RESTfull API with Symfony 3.3
and
User Login with API Authentication

Author: @RKtheDragon

Working as Freelance Corporate Trainer for C, C++, Linux, SDLC, Symfony3, Laravel, PHP, MySql, MongoDB, Advance Java, AngularJS, NodeJS

A Freelance WebMaster for Open Source Web Tools, having a total experience of more than 17 years in LAMP and WAMP.

Email: rkg07n@gmail.com

Index of Topics

In this tutorial I am covering User management, and Login Authentication with Guard.

In the next tutorials I will show the implementation of RESTfull API, Cache System, Logging, Creating Custom Commands for CLI, Using CronJobs, Page Sorting, Searching, Filtering, AjaxWorking, using JSON + MySql + MongoDB in one project with RESTfull API and many more using Symfony3.3.

Just follow the steps mentioned in this tutorial, and give at least 2 days yourself to completely understand the Login Authentication System using Guard, I used Guard Authentication instead of FOSUserBundle, cause customized data in our hands is always better. On the other side FOSUserBundle give us only Login, Register New User, and Resetting. Which we can do here with Guard Authentication System. Plus we know what is going on, and can do register users by creating a Register Page and adding some action in Controller. Although I have implemented only Login System with Guard here in this tutorial.

Thank You
@RKtheDragon ☺

Why I have choose Symfony 3.3 version ?

The answer is very simple, Symfony2 is out of date, Symfony4 is newly released, and new comers will run towards latest version Symfony4, whereas old users are still using Symfony2. So basic theory is that who know Symfony2 can easily understand this code, and who are already working with Symfony3, it's a good tutorial for them, and main thing is that this version Symfony3.3 is a bridge between old, existing, and new users.

Happy ☺

Thanks
@RKtheDragon

This is my first Symfony3.3 Tutorial.

On dated: Sunday, February 04, 2018

1.) Installing Symfony 3.3.* with Composer

1.1) Create a directory / folder in your web server root i.e. RestApi3
⇒ mkdir RestApi3
⇒ composer create-project symfony/framework-standard-edition RestApi3 "3.1.*"
⇒ now move to RestApi3 directory to execute following commands:

⇒ composer require friendsofsymfony/rest-bundle
⇒ composer require jms/serializer-bundle
⇒ composer require nelmio/cors-bundle
⇒ composer require nelmio/api-doc-bundle

FOSRestBundle is the backbone of REST API in Symfony. It accepts the client request and returns the appropriate response accordingly. NelmiCorsBundle allows different domains to call our API, and NelmiApiDoc to create API documentation. JMSSerializerBundle helps in serialization of data according to your requested format namely json, xml or yaml.

1.2) Bundles Registration in "AppKernel.php"
⇒ After successfully installing the bundle, you need to register them in AppKernel.php file located in app folder.
⇒ Open the file and add these lines in bundles array.

```
new FOS\RestBundle\FOSRestBundle(),
new Nelmio\CorsBundle\NelmioCorsBundle(),
new Nelmio\ApiDocBundle\NelmioApiDocBundle(),
new JMS\SerializerBundle\JMSSerializerBundle(),
```

1.3) Configuration in "config.yml"

The registration will enable bundles in symfony project. You can now start working with these but before this we need to do a little configuration. We need to enable our API to call different domains for outputting headers. This action needs a little configuration in config.yml file. Open it and add the following code in it:

```
# Start of RESTApi Configuration in "config.yml" by RKG
 # Nelmio CORS
nelmio_cors:
    defaults:
        allow_origin:  ["%cors_allow_origin%"]
        allow_methods: ["POST", "PUT", "GET", "DELETE", "OPTIONS"]
        allow_headers: ["content-type", "authorization"]
        max_age:       3600
    paths:
        '^/': ~

# Nelmio API Doc
nelmio_api_doc:
    sandbox:
        accept_type:        "application/json"
        body_format:
            formats:        [ "json" ]
            default_format: "json"
        request_format:
            formats:
                json:       "application/json"
```

```
# FOSRest Configuration
fos_rest:
    body_listener: true
    format_listener:
        rules:
           # - { path: '^/', priorities: ['json'], fallback_format: json,
prefer_extension: false }
    param_fetcher_listener: true
    view:
        view_response_listener: 'force'
        formats:
            #json: true
        templating_formats:
            html: true
        force_redirects:
            html: true
        failed_validation: HTTP_BAD_REQUEST
        default_engine: twig
```

Open routing.yml to add following

```
NelmioApiDocBundle:
    resource: "@NelmioApiDocBundle/Resources/config/routing.yml"
    prefix:   /api/doc
```

Open security.yml to add following

```
security:
    encoders:
        AppBundle\Entity\User:
            algorithm: bcrypt
            #cost: 12

   # https://symfony.com/doc/current/security.html#b-configuring-how-users-are-loaded
    providers:
        our_db_provider:
            entity:
                class: AppBundle:User
                property: username
        in_memory:
            memory: ~

    firewalls:
        # disables authentication for assets and the profiler, adapt it according to your
needs
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
            # activate different ways to authenticate

            # https://symfony.com/doc/current/security.html#a-configuring-how-your-users-
will-authenticate
            pattern:    ^/
            http_basic: ~
            provider: our_db_provider
```

So we have configured both FOSRestBundle and NelmioCorsBundle. Now we are all set to make our first entity for manipulating data from different calls.

1.4) Now we need some User data so we will create our first entity to the api_user tables, by issuing following command in CLI:

⇒ php bin/console generate:doctrine:entity

This will ask for the name of entity,

--- The Entity Shortcut name: AppBundle:User

--- Configuration format (yml, xml, php or annotation) [annotation]:
--- (Set field names except ID, cause its created automatically as primary key)

⇒ So after creating schema, we will a User class in AppBundle/src/Entity and a UserRepository in AppBundle/src/Repository, automatically generated
⇒ Go to the AppBundle/src/Entity/User.php and open this file to update as below:

```php
<?php

namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

use Symfony\Component\Validator\Constraints as Assert;

/**
 * User
 *
 * @ORM\Table(name="api_user")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UserRepository")
 */
class User implements UserInterface, \Serializable
{
    /**
     * @var int
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string
     *
     * @ORM\Column(name="username", type="string", length=255)
     */
    private $username;


    /**
         * @var string
         *
     * @Assert\NotBlank()
     * @Assert\Length(max=4096)
     * @ORM\Column(name="plainPassword", type="string", length=255)
     */
    private $plainPassword;


    /**
     * @var string
     *
```

```
     * @ORM\Column(name="password", type="string", length=64)
     */
    private $password;

    /**
     * @var string
     *
     * @ORM\Column(name="email", type="string", length=255)
     */
    private $email;

    /**
     * @ORM\Column(type="string", unique=true)
     */
    private $apiKey;

    /**
     * @var array
     *
     * @ORM\Column(type="json_array")
     */
    private $roles = [];

    /**
     * @var bool
     *
     * @ORM\Column(name="isActive", type="boolean")
     */
    private $isActive;

    /**
     * @var \DateTime
     *
     * @ORM\Column(name="createdAt", type="datetime")
     */
    private $createdAt;


    /**
     * Get id
     *
     * @return int
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set username
     *
     * @param string $username
     *
     * @return User
     */
    public function setUsername($username)
    {
        $this->username = $username;

        return $this;
    }

    /**
     * Get username
     *
     * @return string
     */
    public function getUsername()
    {
        return $this->username;
    }
```

```php
/**
 * Get plain password
 *
 * @return string
 */
public function getPlainPassword()
{
    return $this->plainPassword;
}

/**
 * Set plain password
 *
 * @param string $password
 *
 * @return User
 */
public function setPlainPassword($password)
{
    $this->plainPassword = $password;

            return $this;
}

/**
 * Set password
 *
 * @param string $password
 *
 * @return User
 */
public function setPassword($password)
{
    $this->password = $password;

    return $this;
}

/**
 * Get password
 *
 * @return string
 */
public function getPassword()
{
    return $this->password;
}

/**
 * Set email
 *
 * @param string $email
 *
 * @return User
 */
public function setEmail($email)
{
    $this->email = $email;

    return $this;
}

/**
 * Get apiKey
 *
 * @return string
 */
public function getApiKey()
{
```

```
    return $this->apiKey;
}

/**
 * Set apiKey
 *
 * @param string $apiKey
 *
 * @return User
 */
public function setApiKey($apiKey)
{
    $this->apiKey = $apiKey;

    return $this;
}

/**
 * Get email
 *
 * @return string
 */
public function getEmail()
{
    return $this->email;
}


/**
 * Returns the roles or permissions granted to the user for security.
 */
public function getRoles(): array
{
    $roles = $this->roles;

    // guarantees that a user always has at least one role for security
    if (empty($roles)) {
        $roles[] = 'ROLE_USER';
    }

    return array_unique($roles);
}

public function setRoles(array $roles): void
{
    $this->roles = $roles;
}

/**
 * Set isActive
 *
 * @param boolean $isActive
 *
 * @return User
 */
public function setIsActive($isActive)
{
    $this->isActive = $isActive;

    return $this;
}

/**
 * Get isActive
 *
 * @return bool
 */
public function getIsActive()
{
    return $this->isActive;
}
```

```php
/**
 * Set createdAt
 *
 * @param \DateTime $createdAt
 *
 * @return User
 */
public function setCreatedAt($createdAt)
{
    $this->createdAt = $createdAt;

    return $this;
}

/**
 * Get createdAt
 *
 * @return \DateTime
 */
public function getCreatedAt()
{
    return $this->createdAt;
}

/**
 * Returns the salt that was originally used to encode the password.
 *
 * {@inheritdoc}
 */
public function getSalt(): ?string
{
    // See "Do you need to use a Salt?" at
    //        https://symfony.com/doc/current/cookbook/security/entity_provider.html
    // we're using bcrypt in security.yml to encode the password, so
    // the salt value is built-in and you don't have to generate one

    return null;
}

/**
 * Removes sensitive data from the user.
 *
 * {@inheritdoc}
 */
public function eraseCredentials(): void
{
    // if you had a plainPassword property, you'd nullify it here
    // $this->plainPassword = null;
}

/**
 * {@inheritdoc}
 */
public function serialize(): string
{
    return serialize([
        $this->id,
        $this->username,
        $this->password,
        // see section on salt below
        // $this->salt,
    ]);
}

/**
 * {@inheritdoc}
 */
public function unserialize($serialized): void
{
    list(
```

```
            $this->id,
            $this->username,
            $this->password,
            // see section on salt below
            // $this->salt
        ) = unserialize($serialized);
    }
}
```

1.5) After updated User.php, we need to update parameters.yml to configure database settings:

⇒ Open the parameters.yml file and update for database name, database user, and database password, so we can create and update our user tables in the database.

⇒ Also see on the top of the User.php Class that we have mentioned user table name as api_user, and not only user, so the command will create api_user table in the database having User.php Class.

⇒ Run the following command to update schema:

⇒ php bin/console doctrine:schema:update –force

⇒ The above command will create api_user, empty datatable in the database

Now insert a row in the datatable issuing follwing SQL statement in phpmyadmin:

```
INSERT INTO `api_user` (`id`, `username`, `plainPassword`, `password`,
`email`, `api_key`, `roles`, `isActive`, `createdAt`) VALUES (NULL, 'rkg03',
'mypass', 'mypass', 'rkg07n@gmail.com', '123456790',
'a:0:{i:0;s:10:\"ROLE_ADMIN\";}', '1', '2018-02-01 14:35:27');
```

So, for now one of your user for the RestApi3 has been created, having role "ROLE_ADMIN" and Active (Enabled).

1.6) Creating User GUI with CRUD CLI Command

To create user front end we will use the CRUD (Create, Read, Update, Delete) CLI command as below, so it will create UserController.php with appropriate Forms and Twig templates, we only need to modify the GUI by adding out stylesheet i.e. Bootstrap CSS or any other.

The command to generate CRUD as below:

```
php bin/console generate:doctrine:crud --entity=AppBundle:User --format=annotation
--with-write --no-interaction
```

Now we will create a new file in "app/resources/views" folder as "_base.html.twig" and place the following code inside this new file:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>{% block title %}Welcome!{% endblock %}</title>
    {% block stylesheets %}
        <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
```

```
  <link rel="stylesheet" href="{{ asset('bundles/webLib/my/css/style.css') }}">
        <style type="text/css">
               header{
                           text-align: right;
                           padding: 0.5rem;
                           border-bottom: 0.1rem solid black;
                           background-color: #cacfdd;
                     }
              </style>
    {% endblock %}

    {% block javascripts %}
       <script
       src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
       ></script>
        <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.1/jquery.min.js"></scrip
t>

     {% endblock %}
        <link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}"
/>
    </head>

    <body>
{# Start of Comment.. Remove this comment after login development

     {% set appUser = (app.user.username) ?? app.user.username ?? null %}
       {% if appUser is null %}
          {% if  app.request.getRequestUri()  !=  path('security_login')  %}
<p>
       you are not logged in: <a href="{{ path('security_login') }}">login</a>
</p>
       {% endif %}
{% else %}
              <header>
                     You are logged in as: {{ appUser }}
                     <br>
                     <a href="{{ path('logout') }}">logout</a>
              </header>
{% endif %}

End of comment .. remove this line after login development #}

{% block body %}

{% endblock %}

    </body>
</html>
```

Now we have mentioned all necessary things in our “_base.html.twig” file, so we have to change in other files also where we are calling “base.html.twig”, there we only need to add an underscore in front of those files, and those files we can find in “app/Resources/views/user” directory, as “edit.html.twig, index.html.twig, new.html.twig and show.html.twig” files.
On the top of each file you have to replace “`{% extends 'base.html.twig' %}`” with this “”`{% extends '_base.html.twig' %}`”, now we can call our custom file to include our Stylesheets and Javascripts in all other files.
Now we will update our “index.html.twig” of User folder, to give it a good looking GUI. As below, just replace the code with the following it this file:

```
{% extends '_base.html.twig' %}

{% block title %}User List{% endblock %}

{% block body %}
    <h1 style="margin-left:5%;">Users list</h1>

    <ul style="list-style: none; text-align:right; margin-right:5%;">
        <li>
            <a href="{{ path('user_new') }}" class="btn btn-primary btn-
xs">Create a new user</a>
        </li>
    </ul>

    <table class="table table-striped table-bordered dataTable no-footer"
style="margin-left:5%; width:90%;">
        <thead>
            <tr>
                <th>Count</th>
                <th>Username</th>
                <th>Password</th>
                <th>Email</th>
                <th>Apikey</th>
                <th>Roles</th>
                <th>Isactive</th>
                <th>Createdat</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>

        {% for user in users %}
            <tr>
              <td>{{ loop.index|e }}</td>
                <td>{{ user.username }}</td>
                <td>{{ user.password }}</td>
                <td>{{ user.email }}</td>
                <td>{{ user.apiKey }}</td>
                <td>{{ user.roles|join(', ') }}</td>
                <td>{% if user.isActive %}Yes{% else %}No{% endif %}</td>
                <td>{% if user.createdAt %}{{ user.createdAt|date('Y-m-d
H:i:s') }}{% endif %}</td>
                <td>
                    <ul style="list-style: none;">
                        <li>
                            <a href="{{ path('user_show', { 'id': user.id })
}}" class="btn btn-primary btn-xs">View</a>
                        </li>
                        <li>
                            <a href="{{ path('user_edit', { 'id': user.id })
}}" class="btn btn-default btn-xs">Edit</a>
                        </li>
                    </ul>
                </td>
            </tr>

        {% endfor %}
        </tbody>
    </table>


{% endblock %}
```

You can see now even, we have not tested any of the file in our http://localhost/RestApi3 why, we have not added our appropriate code to run the application. By the way, without modifying the code for base.html.twig and index.html.twig to may test the app, it will run.
After modification of code we have to insert the appropriate path (routing) for /login which is security_login and calling in base.html.twig for checking if a user is logged in or not. After development of section "2" which is "Login Authentication with Guard" remove the commented lines from _base.html.twig for working of checking the user login.

Although, I did not make any change in UserController.php which is generated by CRUD.

So far, we have created a user section with a good looking GUI and CRUD templates and their working. Now put the following code in "AppBundle/Form/UserType.php" file:

```
public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('username')
                    ->add('password', PasswordType::class)
                    ->add('email')
                    ->add('apiKey')

                ->add('roles', ChoiceType::class, [
                'multiple' => true,
                'expanded' => false, // if true, render check-boxes
                'choices' => [
                    'None'    => 'None',
                    'Admin'   => 'ROLE_ADMIN',
                    'Manager' => 'ROLE_MANAGER',
                    'User'    => 'ROLE_USER',
                ],
            ])
                    ->add('isActive')
                    ->add('createdAt');
    }
```

This method of UserType class, will show the USER ROLEs in user-friendly manner. If we will not do it then there will be error,

An exception has been thrown during the rendering of a template ("Notice: Array to string conversion").

And we have to add two components also to use the password and choice type html elements:

```
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
use Symfony\Component\Form\Extension\Core\Type\ChoiceType;
```

Wallah !! That's great… a working app for user is here !!!

Lets now add the Login authentication system for User so only logged in users can view the User Module.

2.) Creating Login System with Guard

Here, we will add the SecurityController.php in AppBundle/Controller/ directory, so this class will handle the user login and redirect the user to appropriate module route according to USER ROLE. Here is the code for SecurityController.php, a single method “loginAction()”, having the name “security_login” and route “login”:

```
<?php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\RedirectResponse;

use AppBundle\Form\LoginForm;

use AppBundle\Entity\User;

class SecurityController extends Controller
{
    /**
     * @Route("/login", name="security_login")
     */
    public function loginAction()
    {
        $auth_checker = $this->get('security.authorization_checker');
        $router = $this->get('router');

        // 307: Internal Redirect
        if ($auth_checker->isGranted(['ROLE_ADMIN'])) {
            // SUPER_ADMIN roles go to the `admin_home` route
            return new RedirectResponse($router->generate('user_index'), 307);
        }

        if ($auth_checker->isGranted('ROLE_USER')) {
            // Everyone else goes to the `home` route
            return new RedirectResponse($router->generate('user_index'), 307);
        }

        $authenticationUtils = $this->get('security.authentication_utils');

        // get the login error if there is one
        $error = $authenticationUtils->getLastAuthenticationError();

        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

               $user = new User();
               $form = $this->createForm(LoginForm::class, $user);

        //$form = $this->createForm(LoginForm::class, [
        //    '_username' => $lastUsername,
        //]);

        return $this->render(
            'security/login.html.twig',
            array(
                'form' => $form->createView(),
                'error' => $error,
            )
        );
    }
}
```

After adding this we will create a new Form, LoginForm.php in AppBundle/Form/ directory. The code for LoginForm.php is as below:

```
<?php

namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

use Symfony\Component\Form\Extension\Core\Type\PasswordType;


class LoginForm extends AbstractType
{
    /**
     * {@inheritdoc}
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {

        $builder->add('_username')
                ->add('_password', PasswordType::class);

    }

    /**
     * {@inheritdoc}
     */
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\User'
        ));
    }

    /**
     * {@inheritdoc}
     */
    public function getBlockPrefix()
    {
        //return 'appbundle_user';
        return ;
    }
}
```

Now, update the security.yml of app/config/ directory, as below:

```
security:
    encoders:
        AppBundle\Entity\User:
            algorithm: bcrypt
            #cost: 12

    # http://symfony.com/doc/current/security.html#hierarchical-roles
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
    firewalls:
        # disables authentication for assets and the profiler, adapt it according to your
needs
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
          main:
            anonymous: ~
            pattern:    ^/
            http_basic: ~
```

```
    provider: our_db_provider

    form_login:
         login_path: security_login
         #check_login: security_login
         csrf_token_generator: security.csrf.token_manager

    logout:
         path:   /logout
         target: /login

    guard:
        authenticators:
    -       AppBundle\Security\LoginFormAuthenticator
```

While doing copy and paste for YAML files, please make sure you are giving an indent of 4 spaces to each sub part, else Symfony can not read the YAML file and raise an error. Here with the Guard, we are generating CSRF_Token as well, for our custom login templates.

As you can see in security.yml, that we have added "Guard➔Authenticators➔LoginFormAuthenticator" which indicates now we have to create LoginFormAuthenticator.php in AppBundle/Security/ directory, which is a prominent part of User Login Security system. If we will not add this class in security.yml, the login system will not work.

The class `class LoginFormAuthenticator extends AbstractFormLoginAuthenticator` has mainly six(6) methods including __construct(). The working of each method is as below:

Below part is taken from Symfony 3.3 docs:
http://symfony.com/doc/3.3/security/guard_authentication.html

The Guard Authenticator Methods

Each authenticator needs the following methods:

01.) **getCredentials(Request $request)**
This will be called on *every* request and your job is to read the token (or whatever your "authentication" information is) from the request and return it. If you return `null`, the rest of the authentication process is skipped. Otherwise, `getUser()` will be called and the return value is passed as the first argument.

02.) **getUser($credentials, UserProviderInterface $userProvider)**
If `getCredentials()` returns a non-null value, then this method is called and its return value is passed here as the `$credentials` argument. Your job is to return an object that implements `UserInterface`. If you do, then `checkCredentials()` will be called. If you return `null` (or throw an AuthenticationException) authentication will fail.

03.) **checkCredentials($credentials, UserInterface $user)**
If `getUser()` returns a User object, this method is called. Your job is to verify if the credentials are correct. For a login form, this is where you would check that the password is correct for the user. To pass authentication, return `true`. If you

return *anything* else (or throw an AuthenticationException), authentication will fail.

04.) **onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)**
This is called after successful authentication and your job is to either return a `Response` object that will be sent to the client or `null` to continue the request (e.g. allow the route/controller to be called like normal). Since this is an API where each request authenticates itself, you want to return `null`.

05.) **onAuthenticationFailure(Request $request, AuthenticationException $exception)**
This is called if authentication fails. Your job is to return the `Response` object that should be sent to the client. The `$exception` will tell you *what* went wrong during authentication.

06.) **start(Request $request, AuthenticationException $authException = null)**
This is called if the client accesses a URI/resource that requires authentication, but no authentication details were sent (i.e. you returned `null` from `getCredentials()`). Your job is to return a `Response` object that helps the user authenticate (e.g. a 401 response that says "token is missing!").

07.) **supportsRememberMe()**
If you want to support "remember me" functionality, return true from this method. You will still need to active `remember_me` under your firewall for it to work. Since this is a stateless API, you do not want to support "remember me" functionality in this example.

08.) **createAuthenticatedToken(UserInterface $user, string $providerKey)**
If you are implementing the `GuardAuthenticatorInterface` instead of extending the `AbstractGuardAuthenticator` class, you have to implement this method. It will be called after a successful authentication to create and return the token for the user, who was supplied as the first argument.

So far, we have learned Guard Authentication Methods. Now, I will show you code for login.html.twig, which is a customized template.

```
{% extends '_base.html.twig' %}

{% block title %}Login!{% endblock %}
{% block body %}
<div class="container">
    <div class="row">
        <div class="col-xs-12">
            <h1>Login!</h1>

            {% if error %}
                <div class="alert alert-danger">
                    {{ error.messageKey|trans(error.messageData, 'security') }}
                </div>
            {% endif %}
```

```
            {{ form_start(form) }}
                <input type="hidden" name="_csrf_token" value="{{
csrf_token('authenticate') }}">
                {{ form_row(form._username) }}
                {{ form_row(form._password) }}
                <button type="submit" class="btn btn-success">Login <span class="fa fa-
lock"></span></button>
            {{ form_end(form) }}
        </div>
    </div>
</div>
{% endblock %}
```

Here I left one mistake, you have to find out and correct yourself. Have you found, so go to view source of the Login html page. Oh yeah !!! you can see two CSRF tokens there. So how to remove one which is auto generated by Form Builder. You know, if not then totally customize the form and remove form_start and form_end, instead put there HTML Form tags and problem solved.
In this twig template you may see the value for CSRF token as {{ csrf_token('authenticate') }}, this is coming form getCredentials() method of LoginFormAuthenticator.php, where as I have mentioned _username and _password as keys of retVal array which are return values of username and password of the authenticated user.

Now we have to update app/config/routing.yml, by adding following lines:

```
logout:
    path: /logout
```

Let's now login to the app, with this URL: http://localhost/RestApi3/login

We have created at least one user in the database with username "rkg03" and password "mypass", although I have not encoded the password, even while checking credentials in method checkCredentials() of LoginFormAuthenticator.php, mentioned `if (!$this->passwordEncoder->isPasswordValid($user, $password))` If password NOT encoded, here we will remove the NOT (!) sign after including the encoder for password.
So let's login first. Here we logged in to the system with given credentials without any problem, now go to _base.html.twig and delete the lines with comments, and refresh the page, you may seed the header section showing you a welcome note with Logout Link.

Let's Logout from the system, we have redirected to Login page, here we will type in the addressbar as http://localhost/RestApi3/user and what happened !!! The page is showing, even we have not logged in !!! Cause we have not protected the USER module. To protect this module we may use at least one of these two methods:

01.) Add `access_control:` in app/config/security.yml, as below

⇒ - { path: ^/user/, role: ROLE_USER }

02.) Just open UserController.php, and put this line before starting the class as below:

```
/**
 * User controller.
 * @Security("has_role('ROLE_USER')")
 * @Route("user")
 */
class UserController extends Controller
```

Here @Security("has_role('ROLE_USER')"), restricts an unauthorized user to view any part of UserController without proper authentication via login form. Here a user need at least a

ROLE as ROLE_USER or above as I have mentioned in role_heirarchy of app/config/security.yml, here ROLE_USER is least entity and all other have greater rights than ROLE_USER.

In this very first part of RestApi3, I have given you a short tutorial using Symfony 3.3, about User Class, User Management and Login Authentication System.

One of issues you may think that the app name is RestApi3, where as we have not touched any part of RESTfull API so far except configuration in app/config/config.yml !!!

The reason behind this is that I have not started any product coding or module for that, in the next part I will introduce the RestApi3 with all its features and other features of Symfony.

Oh yeah !!