18. MVC/MVVM Web applications

Adopt a mORMot

We will now explain how to build a MVC/MVVM web application using *mORMot*, starting from the "*30 - MVC Server*" sample.

This little web application publishes a simple BLOG, not fully finished yet (this is a *Sample*, remember!). But you can still execute it in your desktop browser, or any mobile device (thanks to a simple *Bootstrap*-based responsive design), and see the articles list, view one article and its comments, view the author information, log in and out.

This sample is implemented as such:

MVVM	Source	mORMot
Model	`MVCModel.pas`	`TSQLRestServerDB` ORM over a *SQlite3* database
View	`*.html`	*Mustache template engine* (page 410) in the *Views* sub-folder
ViewModel	`MVCViewModel.pas`	Defined as one `IBlogApplication` interface

For the sake of the simplicity, the sample will create some fake data in its own local *SQlite3* database, the first time it is executed.

18.1. MVCModel

The `MVCModel.pas` unit defines the database *Model*, as regular `TSQLRecord` classes.
For instance, you would find the following type definitions:

```
  TSQLContent = class(TSQLRecordTimeStamped)
  private ...
```

```
  published
    property Title: RawUTF8 index 80 read fTitle write fTitle;
    property Content: RawUTF8 read fContent write fContent;
    property Author: TSQLAuthor read fAuthor write fAuthor;
    property AuthorName: RawUTF8 index 50 read fAuthorName write fAuthorName;
  end;

  TSQLArticle = class(TSQLContent)
  private ...
  public
    class function CurrentPublishedMonth: Integer;
    class procedure InitializeTable(Server: TSQLRestServer; const FieldName: RawUTF8;
      Options: TSQLInitializeTableOptions); override;
  published
    property PublishedMonth: Integer read fPublishedMonth write fPublishedMonth;
    property Abstract: RawUTF8 index 1024 read fAbstract write fAbstract;
  end;

  TSQLComment = class(TSQLContent)
  private ...
  published
    property Article: TSQLArticle read fArticle write fArticle;
  end;
```

Then the whole database model will be created in this function:

```
function CreateModel: TSQLModel;
begin
  result := TSQLModel.Create([TSQLBlogInfo,TSQLCategory,TSQLAuthor,
    TSQLArticle,TSQLComment],'blog');
  TSQLArticle.AddFilterOrValidate('Title',TSynFilterTrim.Create);
  TSQLArticle.AddFilterOrValidate('Title',TSynValidateText.Create);
  TSQLArticle.AddFilterOrValidate('Content',TSynFilterTrim.Create);
  TSQLArticle.AddFilterOrValidate('Content',TSynValidateText.Create);
end;
```

As you can discover:

- We used `class` inheritance to gather properties for similar tables;
- Some classes are *not* part of the model, since they are just `abstract` parents, e.g. `TSQLContent` is not part of the model, but `TSQLArticle` and `TSQLComment` are;
- We defined some regular *one-to-one* relationships, e.g. every `Content` (which may be either an `Article` or a `Comment`) will be tied to one `Author` - see *"One to one" or "One to many"* (page 130);
- We defined some regular *one-to-many* relationships, e.g. every `Comment` will be tied to one `Article`;
- Some properties are defined (and stored) twice, e.g. `TSQLContent` defines one `AuthorName` field in addition to the `Author` ID field, as a convenient direct access to the author name, therefore avoiding a JOINed query at each `Article` or a `Comment` display - see *Shared nothing architecture (or sharding)* (page 133);
- We defined the maximum expected width for text fields (e.g. via `Title: RawUTF8 index 80`), even if it won't be used by *SQLite3* - it would ease any eventual migration to an external database, in the future - see *External database access* (page 205);
- Some validation rules are set using `TSQLArticle.AddFilterOrValidate()` method, which would be applied before an article is stored;
- The whole application would run without writing any SQL, but just high-level ORM methods;
- Even if we want to avoid writing SQL, we tried to modelize the data to fit regular RDBMS expectations, e.g. for most used queries (like the one run from the main page of the BLOG).

Foreign keys and indexes are managed as such:

- The `TSQLRecord.ID` primary key of any ORM class will be indexed;
- For both *one-to-one* and *one-to-many* relationships, indexes are created by the ORM: for instance,

TSQLArticle.Author and TSQLComment.Author will be indexed, just as TSQLComment.Article;
- An index would be needed for TSQLArticle.PublishedMonth field, which is used to display a list of publication months in the main BLOG page, and link to the corresponding articles.
  The following code will take care of it:

```
class procedure TSQLArticle.InitializeTable(Server: TSQLRestServer;
  const FieldName: RawUTF8; Options: TSQLInitializeTableOptions);
begin
  inherited;
  if (FieldName='') or (FieldName='PublishedMonth') then
    Server.CreateSQLIndex(TSQLArticle,'PublishedMonth',false);
end;
```

The ORM is defined to run over a *SQLite3* database in the main MVCServer.dpr program, then served via a HTTP server as defined in MVCServer.dpr:

```
  aModel := CreateModel;
  try
    aServer := TSQLRestServerDB.Create(aModel,ChangeFileExt(paramstr(0),'.db'));
    try
      aServer.DB.Synchronous := smNormal;
      aServer.DB.LockingMode := lmExclusive;
      aServer.CreateMissingTables;
      aApplication := TBlogApplication.Create(aServer);
      try
        aHTTPServer := TSQLHttpServer.Create('8092',aServer,'+',useHttpApiRegisteringURI);
        try
          aHTTPServer.RootRedirectToURI('blog/default'); // redirect localhost:8092
          writeln('"MVC Blog Server" launched on port 8092');
  ....
```

In comparison to a regular *Client-Server process* (page 264), we instantiated a TBlogApplication, which will *inject* the MVC behavior to aServer and aHTTPServer. The same *mORMot* program could be used as a RESTful server for remote *Object-Relational Mapping (ORM)* (page 78) and *Service-Oriented Architecture (SOA)* (page 76), and also for publishing a web application, sharing the same data and business logic, over a single HTTP URI and port.
A call to RootRedirectToURI() will let any http://localhost:8092.. HTTP request be redirected to http://localhost:8092/blog/default.. which is our BLOG application main page. The other URIs could be used as usual, as any *mORMot*'s *JSON RESTful Client-Server* (page 243).

18.2. MVCViewModel

18.2.1. Defining the commands

The MVCViewModel.pas unit defines the *Controller* (or *ViewModel*) of the "*30 - MVC Server*" sample application.
It uses the mORMotMVC.pas unit , which is the main MVC kernel for the framework, allowing to easily create *Controllers* binding the ORM/SOA features (mORMot.pas) to the *Mustache* Views (SynMustache.pas).

First of all, we defined an interface, with the expected methods corresponding to the various *commands* of the web application:

```
  IBlogApplication = interface(IMVCApplication)
    procedure ArticleView(
      ID: integer; var WithComments: boolean; Direction: integer;
      out Article: TSQLArticle; out Author: TSQLAuthor;
      out Comments: TObjectList);
    procedure AuthorView(
      var ID: integer; out Author: variant; out Articles: RawJSON);
```

```
    function Login(
      const LogonName,PlainPassword: RawUTF8): TMVCAction;
    function Logout: TMVCAction;
    procedure ArticleEdit(
      var ID: integer; const Title,Content: RawUTF8;
      const ValidationError: variant;
      out Article: TSQLArticle);
    function ArticleCommit(
      ID: integer; const Title,Content: RawUTF8): TMVCAction;
  end;
```

In fact, `IMVCApplication` is defined as such in `mORMotMVC.pas`:

```
  IMVCApplication = interface(IInvokable)
    ['{C48718BF-861B-448A-B593-8012DB51E15D}']
    procedure Default(var Scope: variant);
    procedure Error(var Msg: RawUTF8; var Scope: variant);
  end;
```

As such, the `IBlogApplication` will define the following web pages, corresponding to each of its methods: *Default, Error, ArticleView, AuthorView, Login, Logout, ArticleEdit* and *ArticleCommit*. Each command of this application will map an URI, e.g. `/blog/default` or `/blog/login` - remember that our model defined `'blog'` as its root URI. You may let all commands be accessible from a sub-URI (e.g. `/blog/web/default`), but here this is not needed, since we are creating a "pure web" application.

Each command will have its own *View*. For instance, you will find `Default.html`, `Error.html` or `ArticleView.html` in the *"Views"* sub-folder of the sample. If you did not supply any file in this folder, some void files would be created.

Incoming method parameters of each method (i.e. defined as `const` or `var`) will be transmitted on the URI, encoded as regular HTTP parameters, whereas outgoing method parameters (i.e. defined as `var` or `out`) would be transmitted to the *View*, as data context for the rendering. Simple types are transmitted (like `integer` or `RawUTF8`); but you would also find ORM classes (like `TSQLAuthor`), an outgoing `TObjectList`, or some `variant` - which may be either values or a complex *TDocVariant custom variant type* (page 97).

In fact, you may find out that the *Login, Logout* and *ArticleCommit* methods do not have any outgoing parameters, but were defined as `function` returning a `TMVCAction` record.
This type is declared as such in `mORMotMVC.pas`:

```
  TMVCAction = record
    RedirectToMethodName: RawUTF8;
    RedirectToMethodParameters: RawUTF8;
    ReturnedStatus: cardinal;
  end;
```

Any method returning a `TMVCAction` content won't render directly any view, but will allow to go directly to another method, for proper rendering, just by providing a method name and some optional parameters.
Note that even the regular views, i.e. the methods which do not have this `TMVCAction` parameter, may break the default rendering process on any error, raising an `EMVCApplication` exception which will in fact redirect the view to another page, mainly the `Error` page.

To better understand how it works, run the "*30 - MVC Server*" sample. Remember that to be able to register the port #8092 for the `http.sys` server, you would need to run the `MVCServer.exe` program at least once with *Windows Administrator* rights - see *URI authorization as Administrator* (page 272). Then point your browser to http://localhost:8092/.. - you will see the main page of the BLOG, filled with some random data. Quite some "blabla", to be sincere!

What you see is the `Default` page rendered. The `IBlogApplication.Default()` method has been

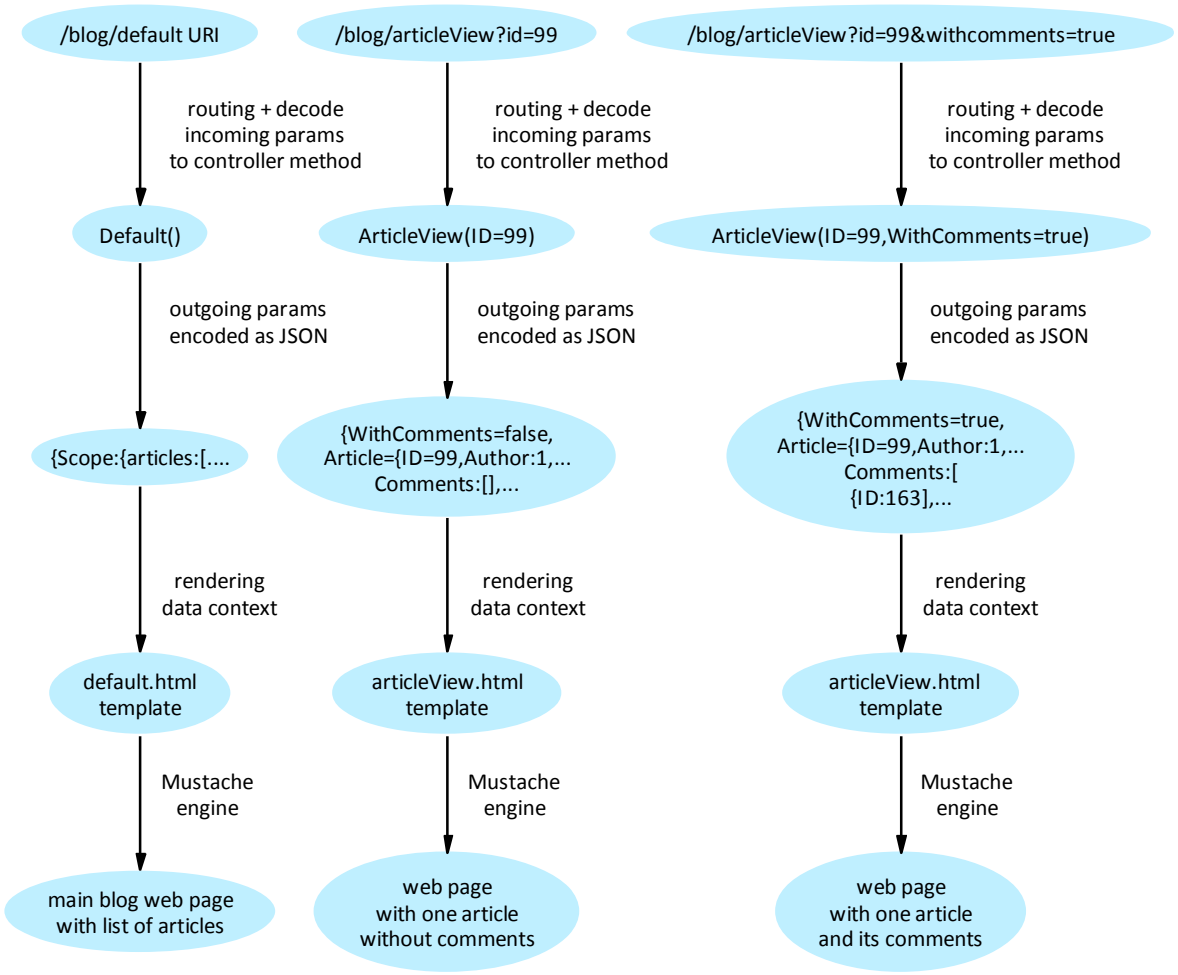
called, then the outgoing `Scope` data has been rendered by the `Default.html` *Mustache* template.

If you click on an article title, it will go to http://localhost:8092/blog/articleView?id=99.. - i.e. calling `IBlogApplication.ArticleView()` with the `ID` parameter containing 99, and other incoming parameters (i.e. `WithComments` and `Direction`) set to their default value (i.e. respectively `false` and `0`). The `ArticleView()` method will then read the `TSQLArticle` data from the ORM, then send it to the `ArticleView.html` *Mustache* template.

Now, just change in your browser the URI from http://localhost:8092/blog/articleView?id=99.. (here we clicked on the `Article` with ID=99) into http://localhost:8092/blog/articleView/json?id=99.. (i.e. entering `/articleView`**`/json`** instead of `/articleView`, as a fake sub-URI).
Now the browser is showing you the JSON data context, as transmitted to the `ArticleView.html` template. Just check both the JSON content and the corresponding *Mustache* template: I think you will find out how it works. Take a look at *Mustache template engine* (page 410) as reference.

From any blog article view, click on the "`Show Comments`" button: you are redirected to a new page, at URI http://localhost:8092/blog/ArticleView?id=99&withComments=true#comments.. and now the comments corresponding to the article are displayed. If you click on the "`Previous`" or "`Next`" buttons, a new URI http://localhost:8092/blog/ArticleView?id=99&withComments=true&direction=1.. will be submitted: in fact, `direction=1` will search for the previous article, and we still have the `withComments=true` parameter set, so that the user would be able to see the comments, as expected. If you click on the "`Hide Comments`" button, the URI would change to be without any `withComments=true` parameter - i.e. http://localhost:8092/blog/ArticleView?id=98#comments.. : now the comments won't be displayed.

The sequence is rendered as such:



mORMot MVC/MVVM URI - Commands sequence

In this diagram, we can see that each HTTP request is stateless, uncoupled from the previous. The user experience is created by changing the URI with additional parameters (like `withComments=true`). This is how the web works.

Then try to go to http://localhost:8092/blog/mvc-info.. - and check out the page which appears. You will get all the information corresponding to your application, especially a list of all available commands:

```
/blog/Default?Scope=..[variant]..
/blog/Error?Msg=..[string]..&Scope=..[variant]..
/blog/ArticleView?ID=..[integer]..&WithComments=..[boolean]..&Direction=..[integer]..
/blog/AuthorView?ID=..[integer]..
/blog/Login?LogonName=..[string]..&PlainPassword=..[string]..
/blog/Logout
/blog/ArticleEdit?ID=..[integer]..&Title=..[string]..&Content=..[string]..&ValidationError=..[vari
ant]..
/blog/ArticleCommit?ID=..[integer]..&Title=..[string]..&Content=..[string]..
```

And every view, including its data context, e.g.

```
/blog/AuthorView?ID=..[integer]..
{{Main}}: variant
{{ID}}: integer
{{Author}}: TSQLAuthor
{{Articles}}: variant
```

You may use this page as reference when writing your *Mustache* Views. It will reflect the exact state of the running application.

18.2.2. Implementing the Controller

To build the application *Controller*, we would need to implement our `IBlogApplication interface`.

```
  TBlogApplication = class(TMVCApplication,IBlogApplication)
  ...
  public
    constructor Create(aServer: TSQLRestServer); reintroduce;
    procedure Default(var Scope: variant);
    procedure ArticleView(ID: integer; var WithComments: boolean;
      Direction: integer;
      out Article: TSQLArticle; out Author: variant;
      out Comments: TObjectList);
    ...
  end;
```

We defined a new class, inheriting from `TMVCApplication` - as defined in `mORMotMVC.pas`, and implementing our expected interface. `TMVCApplication` will do all the low-level plumbing for you, using a set of implementation classes.

Let's implement a simple command:

```
procedure TBlogApplication.AuthorView(var ID: integer; out Author: TSQLAuthor;
  out Articles: RawJSON);
begin
  RestModel.Retrieve(ID,Author);
  if Author.ID<>0 then
    Articles := RestModel.RetrieveListJSON(
      TSQLArticle,'Author=? order by id desc limit 50',[ID],ARTICLE_FIELDS) else
    raise EMVCApplication.CreateGotoError(HTML_NOTFOUND);
end;
```

By convention, all parameters are allocated when `TMVCApplication` will execute a method. So you do not need to allocate or handle the `Author: TSQLAuthor` instance lifetime.
You have direct access to the underlying `TSQLRest` instance via `TMVCApplication.RestModel`: so all CRUD operations are available. You can let the ORM do the low level SQL work for you: to retrieve all information about one `TSQLAuthor` and get the list of its associated articles, we just use a `TSQLRest` method with the appropriate WHERE clause. Here we returned the list of articles as a `RawJSON`, so that they will be transmitted as a JSON array, without any intermediate marshalling to `TSQLArticle` instances.
In case of any error, an `EMVCApplication` will be raised: when such an exception happens, the `TMVCApplication` will handle and convert it into a page change, and a redirection to the `IBlogApplication.Error()` method, which will return an error page, using the `Error.html` view template.

Let's take a look at a bit more complex method, which we talked about in *mORMot MVC/MVVM URI - Commands sequence* (page 427):

```
procedure TBlogApplication.ArticleView(
  ID: integer; var WithComments: boolean; Direction: integer;
  out Article: TSQLArticle; out Author: variant; out Comments: TObjectList);
var newID: integer;
const WHERE: array[1..2] of PUTF8Char = (
  'ID<? order by id desc','ID>? order by id');
begin
  if Direction in [1,2] then // allows fast paging using index on ID
    if RestModel.OneFieldValue(TSQLArticle,'ID',WHERE[Direction],[],[ID],newID) and
      (newID<>0) then
      ID := newID;
```

```
  RestModel.Retrieve(ID,Article);
  if Article.ID<>0 then begin
    Author := RestModel.RetrieveDocVariant(
      TSQLAuthor,'ID=?',[Article.Author.ID],'FirstName,FamilyName');
    if WithComments then begin
      Comments.Free; // we will override the TObjectList created at input
      Comments := RestModel.RetrieveList(TSQLComment,'Article=?',[Article.ID]);
    end;
  end else
    raise EMVCApplication.CreateGotoError(HTML_NOTFOUND);
end;
```

This method has to manage several use cases:

- Display an `Article` from the database;
- Retrieve the `Author` first name and family name;
- Optionally display the associated `Comments`;
- Optionally get the previous or next `Article`;
- Trigger an error in case of an invalid request.

Reading the above code is enough to understand how those 5 features are implemented in this method. The incoming parameters, as triggered by the *Views*, are used to identify the action to be taken. Then `TMVCApplication.RestModel` methods are used to retrieve the needed information directly from the ORM. Outgoing parameters (`Article,Author,Comments`) are transmitted to the *Mustache View*, for rendering.

In fact, there are several ways to retrieve your data, using the `RestModel` ORM methods. For instance, in the above code, we used a `TObjectList` to transmit our comments.
But we may have used a *TDocVariant custom variant type* (page 97) parameter:

```
procedure TBlogApplication.ArticleView(
  ID: integer; var WithComments: boolean; Direction: integer;
  out Article: TSQLArticle; out Author: variant; out Comments: variant);
 ...
    if WithComments then
      Comments := RestModel.RetrieveDocVariantArray(TSQLComment,'','Article=?',[Article.ID],'');
```

Or with a `RawJSON` kind of output parameter:

```
procedure TBlogApplication.ArticleView(
  ID: integer; var WithComments: boolean; Direction: integer;
  out Article: TSQLArticle; out Author: variant; out Comments: RawJSON);
 ...
    if WithComments then
      Comments := RestModel.RetrieveListJSON(TSQLComment,'Article=?',[Article.ID],'');
```

Using a `RawJSON` will be in fact the fastest way of processing the information on the server side. If your purpose is just to retrieve some data and push it back to the view, `RawJSON` is just perfect. But having a `TObjectList` may be convenient if you need to run some `TSQLRecord` methods on the returned list; or a `TDocVariant` array may have its needs, if you want to create some meta-object gathering all information, e.g. for `Scope` as returned by the `Default` method:

```
procedure TBlogApplication.Default(var Scope: variant);
 ...
  if not fDefaultData.AddExistingProp('Archives',Scope) then
    fDefaultData.AddNewProp('Archives',RestModel.RetrieveDocVariantArray(
      TSQLArticle,'','group by PublishedMonth order by PublishedMonth desc limit 12',[],
      'distinct(PublishedMonth),max(ID)+1 as FirstID'),Scope);
end;
```

You can notice how the calendar months are retrieved from the database, using a safe `fDefaultData: ILockedDocVariant` private field to store the value as cache, in a thread-safe manner (we will see later more about how to implement thread-safety). If the `'Archives'` value is in the

fDefaultData cache, it will be returned immediately as part of the Scope returned document. Otherwise, it will use RestModel.RetrieveDocVariantArray to retrieve the last 12 available months. When a new Article is created, or modified, TBlogApplication.FlushAnyCache will call fDefaultData.Clear to ensure that the updated information will be retrieved from the database on next Default() call.

The above ORM request will generate the following SQL statement:

```
SELECT distinct(PublishedMonth),max(ID)+1 as FirstID FROM Article
 group by PublishedMonth order by PublishedMonth desc limit 12
```

The Default() method will therefore return the following JSON context:

```
{
  "Scope": {
    ...
    "Archives":
    [
      {
        "PublishedMonth": 24178,
        "FirstID": 101
      },
      {
        "PublishedMonth": 24177,
        "FirstID": 100
      },
      ...
```

... which will be processed by the *Mustache* engine.
If you put a breakpoint at the end of this Default() method, and inspect the "Scope" variable, the Delphi debugger will in fact show you in real time the exact JSON content, retrieved from the ORM.

I suspect you just find out how *mORMot*'s ORM/SOA abilites, and JSON / TDocVariant offer amazing means of processing your data. You have the best of both worlds: ORM/SOA gives you fixed structures and strong typing (like in C++/C#/Java), whereas TDocVariant gives you a flexible object scheme, using late-binding to access its content (like in Python/Ruby/JavaScript).

18.2.3. Controller Thread Safety

When run from a TSQLRestServer instance, our *MVC* application commands will be executed by default without any thread protection. When hosted within a TSQLHttpServer instance - see *High-performance http.sys server* (page 271) - several threads may execute the same *Controller* methods at the same time. It is therefore up to your application code to ensure that your TMVCApplication process is thread safe.

Note that by design, all TMVCApplication.RestModel ORM methods are thread-safe.
If your *Controller* business code only uses ORM methods, sending back the information to the *Views*, without storing any data locally, it will be perfectly thread safe.
See for instance the TBlogApplication.AuthorView method we described above.

But consider this method (simplified from the real "*30 - MVC Server*" sample):

```
type
  TBlogApplication = class(TMVCApplication,IBlogApplication)
  protected
    fDefaultArticles: variant;
   ...

procedure TBlogApplication.Default(var Scope: variant);
begin
  if VarIsEmpty(fDefaultArticles) then
```

```
    fDefaultArticles := RestModel.RetrieveDocVariantArray(
      TSQLArticle,'','order by ID desc limit 20',[],ARTICLE_FIELDS);
  _ObjAddProps(['Articles',fDefaultArticles],Scope);
end;
```

In fact, even if this method may sound safe, we have an issue when it is executed by several threads: one thread may be assigning a value to `fDefaultArticles`, whereas another thread may be using the `fDefaultArticles` content. This may result into random runtime errors, very difficult to solve. Even creating a local variable may not be safe, since any access to `fDefaultArticles` should be protected.

A first - and brutal - solution could be to force the `TSQLRestServer` instance to execute all method-based services (including our *MVC* commands) in a giant lock, as stated about *Thread-safety* (page 278):

```
aServer.AcquireExecutionMode[execSOAByMethod] := amLocked; // or amBackgroundThread
```

But this may slow down the whole server process, and reduce its scaling abilities.

You could also lock explictly the *Controller* method, for instance:

```
procedure TBlogApplication.Default(var Scope: variant);
begin
  Locker.ProtectMethod;
  if VarIsEmpty(fDefaultData) then
 ...
```

Using the `TMVCApplication.Locker: IAutoLocker` is a simple and efficient way of protecting your method. In fact, `ProtectMethod` will return an `IUnknown` variable, which will let the compiler create an hidden `try .. finally` block in the method body, to release the lock when it quits. But this locker would be shared by the whole `TMVCApplication` instance, so will be like a giant lock on your *Controller* process.

A more tuned and safe implementation may be to use a `ILockedDocVariant` instead of a plain `TDocVariant` for caching the data. You may therefore write:

```
type
  TBlogApplication = class(TMVCApplication,IBlogApplication)
  protected
    fDefaultData: ILockedDocVariant;
   ...
constructor TBlogApplication.Create(aServer: TSQLRestServer);
begin
  fDefaultData := TLockedDocVariant.Create;
  ...

procedure TBlogApplication.Default(var Scope: variant);
begin
  if not fDefaultData.AddExistingProp('Articles',Scope) then
    fDefaultData.AddNewProp('Articles',RestModel.RetrieveDocVariantArray(
      TSQLArticle,'','order by ID desc limit 20',[],ARTICLE_FIELDS),Scope);
end;
```

Using `ILockedDocVariant` will ensure that only access to this resource will be locked (no giant lock any more), and that slow ORM process (like `RestModel.RetrieveDocVariantArray`) would take place lock-free, to maximize the resource usage.
This is in fact the pattern used by the "*30 - MVC Server*" sample.

18.2.4. Web Sessions

Sessions are usually implemented via cookies, in web sites. A login/logout procedure enhances security of the web application, and User experience can be tuned via small persistence of client-driven data. The `TMVCApplication` class allows creating such sessions.

You can store whatever information you need within the client-side cookie. You can define a `record`, which will be used to store the information as optimized binary, in the browser cache. You can use this cookie information as a cache to the current session, e.g. storing the logged user display name or its rights - avoiding a round trip to the database.
Of course, you should never trust the cookie content (even if our format uses a digital signature via a `crc32` algorithm). But you can use it as a convenient cache, always checking the real data in the database when you are about to perform the action.

For our "*30 - MVC Server*" sample application, we defined the following `record` in `MVCViewModel.pas`:

```
  TCookieData = packed record
    AuthorName: RawUTF8;
    AuthorID: cardinal;
    AuthorRights: TSQLAuthorRights;
  end;
```

This record will be serialized in two ways:

- As raw binary, without the field names, within the cookie, after Base64 encoding and digital signature;
- As a JSON object, with explicit field names, when transmitted to the *Views*.

In order to have proper JSON serialization of the `record`, you would need to specify its structure, if you use a version of Delphi without the new RTII (i.e. before Delphi 2010) - see *Record serialization* (page 245).

Then we can use the `TMVCApplication.CurrentSession` property to perform the authentication:

```
function TBlogApplication.Login(const LogonName, PlainPassword: RawUTF8): TMVCAction;
var Author: TSQLAuthor;
    SessionInfo: TCookieData;
begin
  if CurrentSession.CheckAndRetrieve<>0 then begin
    GotoError(result,HTML_BADREQUEST);
    exit;
  end;
  Author := TSQLAuthor.Create(RestModel,'LogonName=?',[LogonName]);
  try
    if (Author.ID<>0) and Author.CheckPlainPassword(PlainPassword) then begin
      SessionInfo.AuthorName := Author.LogonName;
      SessionInfo.AuthorID := Author.ID;
      SessionInfo.AuthorRights := Author.Rights;
      CurrentSession.Initialize(@SessionInfo,TypeInfo(TCookieData));
      GotoDefault(result);
    end else
      GotoError(result,sErrorInvalidLogin);
  finally
    Author.Free;
  end;
end;
```

As you can see, this `Login()` method will be trigerred from http://localhost:8092/blog/login.. with `LogonName=...&plainpassword=...` parameters. It will first check that there is no current session, retrieve the ORM `Author` corresponding to the `LogonName`, check the supplied password, and set the `SessionInfo: TCookieData` structure with the needed information.
A call to `CurrentSession.Initialize()` will compute the cookie, then prepare to send it to the client browser.

The `Login()` method returns a `TMVCAction` structure. As a consequence, the call to `GotoDefault(result)` will let the `TMVCApplication` processor render the `Default()` method, as if the `/blog/default` URI would have been requested.

When a web page is computed, the following overriden method will be executed:

```
function TBlogApplication.GetViewInfo(MethodIndex: integer): variant;
begin
  result := inherited GetViewInfo(MethodIndex);
  _ObjAddProps(['blog',fBlogMainInfo,
    'session',CurrentSession.CheckAndRetrieveInfo(TypeInfo(TCookieData))],result);
end;
```

It will append the session information from the cookie to the returned *View* data context, as such:

```
{
  "Scope": {
    "articles":
    ...
  },
  "main": {
    "pageName": "Default",
    "blog": {
      "Title": "mORMot BLOG",
      ...
    },
    "session": {
      "AuthorName": "synopse",
      "AuthorID": 1,
      "AuthorRights": {
        "canComment": true,
        "canPost": true,
        "canDelete": true,
        "canAdministrate": true
      },
      "id": 1
    }
  }
}
```

Here, the `session` object will contain the `TCookieData` information, ready to be processed by the *Mustache View*.

When the browser asks for the `/blog/logout` URI, the following method will be executed:

```
function TBlogApplication.Logout: TMVCAction;
begin
  CurrentSession.Finalize;
  GotoDefault(result);
end;
```

The session cookie will then be deleted on the browser side.

18.3. Writing the Views

See *Mustache template engine* (page 410) for a description of how rendering take place in this MVC/MVVM application. You would find the Mustache templates in the "`Views`" sub-folder of the "*30 - MVC Server*" sample application.

You will find some `*.html` files, one per command expecting a `View`, and some `*.partial` files, which are some kind of re-usable sub-templates - we use them to easily compute the page header and footer, and to have a convenient way of gathering some piece of template code, to be re-used in several `*.html` views.

Here is how `Default.html` is defined:

```
{{>header}}
{{>masthead}}
      <div class="blog-header">
```

```
        <h1 class="blog-title">{{main.blog.title}}</h1>
        <p class="lead blog-description">{{main.blog.description}}</p>
      </div>
      <div class="row">
        <div class="col-sm-8 blog-main">
{{#Scope}}
{{>articlerow}}
        {{#lastID}}
        <p><a href="default?scope={{.}}" class="btn btn-primary btn-sm">Previous Articles</a></p>
        {{/lastID}}
        </div>
        <div class="col-sm-3 col-sm-offset-1 blog-sidebar">
          <div class="sidebar-module sidebar-module-inset">
            <h4>About</h4>
            {{{WikiToHtml main.blog.about}}}
          </div>
          <div class="sidebar-module">
            <h4>Archives</h4>
            <ol class="list-unstyled">
              {{#Archives}}
              <li><a href="default?scope={{FirstID}}">{{MonthToText PublishedMonth}}</a></li>
              {{/Archives}}
            </ol>
          </div>
  </div>
   </div>
{{/Scope}}
{{>footer}}
```

The `{{>partials}}` are easily identified, as other `{{...}}` value tags. The main partial is `{{>articlerow}}`, which will display all articles list.

`{{{WikiToHtml main.blog.about}}}` is an *Expression Block* able to render some simple text into proper HTML, using a simple Wiki syntax.

`{{MonthToText PublishedMonth}}` will execute a custom *Expression Block*, defined in our `TBlogApplication`, which will convert the obfuscated `TSQLArticle.PublishedMonth` integer value into the corresponding name and year:

```
procedure TBlogApplication.MonthToText(const Value: variant;
  out result: variant);
const MONTHS: array[0..11] of string = (
  'January','February','March','April','May','June','July','August',
  'September','October','November','December');
var month: integer;
begin
  if VariantToInteger(Value,month) and (month>0) then
    result := MONTHS[month mod 12]+' '+IntToStr(month div 12);
end;
```

The page displaying the `Author` information is in fact quite simple:

```
{{>header}}
{{>masthead}}
      <div class="blog-header">
        <h1 class="blog-title">User {{Author.LogonName}}</h1>
  <div class="lead blog-description">{{Author.FirstName}} {{Author.FamilyName}}
  </div>
      </div>
   <div class="panel panel-default">
   <div class="panel-heading">Information about <strong>{{Author.LogonName}}</strong></div>
   <div class="panel-body">
      {{{TSQLAuthor.HtmlTable Author}}}
   </div>
   </div>
{{>articlerow}}
{{>footer}}
```

It will share the same `{{>partials}}`, for a consistent and maintainable web site design, but in fact

most of the process would take place by the magic of two tags:

- `{{{TSQLAuthor.HtmlTable Author}}}` is an *Expression Block* linked to `TMVCApplication.RestModel` ORM, which will create a HTML table - with the syntax expected by our *BootStrap* CSS - for a `TSQLAuthor` record, identifying the property types and display them as expected (e.g. for dates or time stamps, or for enumerates or sets).
- `{{>articlerow}}` is a partial also shared with `ArticleView.html`, which will render a list of `TSQLArticle` encoded as `{{#Articles}}...{{/Articles}}` sections.

Take a look at the `mORMotMVC.pas` unit: you will discover that every aspect of the MVC process has been divided into small classes, so that the framework is able to create web applications, but also any kind of MVC applications, including mobile or VCL/FMX apps, and/or reporting - using `mORMotReport.pas`.