# Creazione di una Applicazione Test Server e Client REST

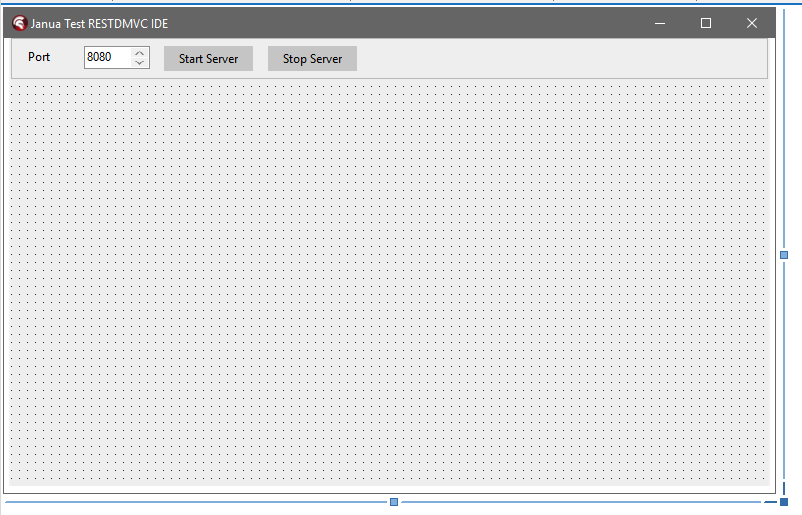
# Container e Server DMVC Interno per i Test

L’idea è di un Form Container (sia VCL che FMX) che possa contenere le varie applicazioni da Testare.

Per il Test di costruzione di questo esempio ho usato

Il form ‘Contenitore’ nasce da una necessità, quella di Mantenere la struttura dell’applicazione da Collaudare e di tutti i suoi Test ma anche poter eseguire ed effettuare il debug dell’application Server REST o di un WebServer all’interno dell’Applicazione (cosa leggermente più complessa).

Quindi occorre Creare una Standard VCL Application come Test Container di tutto il progetto. Che quindi si occupi prima di tutto di avviare il Server REST DMVC Successivamente Testare e verificare che login sia attivo usando un utente di Test.



**type**

TfrmJanuaVCLTestRestDMVCID = **class** (TForm)

pnlMain: TPanel;

**JanuaframeWebServerManager1: TJanuaframeWebServerManager;**

**private**

{ Private declarations }

**public**

{ Public declarations }

**end**;

L’elemento principale è il TJanuaframeWebServerManager che si occupa di fungere da interfaccia tra la Vista (il Form) e il ‘motore’ del DMVC. Si trova come Componente non come ‘frame’ quindi è facile ‘beccarlo’. NOTA l’ho impostato come componente quindi si trova nella Palette.

Per far ‘funzionare’ tutta la baracca Occorrono diverse Uni che vanno riportate in tutti i progetti FMX o VCL in questo caso il nostro esempio è per VCL vedremo di farne uno chiaramente anche per FMX.

Janua.TMS.Error, Janua.Application.Framework, Janua.Uni.Framework, Janua.Vcl.Framework, Janua.ViewModels.Framework, Janua.Orm.Register, Janua.Vcl.MVVM.Framework,

|  |
| --- |
| **uses**  Vcl.Forms,  Janua.TMS.Error,  Janua.Application.Framework,  Janua.Uni.Framework,  Janua.Vcl.Framework,  Janua.ViewModels.Framework,  Janua.Orm.Register,  Janua.Vcl.MVVM.Framework,  ufrmJanuaAllDemosContainer **in** '..\..\..\Samples\Janua\VCL\ufrmJanuaAllDemosContainer.pas' {frmAllDemosContainer};  {$R \*.res}  **var**  errorManager: TJanuaTmsExceptionHandler;    **begin**  TJanuaApplication.AppName := ('desktop.januaproject.it');  Application.Initialize;  errorManager := TJanuaTmsExceptionHandler.Create(Application);  errorManager.Activate;  // \*\*\*\* Here goes the custom configuration for the Application    Application.MainFormOnTaskbar := True;  Application.CreateForm(TfrmAllDemosContainer, frmAllDemosContainer);  Application.Run;  **end**. |

Da notare la configurazione dell’Error Manager ‘custom’ che io uso spesso nelle applicazioni VCL nelle righe, dalla Unit Janua.TMS.Error. Unit che si potrebbe ‘replicare’ per FMX (Tablet/Destk)

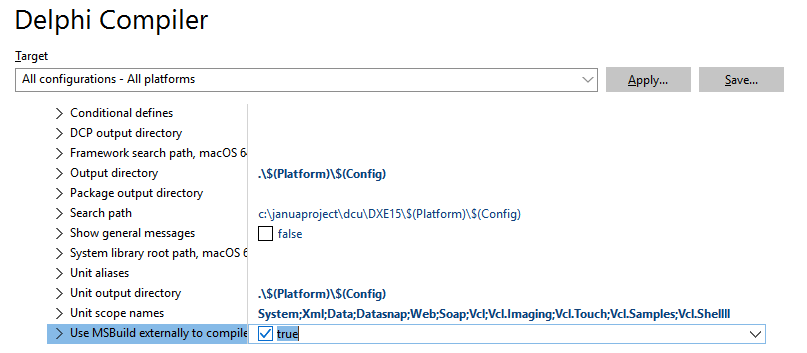
**var**

errorManager: TJanuaTmsExceptionHandler;

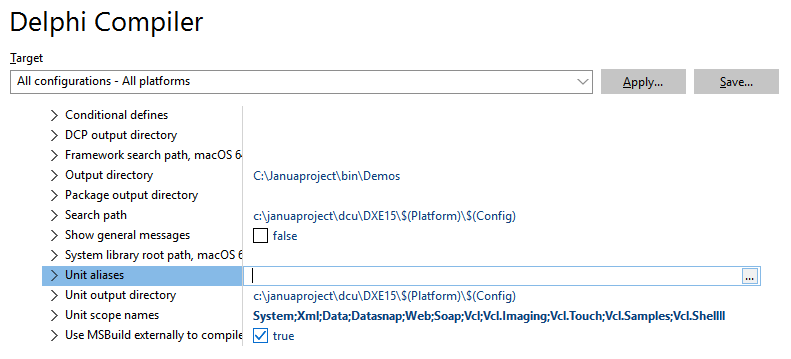
…

errorManager := TJanuaTmsExceptionHandler.Create(Application);

errorManager.Activate;

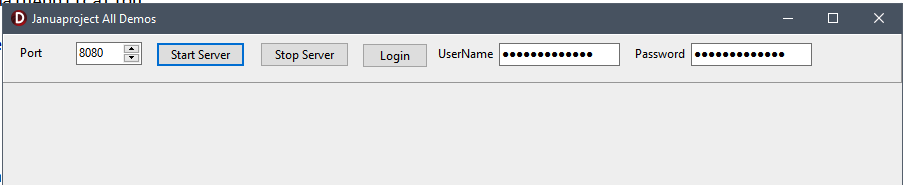


Usando la Build Configurations occorre cancellare i Default di Unit Output e Output Directory



A questo punto si può almeno testare se compila e si ‘avvia’ (NOTA: spesso non ho configurato gli ambienti a 32 bit perché i miei ambienti target sono da diversi anni solo a 64bit.

Il nostro programma così si avvia (in questo momento è presente giusto l’interfaccia e nessun ‘background’) ho aggiunto un pulsante Login e due controlli UserName e Password. Il primo ‘step’ per ‘avviare’ l’applicazione sarebbe Testare una login con DMVC dopo avere ovviamente fatto partire il server. t



A questo punto occorre preparare l’ambiente Applicazione per utilizzare 2 Framework (ambienti di lavoro) differenti, un ambiente Client/Server che servirà come ‘base’ per DMVC cioè il server REST ed uno invece VCL ‘non’ Client-Server cioè solo con un Client REST.

|  |
| --- |
| **unit** uJanuaRestDMVCApplication;  **interface**  **uses** System.Classes, System.SysUtils, Janua.Core.Types, Janua.Bindings.Impl, Janua.VCL.Application;  **type**  TJanuaRestDMVCApplication = **class**(TJanuaVCLCustomApplication)  **public**  **class** procedure RegisterForms; override;  **class** procedure ApplicationSetup(**const** aAppname: string); override;  **class** procedure ApplicationTearDown; override;  **class** procedure ConnectionSetup; override;  **class** procedure LoadMenu; override;  **end**; |

Per il momento lasciamo tutto ‘di Default’ ed andiamo a richiamare il nostro codice di Setup nel Project inserendolo giusto dopo l’attivazione dell’Error Manager.

|  |
| --- |
| // \*\*\*\* Here goes the custom configuration for the Application  TJanuaRestDMVCApplication.ApplicationSetup('desktop.januaproject.it'); |

## Test del Web-REST Server DMVC

Per Testare il nostro REST Server occorrerò Prima di tutto ‘registrare’ sul Frame un web Server da Avviare. Il Frame ha una proprietà per avviare un Web Server che sia un discendente della Classe TJanuaWebServer che si basa su alcuni metodi Virtuali Astratti.

Unico punto avevo usato una class property per la porta ma siccome il WebServer viene creato ed avviato una volta si potrebbe impiegare una proprietà non di Classe magari per testare più server WebBroker o altri server contemporaneamente sulla piattaforma.

|  |
| --- |
| **unit** Janua.Http.WebServer;  ...  **Type**  TJanuaWebServer = **class**  **public**  **constructor** Create; overload;  /// <summary> Imposta la class Var FPort. Variabile 'unica' in tutta l'applicazione </summary>  **class** **Constructor** CreateClass;  **private**  **class** **var** FPort: Integer;  **private**  FLogProc: TMessageLogProc;  **public**  **class** function GetPort (aDefault: Integer): Integer; overload;  procedure StartServer; overload; **virtual**; **abstract**;  procedure StopServer; overload; **virtual**; **abstract**;  procedure WriteStatus; **virtual**; **abstract**;  **public**  **property** IsActive: Boolean read GetIsActive write SetIsActive;  **property** LogProc: TMessageLogProc read FLogProc write SetLogProc;  **class** **property** Port: Integer read GetPort write SetPort;  **end**; |

Per prima cosa il ‘motore’ in uso va Registrato sulla Janua Application Framework Factory

|  |
| --- |
| TJanuaWebServerClass = **class** **of** TJanuaWebServer;  TJanuaWebServerFactory = **class**  **private**  **class** **var** FWebServerClass: TJanuaWebServerClass;  **public**  **class** function CreateWebServer: TJanuaWebServer;  **public**  **class** **property** WebServerClass: TJanuaWebServerClass read FWebServerClass **write** FWebServerClass;  **end**; |

A livello di Classe WebBroker implementa i metodi di WebServer e aggiunge TIdHTTPWebBrokerBridge

|  |
| --- |
| TJanuaWebBrokerServer = **class**(TJanuaWebServer)  **private**  FServer: TIdHTTPWebBrokerBridge;  ...  **class** **property** WebModuleClass: TComponentClass read FWebModuleClass write FWebModuleClass;  **end**; |

Si potrebbero Testare due Server WebBroker differenti in ascolto su due porte in questo caso sia la property WebModuleClass che la property Port dovrebbero essere proprietà non di Classe.

Per semplificare l’avvio di un Server WebBroker da un’Applicazione VCL ho creato un Frame (registrato come componente) che possa ‘lanciare’ un Web Server e gestirne la porta di ascolto:

|  |
| --- |
| **type**  TJanuaframeWebServerManager = **class**(TFrame)  sedPort: TscSpinEdit;  …  **private**  FWebModuleClass: TComponentClass;  …  **protected**  procedure StartServer; **virtual**;  procedure StopServer; **virtual**;  **public**  { Public declarations }  **property** WebModuleClass: TComponentClass **read** FWebModuleClass **write** FWebModuleClass;  **property** WebBrokerClass: TJanuaWebBrokerServerClass **…;**  **property** Url: string **read** FUrl **write** SetUrl;  **published**  **property** OnCreate: TNotifyEvent **read** FOnCreate **write** SetOnCreate;  **property** OnAfterStartServer: TNotifyEvent **…;**  **property** OnBeforeStartServer: TNotifyEvent **…;**  **end**; |
| **procedure** **TJanuaframeWebServerManager.StartServer;**  **begin**  if Assigned(OnBeforeStartServer) **then** OnBeforeStartServer(Self);  **if** Assigned(FWebBrokerClass) **then**  **begin**  FWebBrokerClass.WebModuleClass := FWebModuleClass;  TJanuaWebServerFactory.WebServerClass := FWebBrokerClass;  **end**;  FUrl := Format('http://localhost:%d', [sedPort.ValueAsInt]);  FWebServer := TJanuaWebServerFactory.CreateWebServer as TJanuaWebBrokerServer;  FWebServer.StartServer;  **if** Assigned(OnAfterStartServer) **then** OnAfterStartServer(**Self**);  **end**; |

Il ‘Frame’ è in grado quindi di gestire la Creazione e la Configurazione di un Server (in questo caso ‘compatibile’ con WebBroker).

Cosa ‘regola’ il nostro Frame? L’evento **OnBeforeStartServer** e la proprietà **WebModuleClass.**

|  |
| --- |
| procedure TfrmVCLWebBrokerDMVCTest.frameDMVCWebBrokerBeforeStartServer (Sender: TObject);  // prima dell'avvio del Web Server (ma dopo la sua creazione) possiamo  // indicargli quale sia la classe del webModule che la sua Factory  // si occuperà di creare ad ogni avvio del Server Stesso  **begin**  // nota JanuaWebBrokerServerClass è una Variabile di tipo TJanuaWebBrokerServerClass  frameDMVCWebBroker.WebBrokerClass := JanuaDMVCWebBrokerServerClass;  frameDMVCWebBroker.WebModuleClass := JanuaDMVCTestWebModuleClass;  **end**; |

### Come implementare il Form di Test DMVC

**implementation**

**uses** Janua.WebBroker.Server, Janua.Test.WebBroker.WebModule, Janua.DMVC.WebBroker.Server,

Janua.DMVC.Test.CoreWebModule, Janua.Core.Functions, Janua.Application.Framework;

Per prima cose in Uses aggiungere le Unit ‘Standard’ per la configurazione del server riportate qui sopra. E quindi implementare l’evento OnBeforeStartServer come nel riquadro e testare il server DMVC.

Importando nel progetto solo il WebModule di Test sui cui andremo a lavorare per aggiungere Classi e metodi di DMVC

Janua.DMVC.Test.CoreWebModule **in** '..\..\..\src\januacore\dmvc\Test\Janua.DMVC.Test.CoreWebModule.pas' {JanuaDMVCTestWebModule: TWebModule};

Janua.Test.DMVC.www **in** '..\..\..\src\januacore\dmvc\Test\Janua.Test.DMVC.www.pas';

La unit Janua.Test.DMVC.www come si evince dal nome risponde su un responder Model View Controller con una risposta in html (World Wide Web) che avevo implementato per verificare l’interpretazione di alcuni parametri passati alla pagina Web. Analisi Utile per le landing Page e per i siti Web più o meno ‘dinamici’ ad esempio basati su htmx, tecnologia che andremo ad approfondire in dettaglio successivamente.

Tecnicamente abbiamo quindi Creato un’applicazione che permetterebbe di testare un Server REST e, come vedremo, anche un Client REST con poche linee di codice. Sarebbe simpatico avere una App di Test in FireMonkey Sia Desktop/Tablet che Cellulare. NOTA: Su Windows, Linux e Android credo si possa fare un test con una Applicazione Server; su Android non ho mai fatto un simile Test e per quanto riguarda Mac/iOS il test va ‘separato’ in due Con magari un delphi su una VM che collauda il Server ed uno FMX/Client che testa il Client. Ovviamente una App FMX Windows grazie alla portabilità di FMX ci permette di effettuare molti Test.

## Configurazione connessione al DB per i test Locali

Nel nostro esempio ‘forzeremo’ la configurazione della connessione al DB per i test mentre per i servizi DMVC o le applicazioni C/S questa connessione è letta da un file di configurazione.

|  |
| --- |
| procedure TfrmAllDemosContainer.FormCreate(Sender: TObject);  **begin**  // Setup Basic Database Connection for Testing  TJanuaApplication.ServerUserName := 'ergo';  TJanuaApplication.ServerSchema := 'ergomercator';  TJanuaApplication.DBSchemaID := 36;  TJanuaApplication.ServerPassword := '3rg0m3rc4t0r';  TJanuaApplication.ServerPort := 5432;  TJanuaApplication.ServerAddress := 'pg.januaservers.com';  **end**; |

## Configurazione Server REST

All’evento on Create del nostro Form di collaudo dovremo aggiungere la parte di Configurzione per il nostro Server REST (posto che abbiamo già impostato la connessione al DB).

|  |
| --- |
| procedure TfrmAllDemosContainer.FormCreate(Sender: TObject);  **begin**  …  // Rest Server Conf  IsMultiThread := True;  // Januaproject Settings  TJanuaApplication.TestMode := False; // Set this or unset this to enable Test Database connection  TJanuaApplication.RestServerConf.Port := 9010; // port of the service listener can be overwrittn by file conf  **end**; |

In questo caso la porta di Ascolto è la 9010 e la connessine è al DB di produzione ma sullo schema di Test on line.

## Configurazione Client REST

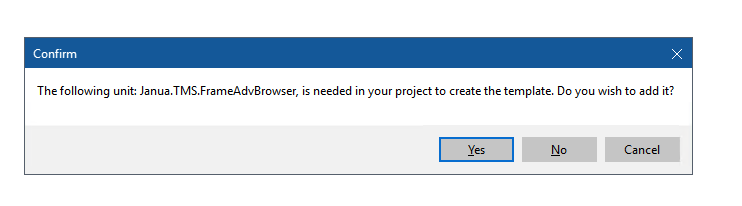
Per configurare il Client REST ed anche verificare che il server si sia avviato Regolarmente possiamo sfruttare l’evento AfterStartServer del frame di Avvio del Server.

Per verificare che l’avvio del Server sia andato a Buon Fine dovremo testare l’Url base del Server e navigare sulla sua pagina di test (che per ora coincide con la ‘home’) . Per farlo useremo il Frame

TframeTmsAdvBrowser1: TframeTmsAdvBrowser;

Che solitamente è disponibile in ‘palette’

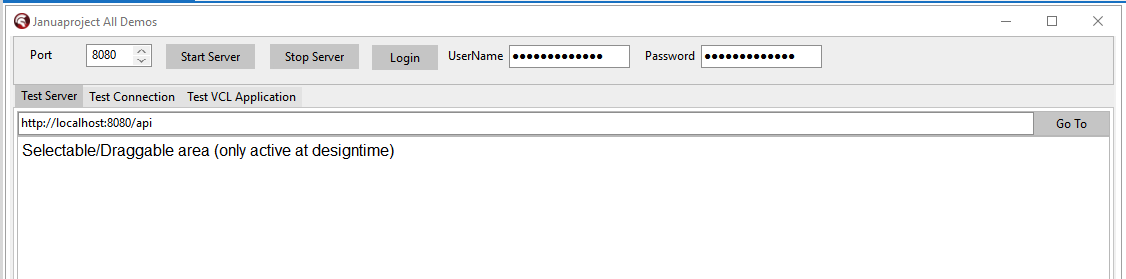




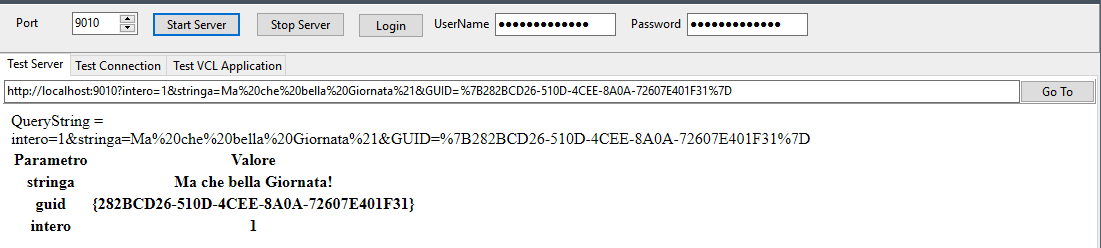
In ogni caso è possibile aggiungere il file ‘a mano’

Janua.TMS.FrameAdvBrowser in '..\..\..\src\TMS\Janua.TMS.FrameAdvBrowser.pas' {frameTmsAdvBrowser: TFrame};

Per farlo aggiungiamo un page Control alla finestra Principale con il WebBrowser



Il test in se è decisamente Banale al Server vengono passati dei normali parametri tramite una ‘GET’ (normalmente DMVC usa l’url per il passaggio dei parametri non delle liste di parametri esplicite). Il Codice può essere aggiunto all’evento



Ed ecco il codice che configura i dati del Client REST (server, localhost e porta, quella configurata all’avvio del Server).

|  |
| --- |
| **procedure** TfrmAllDemosContainer.frameDMVCWebBrokerAfterStartServer(Sender: TObject);  **begin**  // Remote Configuration {'https://api.pikapp.it/'}  TJanuaApplication.RestServer := 'http://localhost';  TJanuaApplication.RESTClientConf.Server := 'http://localhost';  TJanuaApplication.RESTClientConf.Port := frameDMVCWebBroker.sedPort.ValueAsInt;  // usa la porta di Default 443 per https in questo caso    **var**  int := 'intero=1';  **var**  stringa := '&stringa=' + TURI.URLEncode('Ma che bella Giornata!');  v**ar**  guid := '&GUID=' + TURI.URLEncode('{282BCD26-510D-4CEE-8A0A-72607E401F31}');    TframeTmsAdvBrowser1.edtUrl.Text := frameDMVCWebBroker.Url + '?' + int + stringa + guid;  TframeTmsAdvBrowser1.AdvWebBrowser1.Navigate(TframeTmsAdvBrowser1.edtUrl.Text)  **end**; |

### Testiamo ora la Login

Dovremo inserire nome Utente e password e quindi con il Tasto di Login effettuare una Login impiegando il framework REST e recuperare i dati in formato Json dell’utente che fa la login.

Testare la login implica anche impostare e testare il ViewModel di una Sessione Utente. Quindi al nostro Form di Test dovremo aggiungere i ‘pezzi’ e le Unit per effettuare una Login. In un secondo tempo realizzeremo un REST Login Model più adatto a gestire la login secondo lo standard Model-View-ViewModel

|  |
| --- |
| **strict** **private**  FViewModel: IJanuaSystemUserSessionViewModel;  FClientLogin: IRESTLoginClient;  FPassword: string;  FUserName: string;  FLastMessage: string;  **private**  procedure StorePassword;  function InternalLogin: boolean; overload;  function InternalLogin(**const** aUsername, aPassword: string): boolean; overload;  **protected**  function GetViewModel: IJanuaSystemUserSessionViewModel;  procedure SetViewModel(**const** **Value**: IJanuaSystemUserSessionViewModel);  **public**  **property** ViewModel: IJanuaSystemUserSessionViewModel read GetViewModel write SetViewModel; |

Questo codice ‘semplificato’ potremmo inserirlo in un Componente derivato da TJanuaComponent o Creare una semplice TJanuaLoginClass riutilizzabile in altri Form REST non di Test come questo.