LOKAD
forecasting web services

Inject Business Intelligence into .NET apps

Lokad Forecasting Web Services
Partner Integration Case Study

Rinat Abdullin (rinat.abdullin@lokad.com)
Joannes Vermorel (joannes.vermorel@lokad.com)

Last updated: April 2010

Intelligent software brings and saves money. This document illustrates how predictive business intelligence can be injected into a simple .NET application. The intended audience is .NET developers, and in particular those who happen to work on enterprise apps.

We'll see, how easy it is to **model a simple application** using eXpress Application Framework. We will focus on enriching its analytical capabilities with some serious business intelligence by integrating with Lokad Forecasting API in order to get predictive analytics for the published reports.

It is important to note that this work has been done single handedly by one of us (Rinat Abdullin) in less than 2 days of development time.

Basically, Business Intelligence (or BI for short) is a term used for the discovery, extraction and analysis of the business data in order to provide better decision support for the organizations. **Better decisions make business - better**, since this information helps to *make more money and reduce expenses*, performing *better than the competition* in the long run.

There are *multiple forms of Business Intelligence*, ranging from data-mining and reporting up to KPIs and predictive analytics. Each form has unique capabilities and requirements in order to be of any use.

We'll focus on the **simplicity of adding predictive analytics** to a **simple .NET line of business application** that has *not even been designed for data mining*.

Let's consider basic situation, where some company sells products to the customers, while keeping track of the inventory in stock and all the orders. There are lots of these. Such scenario could be easily and rapidly *modeled* with the eXpress Application Framework (XAF for short), a technology developed by the .NET component vendor *DevExpress*.

Given we have customer, products and orders, generated database structure might look like this:

ProductSale	
PK	Oid
FK2,I2	Product
FK1,I1	Order
	Count
	Price
	Discount
	OptimisticLockField
	GCRecord

Order	
PK	Oid
	Created
FK1,I1	Customer
	Total
	Summary
	OptimisticLockField
	GCRecord

Country	
PK	Oid
	Name
	PhoneCode
	OptimisticLockField
	GCRecord

Product	
PK	Oid
	Name
	MnfPN
	Quantity
	Price
	Sku
	Condition
	OptimisticLockField
	GCRecord

Customer	
PK,FK1,I1	Oid

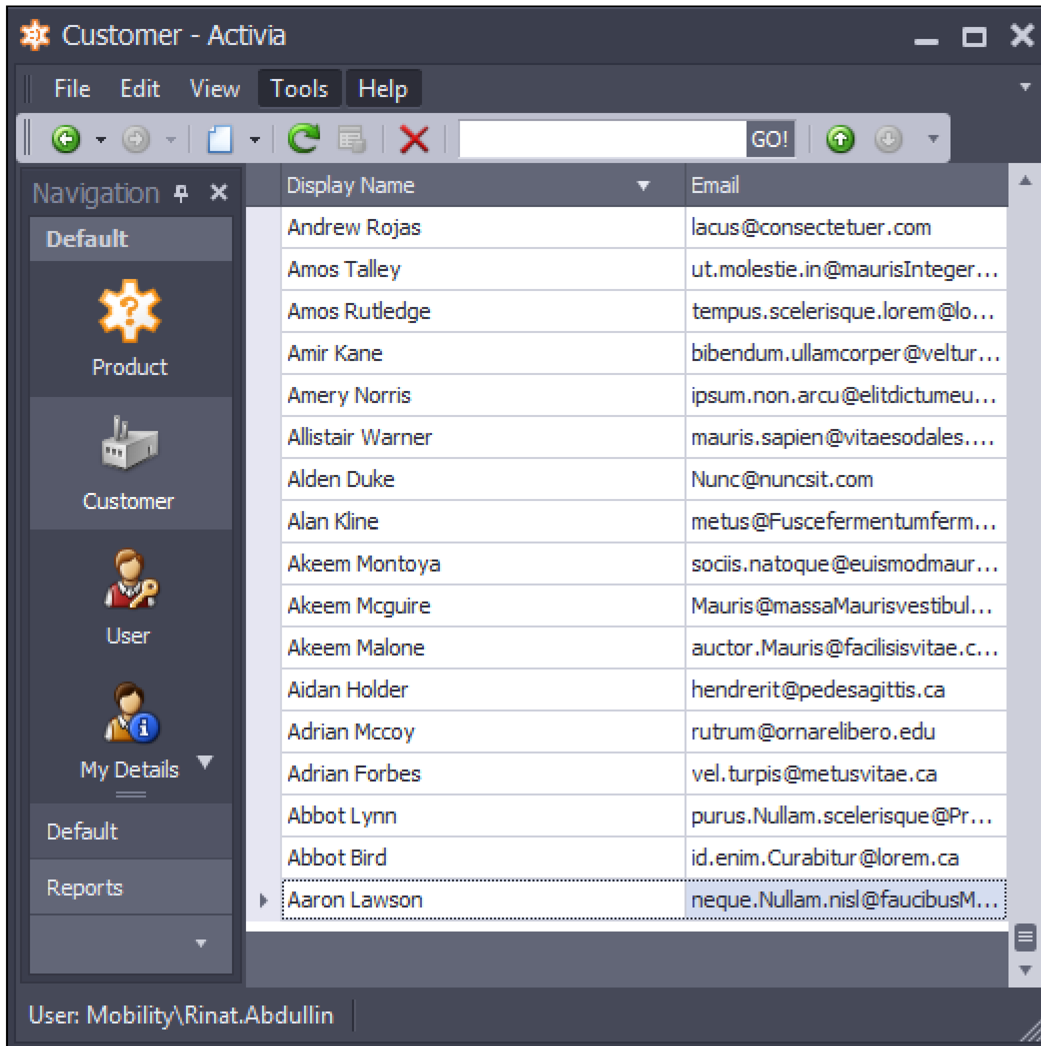
Address	
PK	Oid
	Street
	City
	StateProvince
	ZipPostal
FK1,I1	Country
	OptimisticLockField
	GCRecord

Organization	
PK,FK1,I1	Oid
	FullName
	Profile
	Email
	WebSite
	Description
	Name

PhoneNumber	
PK	Oid
	Number
FK1,I1	Party
	PhoneType
	OptimisticLockField
	GCRecord

Party	
PK	Oid
	Photo
FK1,I1	Address1
FK2,I2	Address2
	OptimisticLockField
	GCRecord

And the generated User Interface would look like this:



Order

File Edit View Tools

Order

Customer: Aaron Lawson Summary: 5 items, 120,20

Total: 120,20€ Created: 13.01.2010

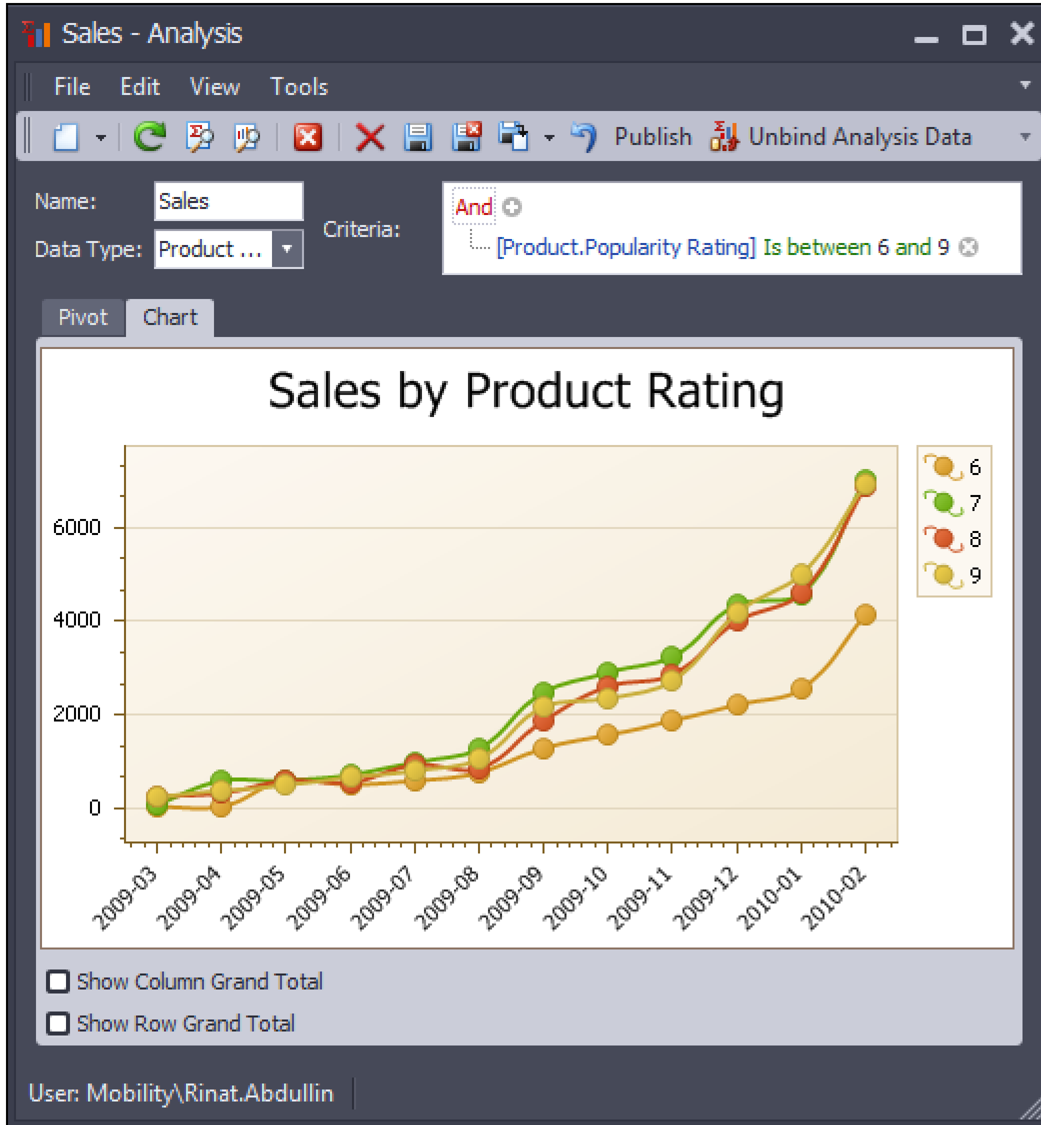
Products

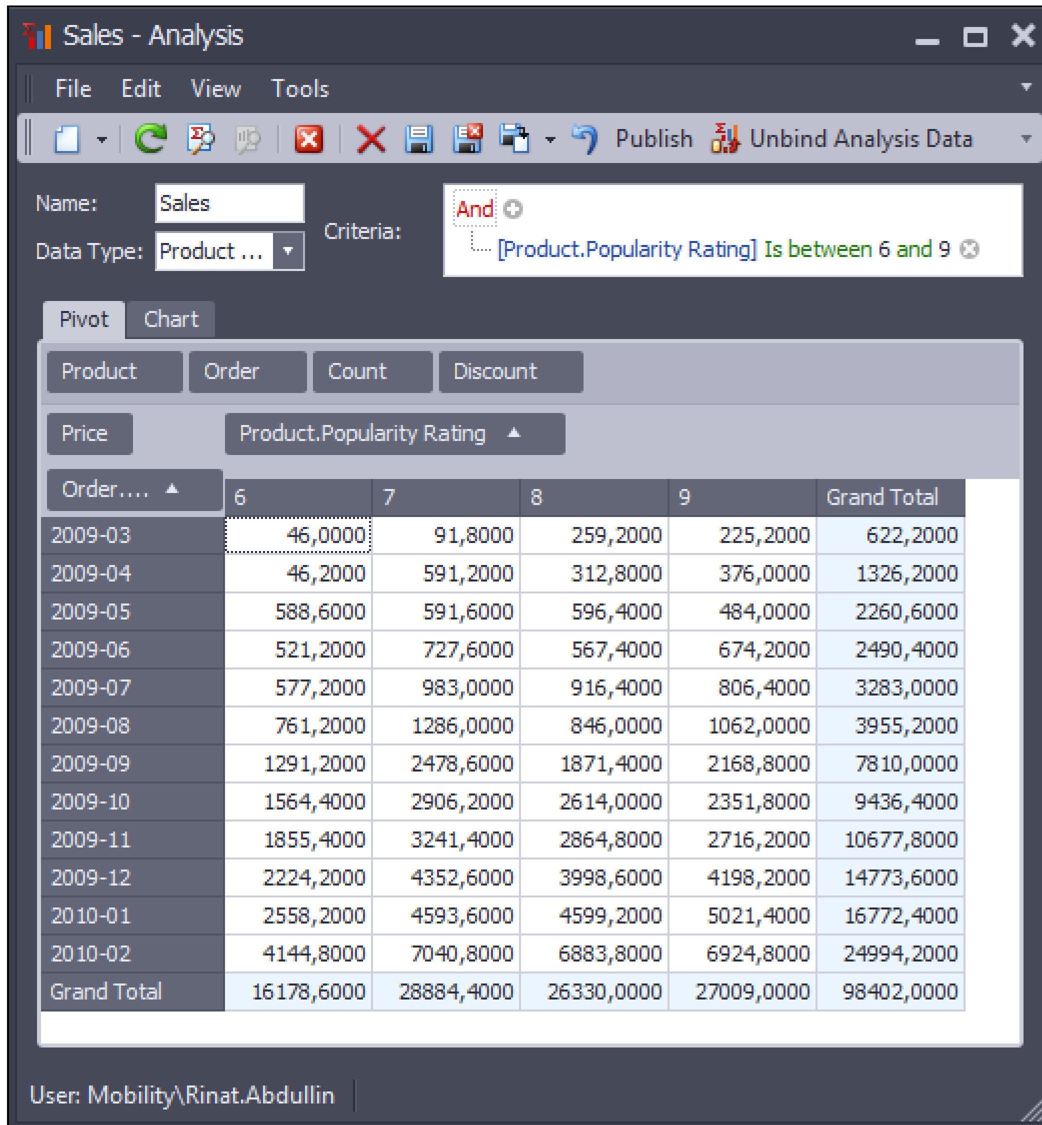
Product	Count	Price	Discount
PLANETSTRIKE RULEBOOK	2	20,00€	0,
40K BATTLE MISSIONS	1	19,80€	0,
IMPERIAL GUARD VALKYRIE	1	46,40€	0,
CADIAN SHOCK TROOPS	3	16,00€	0,1
APOCALYPSE: VOL II RELOADED	3	18,00€	0,1

User: Mobility\Rinat.Abdullin

All screen-shots are real, but represent completely mock data, created by GenerateData.com, googling up and parsing some old catalogues with filehelpers library, Random.Next. API keys are mocked as well.

Let's us get back to the prototype .NET application. If we turn on the Pivot Chart Module within the XAF, then ability for a simple data-mining would be added, along with the charting capabilities. So with a little bit of drag-n-dropping, users can get some nice historical reports and analytics.





Order....	6	7	8	9	Grand Total
2009-03	46,0000	91,8000	259,2000	225,2000	622,2000
2009-04	46,2000	591,2000	312,8000	376,0000	1326,2000
2009-05	588,6000	591,6000	596,4000	484,0000	2260,6000
2009-06	521,2000	727,6000	567,4000	674,2000	2490,4000
2009-07	577,2000	983,0000	916,4000	806,4000	3283,0000
2009-08	761,2000	1286,0000	846,0000	1062,0000	3955,2000
2009-09	1291,2000	2478,6000	1871,4000	2168,8000	7810,0000
2009-10	1564,4000	2906,2000	2614,0000	2351,8000	9436,4000
2009-11	1855,4000	3241,4000	2864,8000	2716,2000	10677,8000
2009-12	2224,2000	4352,6000	3998,6000	4198,2000	14773,6000
2010-01	2558,2000	4593,6000	4599,2000	5021,4000	16772,4000
2010-02	4144,8000	7040,8000	6883,8000	6924,8000	24994,2000
Grand Total	16178,6000	28884,4000	26330,0000	27009,0000	98402,0000

So far so good, but we want more. Let us spend a little bit of time solving **two challenges** that come up with this default XAF Analytics module:

- Running reports *stresses the underlying database* a bit.
- *Getting predictive analytics* - to see how the sales will look like all the way up to June (4 months ahead, historical data is stopping in February).

As it turns out, the solution to these two is rather simple and straightforward. We just need to define our own XAF Module with a little bit of customization.

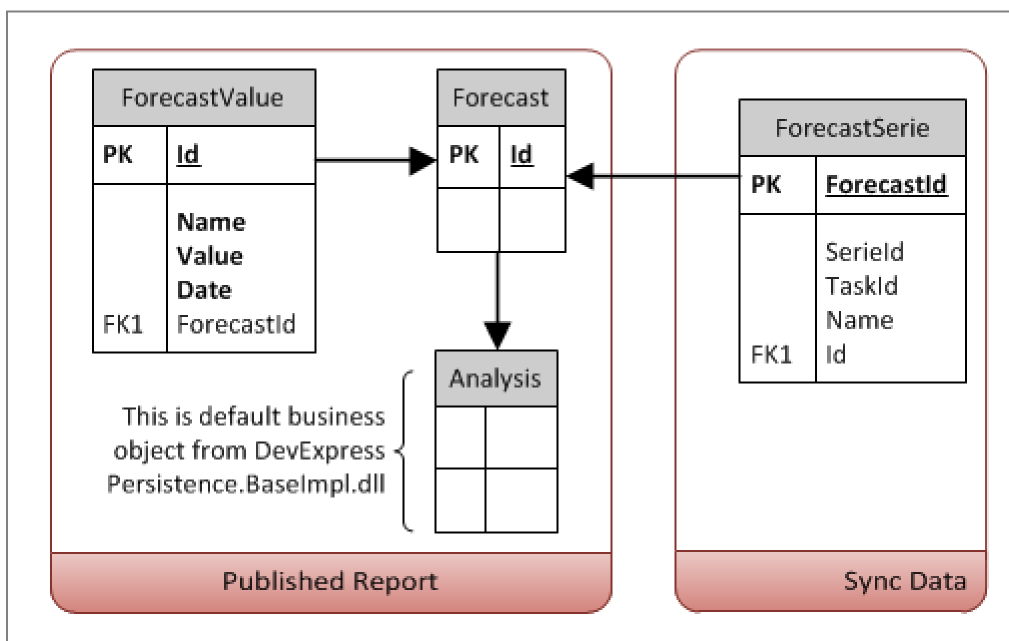
Performance could be optimized *by capturing visible report data* from the Pivot and *publishing it to the separate table*. This table would be in another database or even in the cloud (SQL Azure being the simplest solution and some sort of big table - cheapest), for all we care. Some gain could be obtained even if the dedicated table is in the same database. Indeed, we don't need to perform a 7-JOIN query any more.

Using this dedicated table approach, whenever a manager accesses already published report (i.e.: to make a presentation, export into PDF or simply play with the numbers), database is unlikely to be stressed. In a way that's a simple case of Command-query separation at work here.

Once we've got the report with some data (i.e.: sales), adding good forecasts requires either a development team with PhD and statistical skills or usage of some outsourced forecasting web service.

At Lokad, we believe the second option to be much cheaper and faster. So we'll go for **forecasting service integration to handle the complex math for us**. We send a copy of the published numbers to Lokad Forecasting API and merge the results with the published report afterwards.

Development-wise, we create a controller for the *Analysis* object (available from the existing BaseImpl classes) and a "Publish" action. The action grabs data from our analytics report and publish it to a specialized type of the business intelligence object, which actually inherits from the *Analysis* class.



Within the "Publish" action we grab cell contents and save them to another report. The simplest way of grabbing the data is:

```
var pivotCells = pivotGridControl.Cells;

for (int col = 0; col < pivotCells.ColumnCount; col++)
{
  for (int row = 0; row < pivotCells.RowCount; row++)
  {
    var cell = pivotCells.GetCellInfo(col, row);

    if (cell.Item.IsGrandTotalAppearance)
      continue;

    var colValue = cell.GetFieldValue(cell.ColumnField);
    var rowValue = cell.GetFieldValue(cell.RowField);
    var cellCalue = cell.Value;
```

Once we've got cells, we simply figure out, which side of the cell holds the date and which - the name, parse the results and save them to a forecast value list:

```
    if (TryConvert(colValue, out date))
    {
      name = rowValue.ToString();
    }
    else if (TryConvert(rowValue, out date))
    {
      name = colValue.ToString();
    }
    else
    {
      Messaging.DefaultMessaging.Show("Problem",
        "Either column or row should be convertable to date.");
      return;
    }
    if (!TryConvertValue(cellCalue, out value))
    {
      Messaging.DefaultMessaging.Show("Problem",
        "Can't convert cell to a number.");
      return;
    }
    var forecastValue = new ForecastValue(session)
      {
        Date = date,
        Name = name,
        Value = value
      };
    list.Add(forecastValue);
  }
}

forecast.Values.AddRange(list);
forecast.ObjectTypeName = typeof (ForecastValue).FullName;
forecast.Name = "Forecast " + DateTime.Now.ToString();
forecast.Prefix = Guid.NewGuid().ToString().Substring(0, 6);
session.Save(forecast);
```

Where *ForecastValue* is a collection within the *Forecast* object (derived from the *Analysis* of the base implementation library).

The only important thing is that *Forecast* object is smart enough to fetch its own published values, when we ask for the analytics data. This done with the criteria bound to the *Oid*. Since its primary key might be empty at the time of creation, we lazily update the criteria on next DB load; there is no rush here:

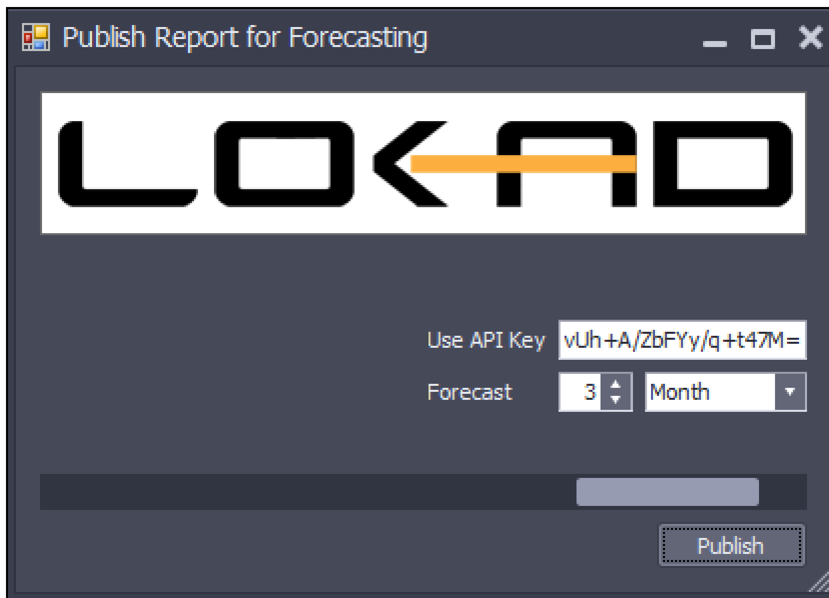
```
protected override void OnLoaded()
{
  if (string.IsNullOrEmpty(Criteria) || Criteria.Contains("00000"))
  {
    Criteria = CriteriaOperator.Parse("Forecast.Oid=?", Oid).ToString();
  }
  base.OnLoaded();
}
```

Although **we could immediately access the published report**, we need forecasts. Forecasts are obtained by uploading data to the *Forecasting API* and then later merging values into the published report. *ForecastSerie* table (and entities) is used to keep basic data needed for this process.

The process could start like this:

```
using (var api = new LokadApiForm(forecast))
{
  api.ShowDialog();
}
```

where the form looks as simple as:



Users just need to enter their Forecasting API key, select forecasting options and press *Publish*. Lokad API keys could be created in the "Users" section of the web management UI:

Lokad API Keys

API keys represent a secure way to access Lokad Forecasting services from a wide number of software products.

	API Key	Name	Last Login
Delete	XqXCSgECVWzA4HgVegH5okOeNCK4sllJySKAG8g=		2010-03-27 08:21

Create new Key

Given all this, with the latest version of Lokad Forecasting SDK, we need to:

- *Connect* to the API.
- *Compose* our scattered forecast values into the *regular time series*.
- *Save series to the API*, while stripping actual names (this keeps data perfectly valid for forecasting, but makes it useless for if others, if we decide to share the Lokad account with the other people in the company).
- Save the API key and sync data into the database, so that later, when accessing the published report, we would be able to *merge forecasts into it*.

The simplified core code (bound to the "Publish" button in *LokadApiForm*) looks like:

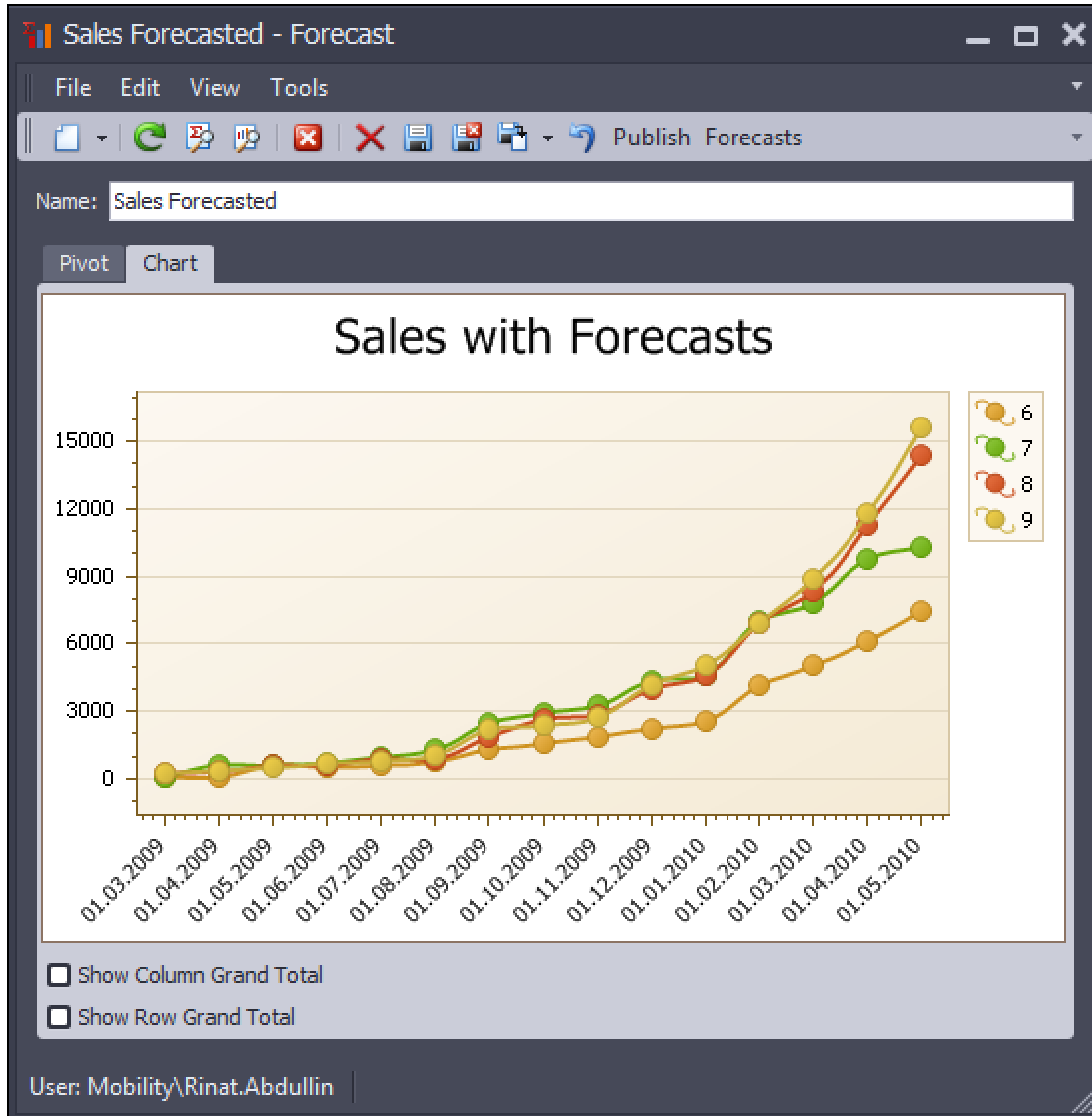
```
var service = ServiceFactory.GetConnector(apiKeyEditor.Text);

// wipe all old series for this report
service.DeleteSeries(service.GetSeriesWithPrefixLegacy(forecast.Prefix));

// compose loose values into time series and upload them
var lookups = forecast.Values.ToLookup(v => v.Name, v =>
    new TimeValue{ Time = v.Date, Value = v.Value }
  );

var nameToKey = lookups.ToDictionary(n => n.Key, n => Guid.NewGuid());
var keyToName = nameToKey.ToDictionary(n => n.Value, n => n.Key);
var series = lookups.ToArray(s => new SerieInfo()
  {
    Name = nameToKey[s.Key].ToString()
  });

service.AddSeriesWithPrefix(series, forecast.Prefix);
var serieIdToKey = series.ToDictionary(si => si.SerieID,
  si => new Guid(si.Name));
service.UpdateSerieSegments(series.Select(g => new SegmentForSerie(
  g, lookups[keyToName[new Guid(g.Name)]].ToArray())));

// define forecasting tasks
var period = MaybeParse
  .Enum<Period>(_periodText.SelectedText)
  .ExposeException("Invalid period");

var interval = Convert.ToInt32(_intervalEdit.Value);
var tasks = series.Convert(si => new TaskInfo(si)
  {
    FuturePeriods = interval,
    Period = period,
  });
service.AddTasks(tasks);

// save sync information into our database
forecast.Key = apiKeyEditor.Text;
forecast.Series.AddRange(tasks.Convert(ti =>
  new ForecastSerie(forecast.Session)
  {
    Serie = ti.SerieID,
    Task = ti.TaskID,
    Name = keyToName[serieIdToKey[ti.SerieID]]
  }));
```

Second part of the job is to retrieve forecasts and merge them back into the published analytics report. The code below goes to a Simple Action that is bound to the *Forecast* business object:

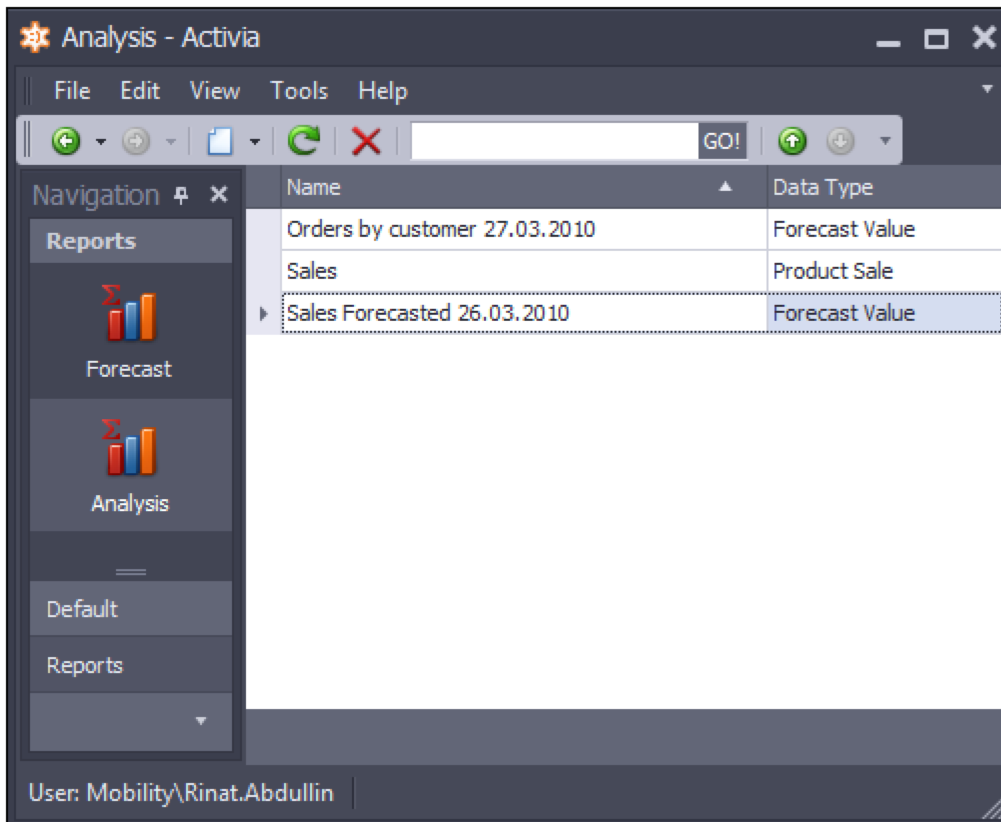
```
// Retrieve the forecasts
var forecast = (Forecast)View.CurrentObject;
var service = ServiceFactory.GetConnector(forecast.Key);
var session = forecast.Session;

var forecasts = service.GetForecasts(forecast.Series.ToArray(s =>
  new TaskInfo()
  {
    SerieID = s.Serie,
```

```
    TaskID = s.Task
  }));

// merge forecast values with the published report
var dates = forecasts.SelectMany(f => f.Values.Select(d => d.Time))
  .Distinct().ToSet();
var intersections = forecast.Values
  .Where(v => dates.Contains(v.Date)).ToArray();
session.Delete(intersections);

var taskToName = forecast.Series.ToDictionary(fs => fs.Task, fs => fs.Name);

var values = forecasts.SelectMany(f => f.Values.Select(v =>
  new ForecastValue(session)
{
  Category = "Forecast",
  Date = v.Time,
  Name = taskToName[f.TaskID],
  Value = v.Value
}));
forecast.Values.AddRange(values);
session.Save(forecast);
Messaging.DefaultMessaging.Show("Ready", "Report Ready");
```

When we access the published report after it had been merged with the forecast values, then we see, that our **data has just got enriched with the forecast estimates** until June:



And since the published report with forecasts is just another *Analytics* object, we can access it alike from the reporting section:



This does not look that bad for a hack-technologies-together-in-a-day effort, does it?

Still, there is **a lot of room for improvement** for specific scenarios with the analytics powered by Lokad forecasting services.

First, from the business viewpoint, we could benefit from the **enterprise integration** in order to automate some repetitive tasks related to handling orders, keeping track of the inventory, automatically and pro-actively handling stock shortages or merely sending out notifications. We could use something like NServiceBus to jump-start the development here. Forecasts and predictive analytics could be used to make the process even more efficient for the business.

Then there's always room for the **enterprise dashboard with some KPI indicators** and drill-down reports, allowing the owner to keep track of the business heart beats. Actually, if we keep history of the important domain events, this also provides rich business audit logs for the stakeholders (what happened when, where and why). This information could also be used for the better business intelligence feedback in form of event analysis and detection of various scenarios. Lokad can handle such information in form of tags and events.

At the very least, we could **automate report publishing and forecast retrieval**, since "click-click-click" is too boring, plus there aren't many users working concurrently during the nights. Forecasts could be visualized better in our Pivot and Chart displays. We could also send more

data from the Pivot (i.e.: values of the outermost fields) to the Lokad API in form of tags, which would enhance the forecasting precision.

Then, how about some business rules that are editable by the end-users, powered by the internal DSL and linked to the analytics:

```
when product sales_estimates
   are_greater_than_supplies_by 25%
   then alert

when product sales
   increase_by_more_than 25%
   compared_to previous.week
   then notify

when customer orders
   increase_by_more_than 50%
   compared_to previous.month
   then make_customer_preferred
```

Obviously these business intelligence features require more than a single day to be implemented. Yet, they are feasible with the technologies and services we have at hand in .NET, if the business could use such capabilities to become more profitable.

Be sure to check up regularly Lokad.com to stay up-to date with our rich integration portfolio and the business opportunities it provides.

All comments, thoughts, questions and any other feedback are welcome and appreciated.

So, what do you think? Lokad Support would love to hear from you!