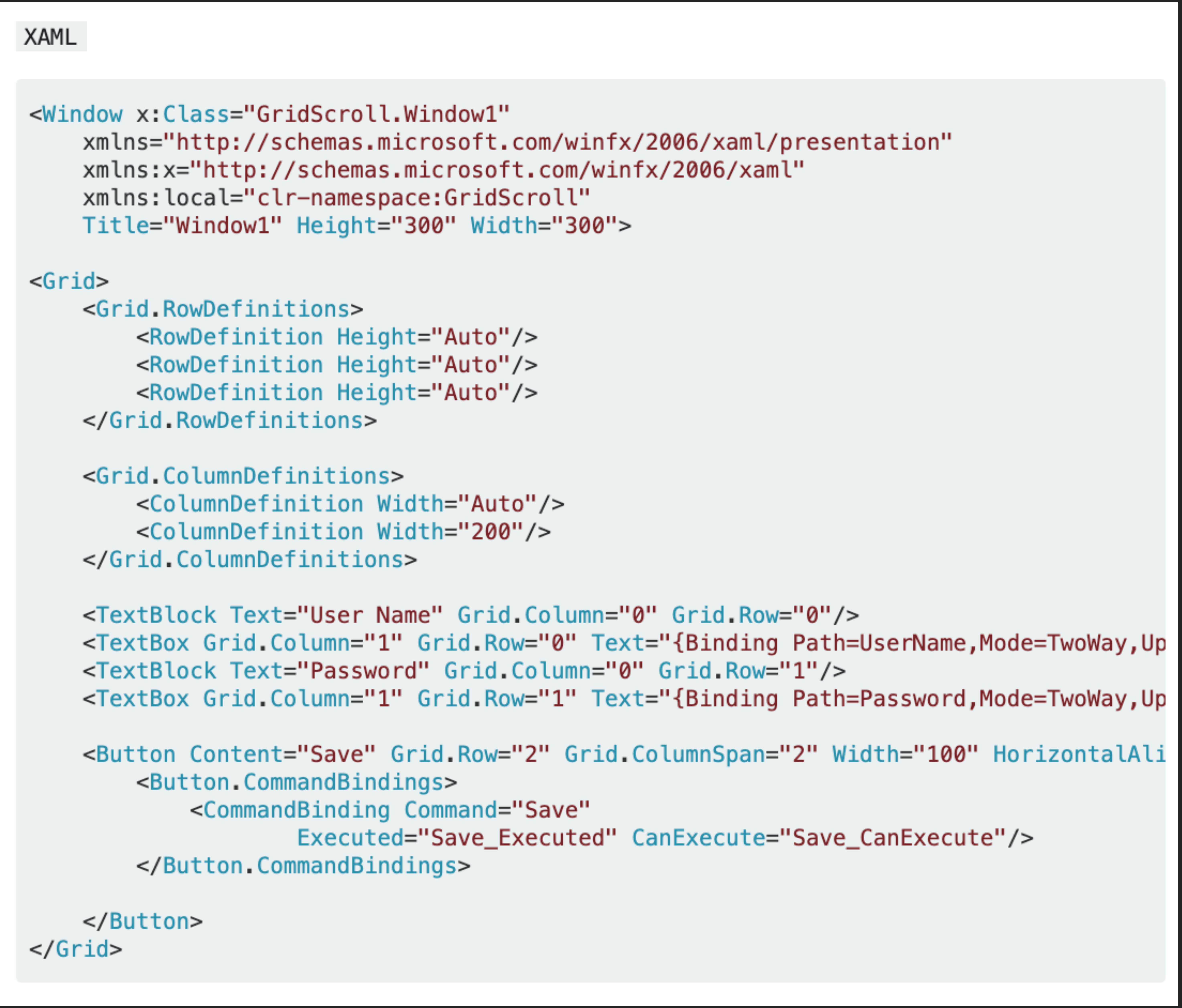
```
<Button Content="Save" Command="Save">

<Button.CommandBindings>
    <CommandBinding Command="Save"
                    Executed="Save_Executed" CanExecute="Save_CanExecute" />
    </Button.CommandBindings>
</Button>
```

Now you can subscribe to the `CanExecute` event to enable or disable your button based on your validation logic. I recommend these reads before you continue:

Validation in Windows Presentation Foundation

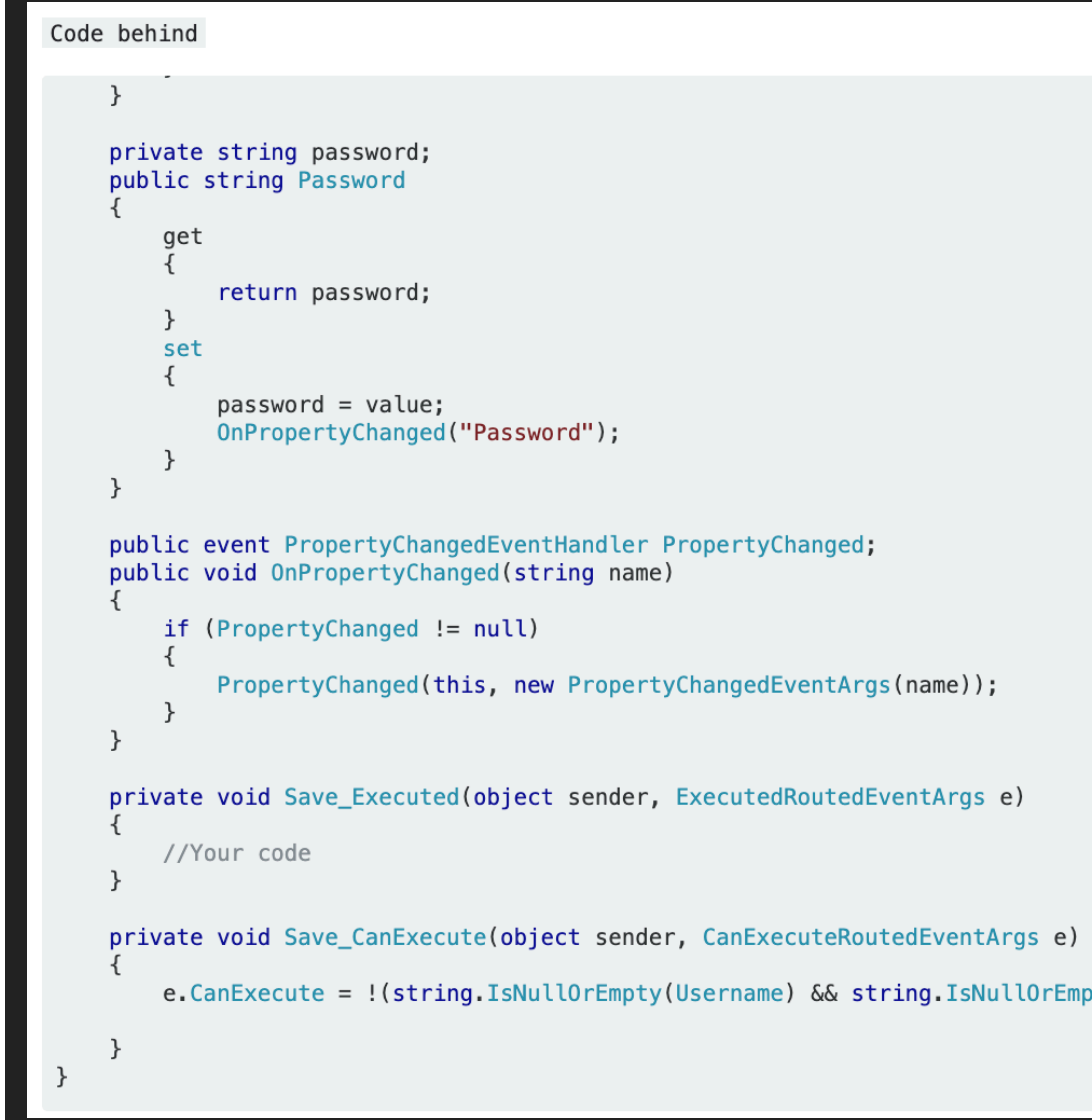
Using Custom Validation Rules in WPF

The simplest way to do your requirement is as given below:



XAML

```
<Window x:Class="GridScroll.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:GridScroll"
    Title="Window1" Height="300" Width="300">

<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="200"/>
    </Grid.ColumnDefinitions>

    <TextBlock Text="User Name" Grid.Column="0" Grid.Row="0"/>
    <TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=UserName,Mode=TwoWay,Up
    <TextBlock Text="Password" Grid.Column="0" Grid.Row="1"/>
    <TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=Password,Mode=TwoWay,Up

    <Button Content="Save" Grid.Row="2" Grid.ColumnSpan="2" Width="100" HorizontalAli
        <Button.CommandBindings>
            <CommandBinding Command="Save"
                    Executed="Save_Executed" CanExecute="Save_CanExecute"/>
        </Button.CommandBindings>

    </Button>
</Grid>
```



Code behind

```
    }

    private string password;
    public string Password
    {
        get
        {
            return password;
        }
        set
        {
            password = value;
            OnPropertyChanged("Password");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    public void OnPropertyChanged(string name)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(name));
        }
    }

    private void Save_Executed(object sender, ExecutedRoutedEventArgs e)
    {
        //Your code
    }

    private void Save_CanExecute(object sender, CanExecuteRoutedEventArgs e)
    {
        e.CanExecute = !(string.IsNullOrEmpty(Username) && string.IsNullOrEmp

    }
}
```

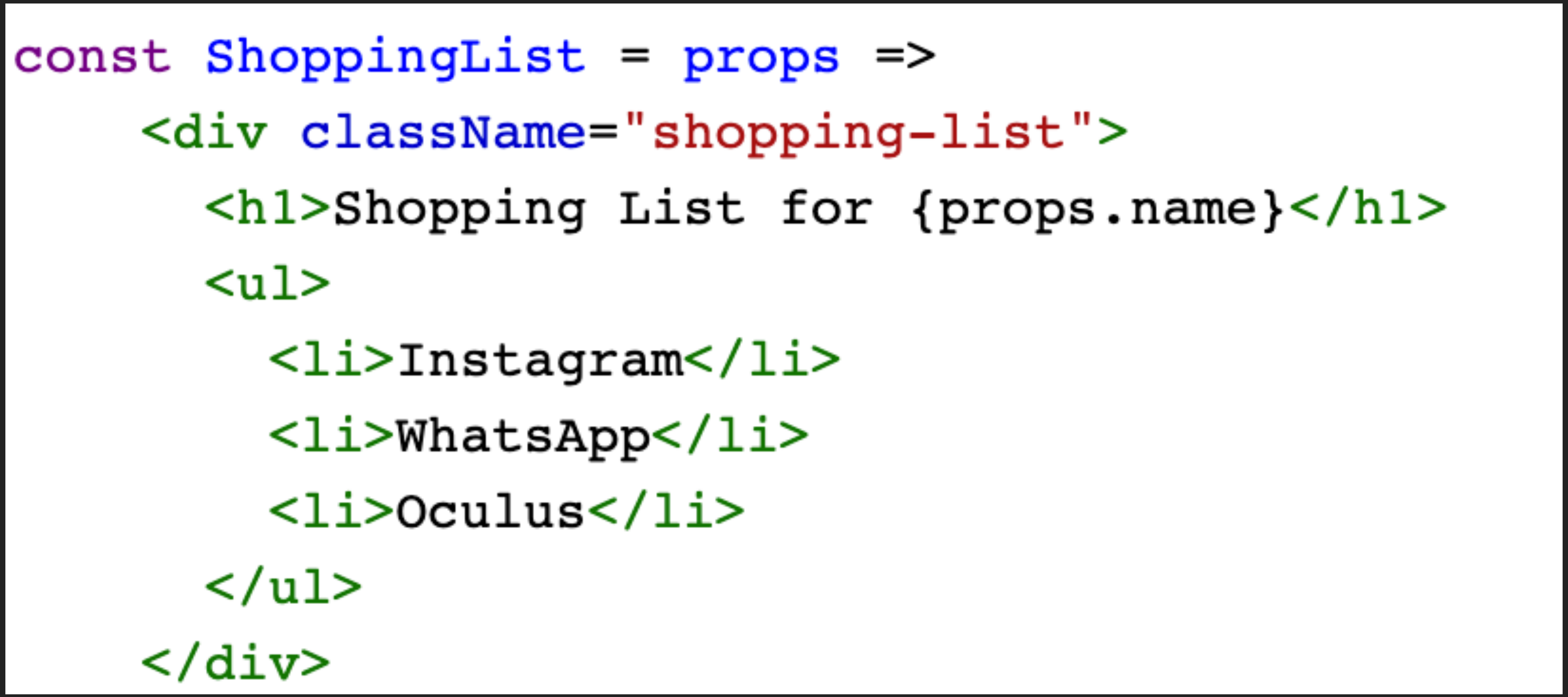
IT WILL FIX ALL THE THINGS!

FUNCTIONAL PROGRAMMING

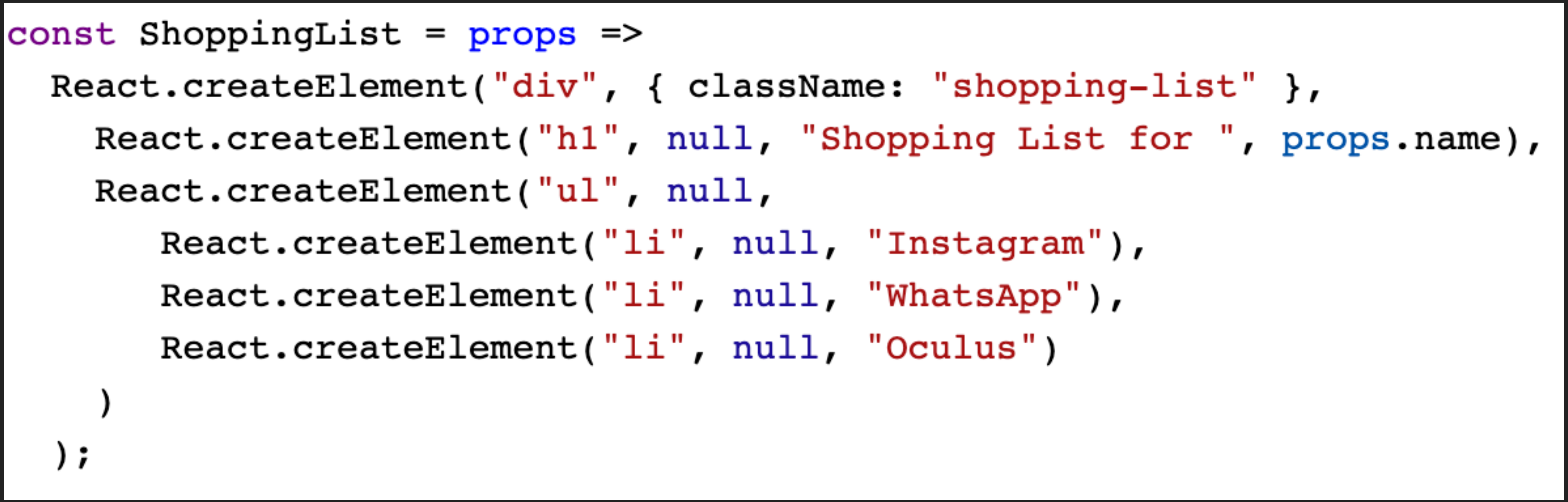
MEANWHILE IN THE WEB…

REACT

- Virtual DOM
- Write UIs **declaratively** using same programming language
- Everything is a **component**
- Components can be **functions**, that are easily composed
- Encourages **immutability** and **one-way** data flow



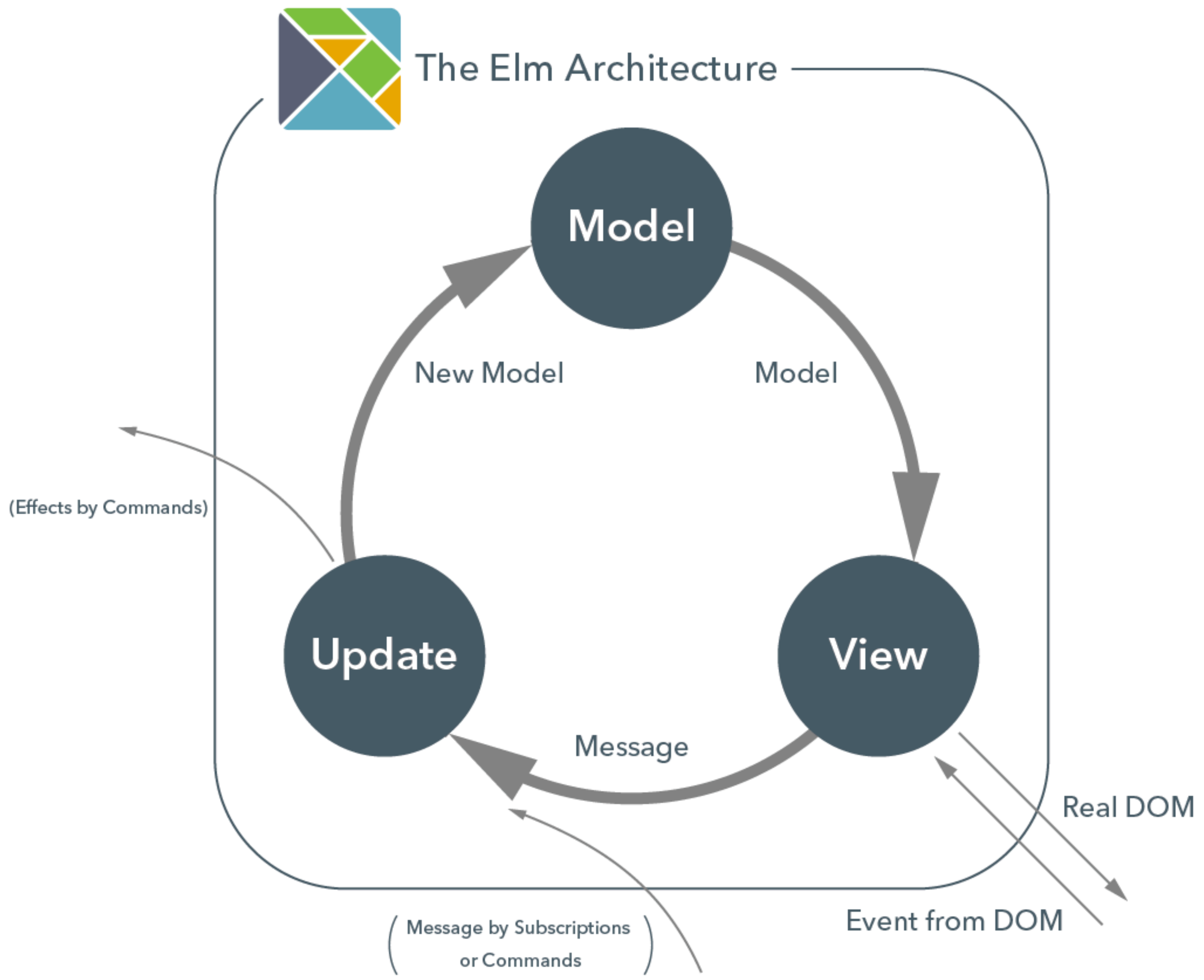
```
const ShoppingList = props =>
    <div className="shopping-list">
      <h1>Shopping List for {props.name}</h1>
      <ul>
        <li>Instagram</li>
        <li>WhatsApp</li>
        <li>Oculus</li>
      </ul>
    </div>
```

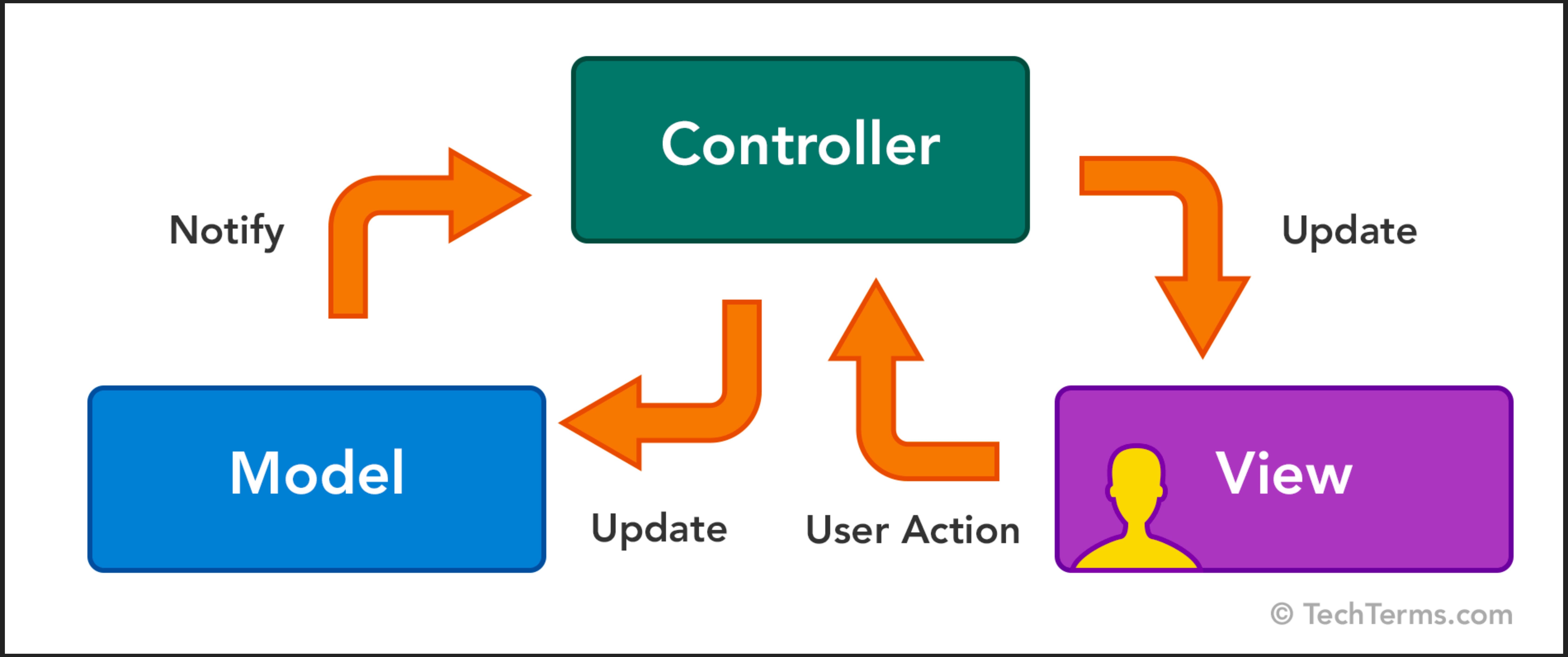


```
const ShoppingList = props =>
  React.createElement("div", { className: "shopping-list" },
    React.createElement("h1", null, "Shopping List for ", props.name),
    React.createElement("ul", null,
       React.createElement("li", null, "Instagram"),
       React.createElement("li", null, "WhatsApp"),
       React.createElement("li", null, "Oculus")
    )
  );
```

ELM (ARCHITECTURE)

- Model-View-Update
- **Model**: Immutable data structure that defines the UI at a specific point
- **View**: Pure function that transforms the model into UI elements
- **Update**: Receives the current state of the model and a **message**, and returns a new model


The Elm Architecture
Model
New Model
Model
(Effects by Commands)
Update
View
Message
Real DOM
Event from DOM
Message by Subscriptions or Commands


Controller
Notify
Update
Model
Update
User Action
View
© TechTerms.com

FABLE

- F# to JS compiler: fable.io/repl
- **Fable.Elmish**: implementation of Elm architecture for Fable
- Uses React as render engine
- Inspired other projects like Fabulous (Elmish for Xamarin)
- C# tends to follow Redux (variant of Elm arch.)

MESSAGES

- View function receives the state and a **dispatch** function
- **Events** dispatch messages when triggered
- Most logic is removed from the events
- Messages improve **semantics** over raw events
- Implementing library must include a queue to deal with messages **sequantially**

```fsharp
type Msg =
  | Increment
  | Decrement

Msg -> Model -> Model
let update (msg:Msg) (model:Model) =
    match msg with
    | Increment -> { model with Value = model.Value + 1 }
    | Decrement -> { model with Value = model.Value - 1 }

Model -> (Msg -> unit) -> ReactElement
let view (model:Model) dispatch =
  div [] [
    button [ OnClick (fun _ -> dispatch Increment) ] [ str "+" ]
    div [] [ str (string model.Value) ]
    button [ OnClick (fun _ -> dispatch Decrement) ] [ str "-" ]
  ]
```

COMMANDS (ASYNCHRONOUS ACTIONS)

- Update function is **synchronous**
- Updates must be fast to prevent locking the UI
- **Asynchronous** actions (like REST calls) can be run inside commands
- Commands are just callbacks that receive the dispatch function as argument
- When the callback is finished, it dispatches a message **triggering another update/render cycle**

```fsharp
open Thoth.Json

let private getRandomUser () = promise {
    let! response = Fetch.fetch "https://randomuser.me/api/" []
    let! responseText = response.text()
    let resultDecoder = Decode.field "results" (Decode.index 0 User.Decoder)
    return Decode.fromString resultDecoder responseText
}
```

```
let update (msg:Msg) (model:Model): Model * Cmd<Msg> =
    match msg with
    | FetchRandomUser ->
        let newModel =
            match model with
            | Loaded user -> Loading (Some user)
            | _ -> Loading None
        newModel, Cmd.OfPromise.either getRandomUser () FetchResponse FetchError

    | FetchResponse parsedJson ->
        match parsedJson with
        | Ok user -> Loaded user, Cmd.none
        | Error _ -> Errored, Cmd.none

    | FetchError error ->
        Errored, Cmd.none
```

COMPONENTS

- Components as a **pattern**: code for Model-View-Update
- Usual file structure: Types/State/View
- Components organize themselves **hierarchically** through **composition**
- Messages bubble up, view and updates flow top-down
- Children can communicate with parent by **external messages**
- The app is just the root component

```
f1 >> f2
```

```
fun x → f2(f1(x))
```

```fsharp
type Msg =
  | Increment
  | Decrement
  | DeltaMsg of Delta.Msg

Msg -> Model -> Model
let update (msg:Msg) (model:Model) =
    match msg with
    | Increment -> { model with Value = model.Value + model.Delta }
    | Decrement -> { model with Value = model.Value - model.Delta }
    | DeltaMsg msg ->
      { model with Delta = Delta.update msg model.Delta }

Model -> (Msg -> unit) -> ReactElement
let view (model:Model) dispatch =
  div [] [
    button [ OnClick (fun _ -> dispatch Increment) ] [ str "+" ]
    div [] [ str (string model.Value) ]
    button [ OnClick (fun _ -> dispatch Decrement) ] [ str "-" ]
    Delta.view model.Delta (DeltaMsg >> dispatch)
  ]
```

LET'S SEE IT IN ACTION

ADVANTAGES

- Removes a lot of cognitive overhead
- Single language for logic and view
- Immutability and message queue make it much easier to reason about model
- Enables hot reloading and time travel debugging
- Easy to maintain thanks to “repetitive” structure

PROBLEMS

- Sometimes doesn’t feel “smart enough”
- Almost impossible to have a designer
- Some boilerplate to wire components and add actions
- Needs some care to avoid unnecessary renders (memoize components)

THANK YOU!

@alfonsogcnunez

@fablecompiler

fable.io/fableconf