## UNDERSTANDING THE BASICS

### What is a bearer token?

This refers to a JWT, which is passed along via the HTTP header called Authorization, in the string format "Bearer $your\_token\_here".

### What is JWT?

JWT stands for JSON Web Token, a common authentication tactic used in modern web apps.

### How does JSON work?

JSON is simply a data format that closely resembles the data literal format allowed by JavaScript. It's a hierarchical format allowing nested objects and arrays, as well as string and number literals.

## What is a RESTful API?

An Application Programming Interface (API) is a set of clearly defined methods of communication between various software components. A good API makes it easier to develop a computer program by providing all the building blocks. While the specifications vary between various APIs, the end goal is to provide value to the programmer through utilization of the services gained from using an API.

REST (or REpresentational State Transfer) is an architectural style that evolved as Fielding wrote the HTTP/1.1 and URI specs and has proven to be well-suited for developing distributed hypermedia applications. While REST is more widely applicable, it is most commonly used within the context of communicating with services via HTTP.

## REST API Authentication Types Overview

RESTful applications rely on the underlying security of the API ecosystem rather than including security within the REST architecture style. In addition to securing RESTful API calls with the HTTPS protocol, session-based authentication should be utilized. Currently, most RESTful applications leverage OAuth 2.0 and JWT is the newcomer that is gaining more and more popularity with API developers.

OAuth, JWT, and Basic Auth all use headers for transmitting credentials, and API providers should be doing the same with all API keys. While easy to do as parameters, they are more secure as headers.

#-🡪With the rising popularity of single page applications, mobile applications, and RESTful API services, the way [web developers](https://www.toptal.com/web) write back-end code has changed significantly. With technologies like AngularJS and BackboneJS, we are no longer spending much time building markup, instead we are building APIs that our front-end applications consume. Our back-end is more about business logic and data, while presentation logic is moved exclusively to the front-end or mobile applications. These changes have led to new ways of implementing authentication in modern applications.

Authentication is one of the most important parts of any web application. For decades, cookies and server-based authentication were the easiest solution. However, handling authentication in modern Mobile and Single Page Applications can be tricky, and demand a better approach. The best known solutions to authentication problems for APIs are the [OAuth 2.0](https://tools.ietf.org/html/rfc6749) and the [JSON Web Token](https://jwt.io/) (JWT).

Before we get into this JSON Web Token tutorial, what exactly is a JWT?

**What is a JSON Web Token?**

A JSON Web Token is used to send information that can be verified and trusted by means of a digital signature. It comprises a compact and URL-safe JSON object, which is cryptographically signed to verify its authenticity, and which can also be encrypted if the payload contains sensitive information.

Because of its compact structure, JWT is usually used in HTTP Authorization headers or URL query parameters.

**Structure of a JSON Web Token**

A JWT is represented as a sequence of [base64url](https://en.wikipedia.org/wiki/Base64) encoded values that are separated by period characters.



Here is a JWT token example:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.

eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2FkbWluIjp0cnVlLCJjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0.

yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK\_ZXw

**Header**

The header contains the metadata for the token and it minimally contains the type of signature and the encryption algorithm. (You can use a [JSON formatter](https://www.toptal.com/developers/json-formatter) tool to prettify the JSON object.)

**Example Header**

{

"alg": "HS256",

"typ": "JWT"

}

This JWT example header declares that the encoded object is a JSON Web Token, and that it is signed using the HMAC SHA-256 algorithm.

Once this is base64 encoded, we have the first part of our JWT.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

**Payload (Claims)**

In the context of JWT, a claim can be defined as a statement about an entity (typically, the user), as well as additional metadata about the token itself. The claim contains the information we want to transmit, and that the server can use to properly handle JSON Web Token authentication. There are multiple claims we can provide; these include registered claim names, public claim names and private claim names.

**Registered JWT Claims**

These are the claims that are registered in the [IANA JSON Web Token Claims registry](https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32#section-10.1). These JWT claims are not intended to be mandatory but rather to provide a starting point for a set of useful, interoperable claims.

These include:

* **iss**: The issuer of the token
* **sub**: The subject of the token
* **aud**: The audience of the token
* **exp**: JWT expiration time defined in Unix time
* **nbf**: “Not before” time that identifies the time before which the JWT must not be accepted for processing
* **iat**: “Issued at” time, in Unix time, at which the token was issued
* **jti**: JWT ID claim provides a unique identifier for the JWT

**Public Claims**

Public claims need to have collision-resistant names. By making the name a URI or URN, naming collisions are avoided for JWTs where the sender and receiver are not part of a closed network.

An example of a public claim name could be: https://www.toptal.com/jwt\_claims/is\_admin, and the best practice is to place a file at that location describing the claim so that it can be dereferenced for documentation.

**Private Claims**

Private claim-names may be used in places where JWTs are only exchanged in a closed environment between known systems, such as inside an enterprise. These are claims that we can define ourselves, like user IDs, user roles, or any other information.

Using claim-names that might have conflicting semantic meanings outside of a closed or private system are subject to collision, so use them with caution.

It is important to note that we want to keep a web token as small as possible, so use only necessary data inside public and private claims.

**JWT Example Payload**

{

"iss": "toptal.com",

"exp": 1426420800,

"https://www.toptal.com/jwt\_claims/is\_admin": true,

"company": "Toptal",

"awesome": true

}

This example payload has two registered claims, one public claim and two private claims. Once it is base64 encoded, we have the second part of our JWT.

eyJpc3MiOiJ0b3B0YWwuY29tIiwiZXhwIjoxNDI2NDIwODAwLCJodHRwOi8vdG9wdGFsLmNvbS9qd3RfY2xhaW1zL2lzX2FkbWluIjp0cnVlLCJjb21wYW55IjoiVG9wdGFsIiwiYXdlc29tZSI6dHJ1ZX0

**Signature**

The JWT standard follows the JSON Web Signature (JWS) specification to generate the final signed token. It is generated by combining the encoded JWT Header and the encoded JWT Payload, and signing it using a strong encryption algorithm, such as HMAC SHA-256. The signature’s secret key is held by the server so it will be able to verify existing tokens and sign new ones.

$encodedContent = base64UrlEncode(header) + "." + base64UrlEncode(payload);

$signature = hashHmacSHA256($encodedContent);

This gives us the final part of our JWT.

yRQYnWzskCZUxPwaQupWkiUzKELZ49eM7oWxAQK\_ZXw

**JWT Security and Encryption**

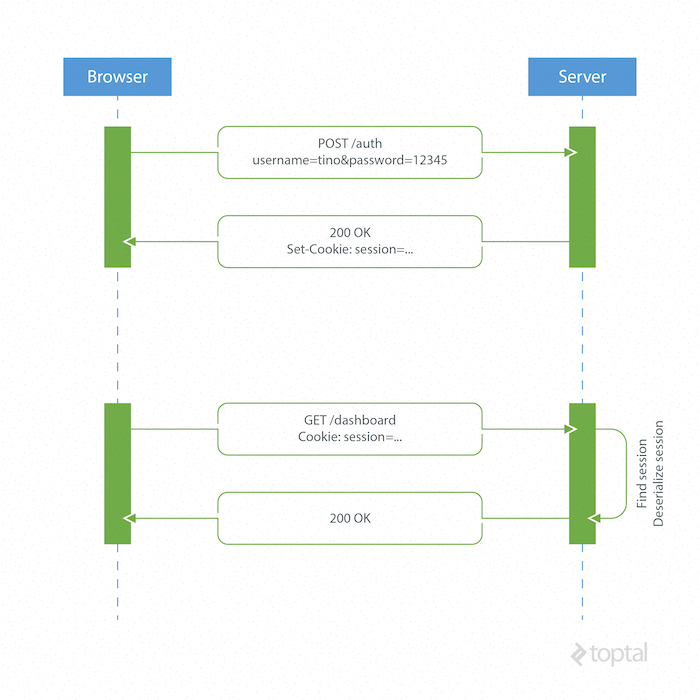
It is critical to use TLS/SSL in conjunction with JWT, to prevent man-in-the-middle attacks. In most cases, this will be sufficient to encrypt the JWT payload if it contains sensitive information. However, if we want to add an additional layer of protection, we can encrypt the JWT payload itself using the [JSON Web Encryption](https://tools.ietf.org/html/draft-ietf-jose-json-web-encryption-40) (JWE) specification.

Of course, if we want to avoid the additional overhead of using JWE, another option is to simply keep sensitive information in our database, and use our token for additional API calls to the server whenever we need to access sensitive data.

**Why the Need for Web Tokens?**

Before we can see all the benefits of using JWT authentication, we have to look at the way authentication has been done in the past.

**Server-Based Authentication**



Because the HTTP protocol is stateless, there needs to be a mechanism for storing user information and a way to authenticate the user on every subsequent request after login. Most websites use cookies for storing user’s session ID.

**How it Works**

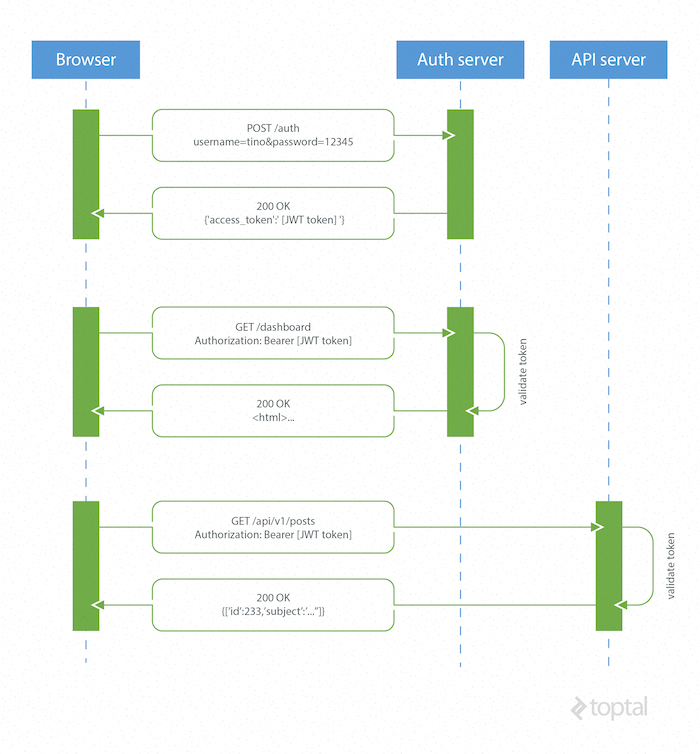
The browser makes a POST request to the server that contains the user’s identification and password. The server responds with a cookie, which is set on the user’s browser, and includes a session ID to identify the user.

On every subsequent request, the server needs to find that session and deserialize it, because user data is stored on the server.

**Drawbacks of Server-Based Authentication**

* **Hard to scale**: The server needs to create a session for a user and persist it somewhere on the server. This can be done in memory or in a database. If we have a distributed system, we have to make sure that we use a separate session storage that is not coupled to the application server.
* **Cross-origin request sharing (CORS)**: When using AJAX calls to fetch a resource from another domain (“cross-origin”) we could run into problems with forbidden requests because, by default, HTTP requests don’t include cookies on cross-origin requests.
* **Coupling with the web framework**: When using server-based authentication we are tied to our framework’s authentication scheme. It is really hard, or even impossible, to share session data between different web frameworks written in different programming languages.

**Token-Based Authentication**



Token based/JWT authentication is stateless, so there is no need to store user information in the session. This gives us the ability to scale our application without worrying where the user has logged in. We can easily use the same token for fetching a secure resource from a domain other than the one we are logged in to.

**How JSON Web Tokens Work**

A browser or mobile client makes a request to the authentication server containing user login information. The authentication server generates a new JWT access token and returns it to the client. On every request to a restricted resource, the client sends the access token in the query string or Authorization header. The server then validates the token and, if it’s valid, returns the secure resource to the client.

The authentication server can sign the token using any secure signature method. For example, a symmetric key algorithm such as HMAC SHA-256 can be used if there is a secure channel to share the secret key among all parties. Alternatively, an asymmetric, public-key system, such as RSA, can be used as well, eliminating the need for further key-sharing.

Top of Form

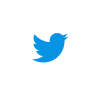
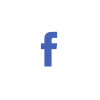
**Like what you're reading?**

*Get the latest updates first.*





No spam. Just great articles & insights.

* 
* 

Bottom of Form

**Advantages of Token-Based Authentication**

**Stateless, easier to scale**: The token contains all the information to identify the user, eliminating the need for the session state. If we use a load balancer, we can pass the user to any server, instead of being bound to the same server we logged in on.

**Reusability**: We can have many separate servers, running on multiple platforms and domains, reusing the same token for authenticating the user. It is easy to build an application that shares permissions with another application.

**JWT Security**: Since we are not using cookies, we don’t have to protect against cross-site request forgery (CSRF) attacks. We should still encrypt our tokens using JWE if we have to put any sensitive information in them, and transmit our tokens over HTTPS to prevent man-in-the-middle attacks.

**Performance**: There is no server side lookup to find and deserialize the session on each request. The only thing we have to do is calculate the HMAC SHA-256 to validate the token and parse its content.

**A JSON Web Token Example using Laravel 5 and AngularJS**

In this JWT tutorial I am going to demonstrate how to implement the basic authentication using JSON Web Tokens in two popular web technologies: Laravel 5 for the backend code and AngularJS for the frontend Single Page Application (SPA) example. (You can find the entire demo [here](https://laravel-angular-jwt.herokuapp.com/), and the source code in [this GitHub repository](https://github.com/ttkalec/laravel5-angular-jwt) so that you can follow along with the tutorial.)

This JSON web token example will not use any kind of encryption to ensure the confidentiality of the information transmitted in the claims. In practice this is often okay, because TLS/SSL encrypts the request. However, if the token is going to contain sensitive information, such as the user’s social security number, it should also be encrypted using JWE.

**Laravel Backend Example**

We will use Laravel to handle user registration, persisting user data to a database and providing some restricted data that needs authentication for the Angular app to consume. We will create an example API subdomain to simulate Cross-origin resource sharing (CORS) as well.

**Installation and Project Bootstrapping**

In order to use Laravel, we have to install the [Composer](https://getcomposer.org/) package manager on our machine. When developing in Laravel I recommend using the Laravel Homestead pre-packaged “box” of Vagrant. It provides us with a complete development environment regardless of our operating system.

The easiest way to bootstrap our JWT Laravel application is to use a Composer package Laravel Installer.

composer global require "laravel/installer=~1.1"

Now we are all ready to create a new Laravel project by running laravel new jwt.

For any questions about this process please refer to the official [Laravel documentation](https://laravel.com/docs/5.0).

After we have created the basic Laravel 5 application, we need to set up our Homestead.yaml, which will configure folder mappings and domains configuration for our local environment.

Example of a Homestead.yaml file:

---

ip: "192.168.10.10"

memory: 2048

cpus: 1

authorize: /Users/ttkalec/.ssh/public.psk

keys:

- /Users/ttkalec/.ssh/private.ppk

folders:

- map: /coding/jwt

to: /home/vagrant/coding/jwt

sites:

- map: jwt.dev

to: /home/vagrant/coding/jwt/public

- map: api.jwt.dev

to: /home/vagrant/coding/jwt/public

variables:

- key: APP\_ENV

value: local

After we’ve booted up our Vagrant box with the vagrant up command and logged into it using vagrant ssh, we navigate to the previously defined project directory. In the example above this would be /home/vagrant/coding/jwt. We can now run php artisan migrate command in order to create the necessary user tables in our database.

**Installing Composer Dependencies**

Fortunately, there is a community of developers working on Laravel and maintaining many great packages that we can reuse and extend our application with. In this example we will use [tymon/jwt-auth](https://packagist.org/packages/tymon/jwt-auth), by Sean Tymon, for handling tokens on the server side, and [barryvdh/laravel-cors](https://packagist.org/packages/barryvdh/laravel-cors), by Barry vd. Heuvel, for handling CORS.

**jwt-auth**

Require the tymon/jwt-auth package in our composer.json and update our dependencies.

composer require tymon/jwt-auth 0.5.\*

Add the JWTAuthServiceProvider to our app/config/app.php providers array.

'Tymon\JWTAuth\Providers\JWTAuthServiceProvider'

Next, in app/config/app.php file, under the aliases array, we add the JWTAuth facade.

'JWTAuth' => 'Tymon\JWTAuth\Facades\JWTAuth'

Finally, we will want to publish the package config using the following command: php artisan config:publish tymon/jwt-auth

JSON Web tokens are encrypted using a secret key. We can generate that key using the php artisan jwt:generate command. It will be placed inside our config/jwt.php file. In the production environment, however, we never want to have our passwords or API keys inside configuration files. Instead, we should place them inside server environment variables and reference them in the configuration file with the env function. For example:

'secret' => env('JWT\_SECRET')

We can find out more about this package and all of it’s config settings [on Github](https://github.com/tymondesigns/jwt-auth).

**laravel-cors**

Require the barryvdh/laravel-cors package in our composer.json and update our dependencies.

composer require barryvdh/laravel-cors 0.4.x@dev

Add the CorsServiceProvider to our app/config/app.php providers array.

'Barryvdh\Cors\CorsServiceProvider'

Then add the middleware to our app/Http/Kernel.php.

'Barryvdh\Cors\Middleware\HandleCors'

Publish the configuration to a local config/cors.php file by using the php artisan vendor:publish command.

Example of a cors.php file configuration:

return [

'defaults' => [

'supportsCredentials' => false,

'allowedOrigins' => [],

'allowedHeaders' => [],

'allowedMethods' => [],

'exposedHeaders' => [],

'maxAge' => 0,

'hosts' => [],

],

'paths' => [

'v1/\*' => [

'allowedOrigins' => ['\*'],

'allowedHeaders' => ['\*'],

'allowedMethods' => ['\*'],

'maxAge' => 3600,

],

],

];

**Routing and Handling HTTP Requests**

For the sake of brevity, I will put all my code inside the routes.php file that is responsible for Laravel routing and delegating requests to controllers. We would usually create dedicated controllers for handling all our HTTP requests and keep our code modular and clean.

We will load our AngularJS SPA view using

Route::get('/', function () {

return view('spa');

});

**User Registration**

When we make a POST request to /signup with a username and password, we will try to create a new user and save it to the database. After the user has been created, a JWT is created and returned via JSON response.

Route::post('/signup', function () {

$credentials = Input::only('email', 'password');

try {

$user = User::create($credentials);

} catch (Exception $e) {

return Response::json(['error' => 'User already exists.'], HttpResponse::HTTP\_CONFLICT);

}

$token = JWTAuth::fromUser($user);

return Response::json(compact('token'));

});

**User Sign In**

When we make a POST request to /signin with a username and password, we verify that the user exists and returns a JWT via the JSON response.

Route::post('/signin', function () {

$credentials = Input::only('email', 'password');

if ( ! $token = JWTAuth::attempt($credentials)) {

return Response::json(false, HttpResponse::HTTP\_UNAUTHORIZED);

}

return Response::json(compact('token'));

});

**Fetching a Restricted Resource on the Same Domain**

Once the user is signed in, we can fetch the restricted resource. I’ve created a route /restricted that simulates a resource that needs an authenticated user. In order to do this, the request Authorization header or query string needs to provide the JWT for the backend to verify.

Route::get('/restricted', [

'before' => 'jwt-auth',

function () {

$token = JWTAuth::getToken();

$user = JWTAuth::toUser($token);

return Response::json([

'data' => [

'email' => $user->email,

'registered\_at' => $user->created\_at->toDateTimeString()

]

]);

}

]);

In this example, I’m using the jwt-auth middleware provided in the jwt-auth package using 'before' => 'jwt-auth'. This middleware is used to filter the request and validate the JWT token. If the token is invalid, not present, or expired, the middleware will throw an exception that we can catch.

In Laravel 5, we can catch exceptions using the app/Exceptions/Handler.php file. Using the render function we can create HTTP responses based on the thrown exception.

public function render($request, Exception $e)

{

if ($e instanceof \Tymon\JWTAuth\Exceptions\TokenInvalidException)

{

return response(['Token is invalid'], 401);

}

if ($e instanceof \Tymon\JWTAuth\Exceptions\TokenExpiredException)

{

return response(['Token has expired'], 401);

}

return parent::render($request, $e);

}

If the user is authenticated and the token is valid, we can safely return the restricted data to the frontend via JSON.

**Fetching Restricted Resources from the API Subdomain**

In the next JSON web token example, we’ll take a different approach for token validation. Instead of using jwt-auth middleware, we will handle exceptions manually. When we make a POST request to an API server api.jwt.dev/v1/restricted, we are making a cross-origin request, and have to enable CORS on the backend. Fortunately, we have already configured CORS in the config/cors.php file.

Route::group(['domain' => 'api.jwt.dev', 'prefix' => 'v1'], function () {

Route::get('/restricted', function () {

try {

JWTAuth::parseToken()->toUser();

} catch (Exception $e) {

return Response::json(['error' => $e->getMessage()], HttpResponse::HTTP\_UNAUTHORIZED);

}

return ['data' => 'This has come from a dedicated API subdomain with restricted access.'];

});

});

**AngularJS Frontend Example**

We are using AngularJS as a front-end, relying on the API calls to the Laravel back-end authentication server for user authentication and sample data, plus the API server for cross-origin example data. Once we go to the homepage of our project, the backend will serve the resources/views/spa.blade.php view that will bootstrap the Angular application.

Here is the folder structure of the Angular app:

public/

|-- css/

`-- bootstrap.superhero.min.css

|-- lib/

|-- loading-bar.css

|-- loading-bar.js

`-- ngStorage.js

|-- partials/

|-- home.html

|-- restricted.html

|-- signin.html

`-- signup.html

`-- scripts/

|-- app.js

|-- controllers.js

`-- services.js

**Bootstrapping the Angular Application**

spa.blade.php contains the bare essentials needed to run the application. We’ll use Twitter Bootstrap for styling, along with a custom theme from [Bootswatch](https://bootswatch.com/). To have some visual feedback when making an AJAX call, we’ll use the [angular-loading-bar](https://chieffancypants.github.io/angular-loading-bar/) script, which intercepts XHR requests and creates a loading bar. In the header section, we have the following stylesheets:

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

<link rel="stylesheet" href="/css/bootstrap.superhero.min.css">

<link rel="stylesheet" href="/lib/loading-bar.css">

The footer of our markup contains references to libraries, as well as our custom scripts for Angular modules, controllers and services.

<script src="http://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>

<script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.2/js/bootstrap.min.js"></script>

<script src="http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.14/angular.min.js"></script>

<script src="http://cdnjs.cloudflare.com/ajax/libs/angular.js/1.3.14/angular-route.min.js"></script>

<script src="/lib/ngStorage.js"></script>

<script src="/lib/loading-bar.js"></script>

<script src="/scripts/app.js"></script>

<script src="/scripts/controllers.js"></script>

<script src="/scripts/services.js"></script>

</body>

We are using [ngStorage](https://github.com/gsklee/ngStorage) library for AngularJS, to save tokens into the browser’s local storage, so that we can send it on each request via the Authorization header.

In the production environment, of course, we would minify and combine all our script files and stylesheets in order to improve performance.

I’ve created a navigation bar using Bootstrap that will change the visibility of appropriate links, depending on the sign-in status of the user. The sign-in status is determined by the presence of a token variable in the controller’s scope.

<div class="navbar-header">

<button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target=".navbar-collapse">

<span class="sr-only">Toggle navigation</span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

<span class="icon-bar"></span>

</button>

<a class="navbar-brand" href="#">JWT Angular example</a>

</div>

<div class="navbar-collapse collapse">

<ul class="nav navbar-nav navbar-right">

<li data-ng-show="token"><a ng-href="#/restricted">Restricted area</a></li>

<li data-ng-hide="token"><a ng-href="#/signin">Sign in</a></li>

<li data-ng-hide="token"><a ng-href="#/signup">Sign up</a></li>

<li data-ng-show="token"><a ng-click="logout()">Logout</a></li>

</ul>

</div>

**Routing**

We have a file named app.js which is responsible for configuring all our front end routes.

angular.module('app', [

'ngStorage',

'ngRoute',

'angular-loading-bar'

])

.constant('urls', {

BASE: 'http://jwt.dev:8000',

BASE\_API: 'http://api.jwt.dev:8000/v1'

})

.config(['$routeProvider', '$httpProvider', function ($routeProvider, $httpProvider) {

$routeProvider.

when('/', {

templateUrl: 'partials/home.html',

controller: 'HomeController'

}).

when('/signin', {

templateUrl: 'partials/signin.html',

controller: 'HomeController'

}).

when('/signup', {

templateUrl: 'partials/signup.html',

controller: 'HomeController'

}).

when('/restricted', {

templateUrl: 'partials/restricted.html',

controller: 'RestrictedController'

}).

otherwise({

redirectTo: '/'

});

Here we can see that we have defined four routes that are handled by either HomeController or RestrictedController. Every route corresponds to a partial HTML view. We have also defined two constants that contain URLs for our HTTP requests to the backend.

**Request Interceptor**

The $http service of AngularJS allows us to communicate with the backend and make HTTP requests. In our case we want to intercept every HTTP request and inject it with an Authorization header containing our JWT if the user is authenticated. We can also use an interceptor to create a global HTTP error handler. Here is an example of our interceptor that injects a token if it’s available in browser’s local storage.

$httpProvider.interceptors.push(['$q', '$location', '$localStorage', function ($q, $location, $localStorage) {

return {

'request': function (config) {

config.headers = config.headers || {};

if ($localStorage.token) {

config.headers.Authorization = 'Bearer ' + $localStorage.token;

}

return config;

},

'responseError': function (response) {

if (response.status === 401 || response.status === 403) {

$location.path('/signin');

}

return $q.reject(response);

}

};

}]);

**Controllers**

In the controllers.js file, we have defined two controllers for our application: HomeController and RestrictedController. HomeController handles sign-in, sign-up and logout functionality. It passes the username and password data from the sign-in and sign-up forms to the Auth service, which sends HTTP requests to the backend. It then saves the token to local storage, or shows an error message, depending on the response from the backend.

angular.module('app')

.controller('HomeController', ['$rootScope', '$scope', '$location', '$localStorage', 'Auth',

function ($rootScope, $scope, $location, $localStorage, Auth) {

function successAuth(res) {

$localStorage.token = res.token;

window.location = "/";

}

$scope.signin = function () {

var formData = {

email: $scope.email,

password: $scope.password

};

Auth.signin(formData, successAuth, function () {

$rootScope.error = 'Invalid credentials.';

})

};

$scope.signup = function () {

var formData = {

email: $scope.email,

password: $scope.password

};

Auth.signup(formData, successAuth, function () {

$rootScope.error = 'Failed to signup';

})

};

$scope.logout = function () {

Auth.logout(function () {

window.location = "/"

});

};

$scope.token = $localStorage.token;

$scope.tokenClaims = Auth.getTokenClaims();

}])

RestrictedController behaves the same way, only it fetches the data by using the getRestrictedData and getApiData functions on the Data service.

.controller('RestrictedController', ['$rootScope', '$scope', 'Data', function ($rootScope, $scope, Data) {

Data.getRestrictedData(function (res) {

$scope.data = res.data;

}, function () {

$rootScope.error = 'Failed to fetch restricted content.';

});

Data.getApiData(function (res) {

$scope.api = res.data;

}, function () {

$rootScope.error = 'Failed to fetch restricted API content.';

});

}]);

The backend is responsible for serving the restricted data only if the user is authenticated. This means that in order to respond with the restricted data, the request for that data needs to contain a valid JWT inside its Authorization header or query string. If that is not the case, the server will respond with a 401 Unauthorized error status code.

**Auth Service**

The Auth service is responsible for making the sign in and sign up HTTP requests to the backend. If the request is successful, the response contains the signed token, which is then base64 decoded, and the enclosed token claims information is saved into a tokenClaims variable. This is passed to the controller via the getTokenClaims function.

angular.module('app')

.factory('Auth', ['$http', '$localStorage', 'urls', function ($http, $localStorage, urls) {

function urlBase64Decode(str) {

var output = str.replace('-', '+').replace('\_', '/');

switch (output.length % 4) {

case 0:

break;

case 2:

output += '==';

break;

case 3:

output += '=';

break;

default:

throw 'Illegal base64url string!';

}

return window.atob(output);

}

function getClaimsFromToken() {

var token = $localStorage.token;

var user = {};

if (typeof token !== 'undefined') {

var encoded = token.split('.')[1];

user = JSON.parse(urlBase64Decode(encoded));

}

return user;

}

var tokenClaims = getClaimsFromToken();

return {

signup: function (data, success, error) {

$http.post(urls.BASE + '/signup', data).success(success).error(error)

},

signin: function (data, success, error) {

$http.post(urls.BASE + '/signin', data).success(success).error(error)

},

logout: function (success) {

tokenClaims = {};

delete $localStorage.token;

success();

},

getTokenClaims: function () {

return tokenClaims;

}

};

}

]);

**Data Service**

This is a simple service that makes requests to the authentication server as well as the API server for some dummy restricted data. It makes the request, and delegates success and error callbacks to the controller.

angular.module('app')

.factory('Data', ['$http', 'urls', function ($http, urls) {

return {

getRestrictedData: function (success, error) {

$http.get(urls.BASE + '/restricted').success(success).error(error)

},

getApiData: function (success, error) {

$http.get(urls.BASE\_API + '/restricted').success(success).error(error)

}

};

}

]);

**Beyond This JSON Web Token**

Token-based authentication enables us to construct decoupled systems that are not tied to a particular authentication scheme. The token might be generated anywhere and consumed on any system that uses the same secret key for signing the token. They are mobile ready, and do not require us to use cookies.

JSON Web Tokens work across all popular programming languages and are quickly gaining in popularity. They are backed by companies like Google, Microsoft and Zendesk. Their standard specification by Internet Engineering Task Force (IETF) is [still in the draft version](https://tools.ietf.org/html/draft-ietf-oauth-json-web-token-32) and may change slightly in the future.

There is still a lot to cover about JWTs, such with how to handle the [security](https://www.toptal.com/security/10-most-common-web-security-vulnerabilities) details, and refreshing tokens when they expire, but the JSON Web Token tutorial should demonstrate the basic usage and, more importantly, the advantages of using JWTs.

Interview Qustion releted this Qustion and answer

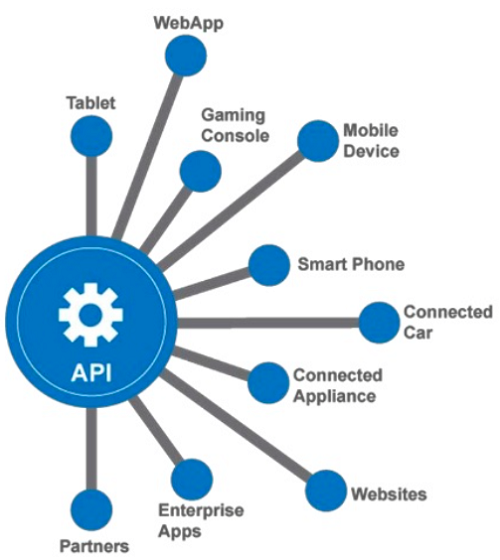
**SECTION 2**

**Why Does Security Matter?**

Security aspects should be a serious consideration when designing and deploying a RESTful API.

**Organizations Understand the Need for API Security**

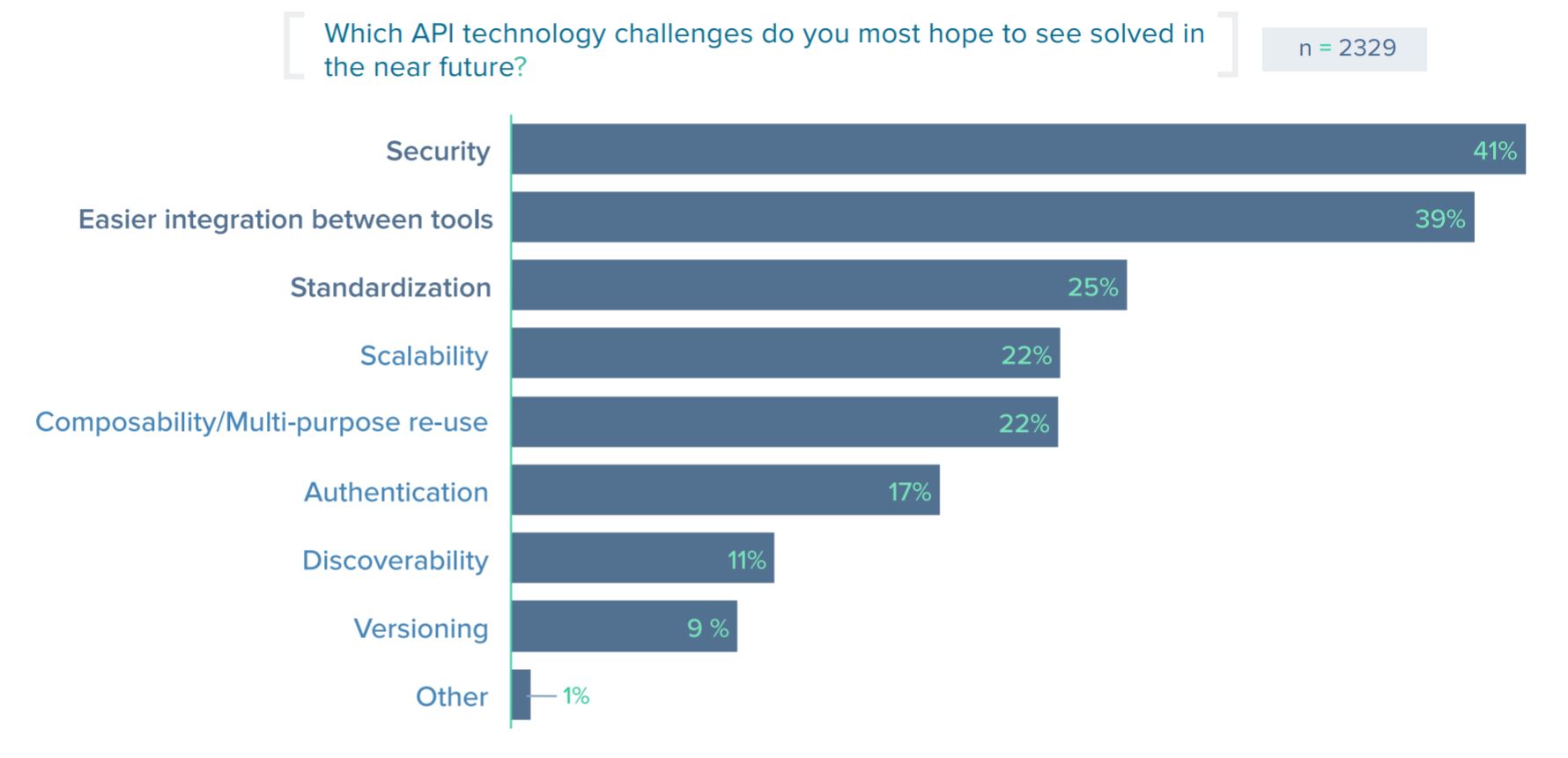
In today’s connected world, where information is being shared via APIs to external stakeholders and within internal teams, security is a top concern.



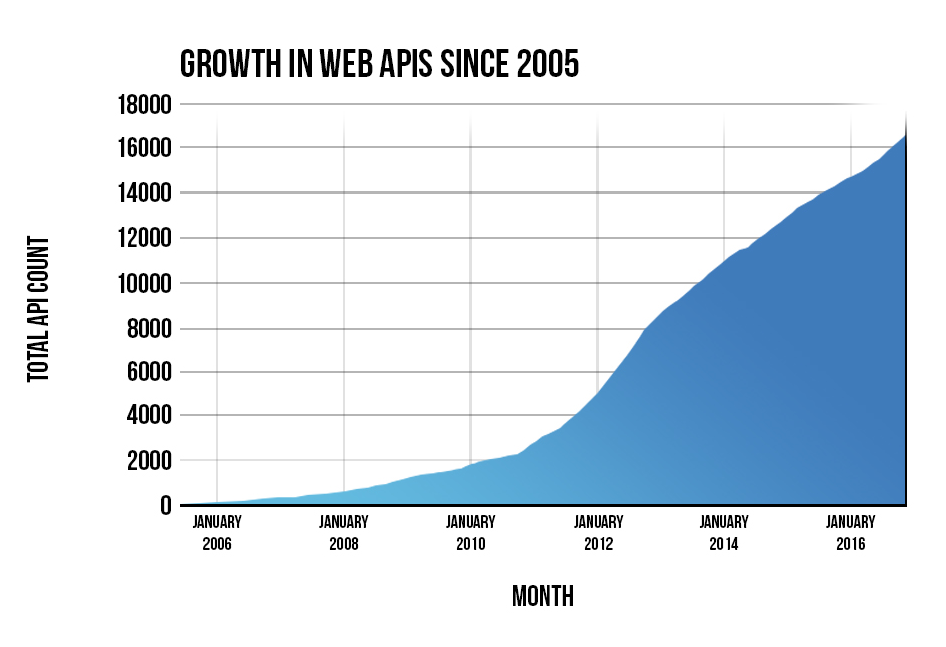
API security is the single biggest challenge organizations want to see solved in the years ahead, and solving the security challenge is expected to be a catalyst for growth in the API world.

According to research by SmartBear presented in their State of APIs Report 2016:

* Security is the #1 technology challenge teams want to see solved; 41.2% of respondents say security is the biggest API technology challenge they hope to see solved.
* Security is the #4 technology area expected to drive the most API growth in the next two years; 24% of API providers say digital security will drive the most API growth in the next two years.
* 40.4% of API providers are currently utilizing a tool for testing API security.



With the explosive growth of RESTful APIs, the security layer is often the one that is most overlooked in the architectural design of the API.



Below, one will find some recommendations to consider prior starting to develop a RESTful API.

**Data Protection**

A RESTful API is the way in which a given service can present value to the world. As a result, protection of the data provided via RESTful endpoints should always be a high priority.

You have to define clear access rights, especially for methods like DELETE (deletes a resource) and PUT (updates a resource). Those methods must be accessed by authenticated users only, and for each such call, an audit must be saved.

**TLS**

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols that provide communications security over a computer network.

When secured by TLS, connections between a client and a server have one or more of the following properties:

* The connection is private (or secure) because symmetric cryptography is used to encrypt the data transmitted.
* The keys for this symmetric encryption are generated uniquely for each connection and are based on a shared secret negotiated at the start of the session.
* The identity of the communicating parties can be authenticated using public-key cryptography.
* The connection ensures integrity because each message transmitted includes a message integrity check using a message authentication code to prevent undetected loss or alteration of the data during transmission.

**DOS Attacks**

In a Denial of Service (DOS) attack, the attacker usually sends excessive messages asking the network or server to authenticate requests that have invalid return addresses. DOS attacks can render a RESTful API into a non-functional state if the right security measures are not taken.

Today, even if your API is not exposed to the public, it still might be accessible by others. This means that REST API security is getting more and more valuable and important. Consider that someone succeeds in making a DOS attack- it means that all the connected clients (partners, apps, mobile devices, and more) will not be able to access your API.

**Anti-Farming**

Today, there are several marketing-heavy websites that offer consumers the best deal on everything from flights to vehicles and even groceries. In many of these cases, the aggregated service is taking advantage of other APIs to obtain the information they want you to utilize. When this happens, the RESTful API is being farmed out for the benefit of another entity.

In case your API does not have an Authorization/Authentication mechanism, it might lead to misuse of your API, loading the servers and the API itself, making it less responsive to others.

## REST API Authentication Types Overview

RESTful applications rely on the underlying security of the API ecosystem rather than including security within the REST architecture style. In addition to securing RESTful API calls with the HTTPS protocol, session-based authentication should be utilized. Currently, most RESTful applications leverage OAuth 2.0 and JWT is the newcomer that is gaining more and more popularity with API developers.

OAuth, JWT, and Basic Auth all use headers for transmitting credentials, and API providers should be doing the same with all API keys. While easy to do as parameters, they are more secure as headers.

## API Keys

Not any formal standard, but something in common practice by API providers, and supported by API management providers. It is the usage of one or two keys what accompany every API call. API keys are really more about identifying the application and user over being anything about security, but is perceived as secure by many.

Public REST services without access control run the risk of being farmed, leading to excessive bills for bandwidth or compute cycles. API keys can be used to mitigate this risk. They are also often used by organization to monetize APIs; instead of blocking high-frequency calls, clients are given access in accordance to a purchased access plan.

Typically, an API key gives full access to every operation an API can perform, including writing new data or deleting existing data. If you use the same API key in multiple apps, a broken app could destroy your users' data without an easy way to stop just that one app. Some apps let users generate new API keys, or even have multiple API keys with the option to revoke one that may have gone into the wrong hands. The ability to change an API key limits the security downsides.

## Security Assessment Markup Language (SAML)

Security Assertion Markup Language (SAML) is an XML-based framework for authentication and authorization between two entities: A Service Provider and an Identity Provider.

The Service Provider agrees to trust the Identity Provider to authenticate users. In return, the Identity Provider generates an authentication assertion, which indicates that a user has been authenticated.

SAML is a standard single sign-on (SSO) format. Authentication information is exchanged through digitally signed XML documents. It's a complex single sign-on (SSO) implementation that enables seamless authentication, mostly between businesses and enterprises.

**OAuth 2**

Created in 2006, OAuth 2 is an open standard for authentication protocol that provides authorization workflow over HTTP and authorizes devices, servers, applications, and APIs with access tokens instead of credentials. OAuth gained popularity from usage by Facebook, Google, Microsoft, and Twitter, who allow usage of their accounts to be shared with third-party applications or websites.

OAuth 2.0 can be used to read data of a user from another application without compromising the user’s personal and sensitive data, like user credentials. It also supplies the authorization workflow for web, desktop applications, and mobile devices.

The previous versions of this spec, OAuth 1.0 and 1.0a, were much more complicated than OAuth 2.0. The biggest change in the latest version is that it’s no longer required to sign each call with a keyed hash. The most common implementations of OAuth use one or both of these tokens instead:

* **access token**: sent like an API key, it allows the application to access a user’s data; optionally, access tokens can expire.
* **refresh token**: optionally part of an OAuth flow, refresh tokens retrieve a new access token if they have expired.

Since an access token is like a special type of API key, the most likely place to put it is the authorization header, like so:

Authorization: Bearer 1234567890abcdef

## JSON Web Token (JWT)

JSON Web Token (JWT) is an open standard extension of OAuth 2.0 for creating access tokens that assert some number of claims.

Whereas API keys and OAuth tokens are always used to access APIs, JSON Web Tokens (JWT) can be used in many different scenarios. In fact, JWT can store any type of data, which is where it excels in combination with OAuth.

Like OAuth access tokens, JWT tokens should be passed in the Authorization header:

## JSON Web Token (JWT)

JSON Web Token (JWT) is an open standard extension of OAuth 2.0 for creating access tokens that assert some number of claims.

Whereas API keys and OAuth tokens are always used to access APIs, JSON Web Tokens (JWT) can be used in many different scenarios. In fact, JWT can store any type of data, which is where it excels in combination with OAuth.

Like OAuth access tokens, JWT tokens should be passed in the Authorization header:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJob21lcGFnZSI6Imh0dHBzOi8vemFwaWVyLmNvbSIsInRhZ2xpbmUiOiJaYXBpZXIgbWFrZXMgeW91IGhhcHBpZXIiLCJmb3JtIjoiaHR0cHM6Ly96YXBpZXIudHlwZWZvcm0uY29tL3RvL0hkRVk0eiJ9.E3EtYy2y7BRn4eS0RIyDAAh-KAsa6dVV91ULbBJCRJw

**What Are JSON Web Tokens?**

JSON Web Token (JWT), pronounced "jot," is an open standard ([RFC 7519](https://tools.ietf.org/html/rfc7519)) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed.

JWT features:

* **Compact**: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.
* **Self-contained**: The payload contains all the required information about the user, avoiding the need to query the database more than once.

**What Does a JWT Look Like?**

JSON Web Tokens consist of three parts separated by dots (.), which are:

• Header

• Payload

• Signature

For example:

aaaaaaaaaa.bbbbbbbbbbb.cccccccccccc

Since there are three parts separated by a (.), each section is created differently.



**Header**

The header carries two parts:

Declaring the type, which is JWT, and the hashing algorithm to use (HMAC SHA256 in this case).

{

"typ": "JWT",

"alg": "HS256"

}

In this JSON, the value of the “typ” key specifies that the object is a JWT, and the value of the “alg” key specifies which hashing algorithm is being used to create the JWT signature component. In our example, we’re using the HMAC-SHA256 algorithm, a hashing algorithm that uses a secret key, to compute the signature.

Now all that is left is to Base64 Encode this JSON and we have the first part of our JSON web token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

**Payload**

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically the user) and additional metadata. Notice that the claim names are only three characters long, as JWT is meant to be compact.

There are three types of claims: reserved, public, and private claims.

* **Reserved claims**: These are a set of predefined claims which are not mandatory but recommended, to provide a set of useful, interoperable claims. In the preferred JWT standard, some claims are:
  1. **exp**: Expiration Value (Type: Number – DateTime). This is commonly used by the issuer. It has a numeric value that denotes a time interval. The token will expire after this time interval elapses from the current time.
  2. **iss**: Issuer (Type: String or URI)
  3. **sub**: Subject (Type: String)
  4. **aud**: Audience (Type: String or URI)
  5. **nbf**: Not Before (Type: Number – DateTime). Denotes a numeric value of time, before which the token must not be accepted for processing.
  6. **iat**: Issued At (Type: Number – DateTime). Denotes the numeric format of issued date and time.
  7. **jti**: JWT ID (Type: String). The unique identifier for the JWT token.
* **Public claims**: These can be defined at will using JWTs, but to avoid collisions, they should be defined in the IANA JSON Web Token Registry or defined as a URI that contains a collision-resistant namespace.
* **Private claims**: These are the custom claims created to share information between parties that agree on using them. An example of a payload is as follows:

{

"sub": "AB324901",

"name": "Michael",

"admin": true,

"exp": 2601638760

}

The Base64 of the above encoded form should look like:

eyJzdWIiOiJBQjMyNDkwMSIsIm5hbWUiOiJNaWNoYWVsIiwiYWRtaW4iOnRydWUsImV4cCI6MjYwMTYzODc2MH0

**Signature**

To create the signature part, you have to take the encoded header, the encoded payload, a secret, and the algorithm specified in the header, and sign it.

For example, if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

data = base64urlEncode( header ) + "." + base64urlEncode( payload )

signature = Hash( data, secret );

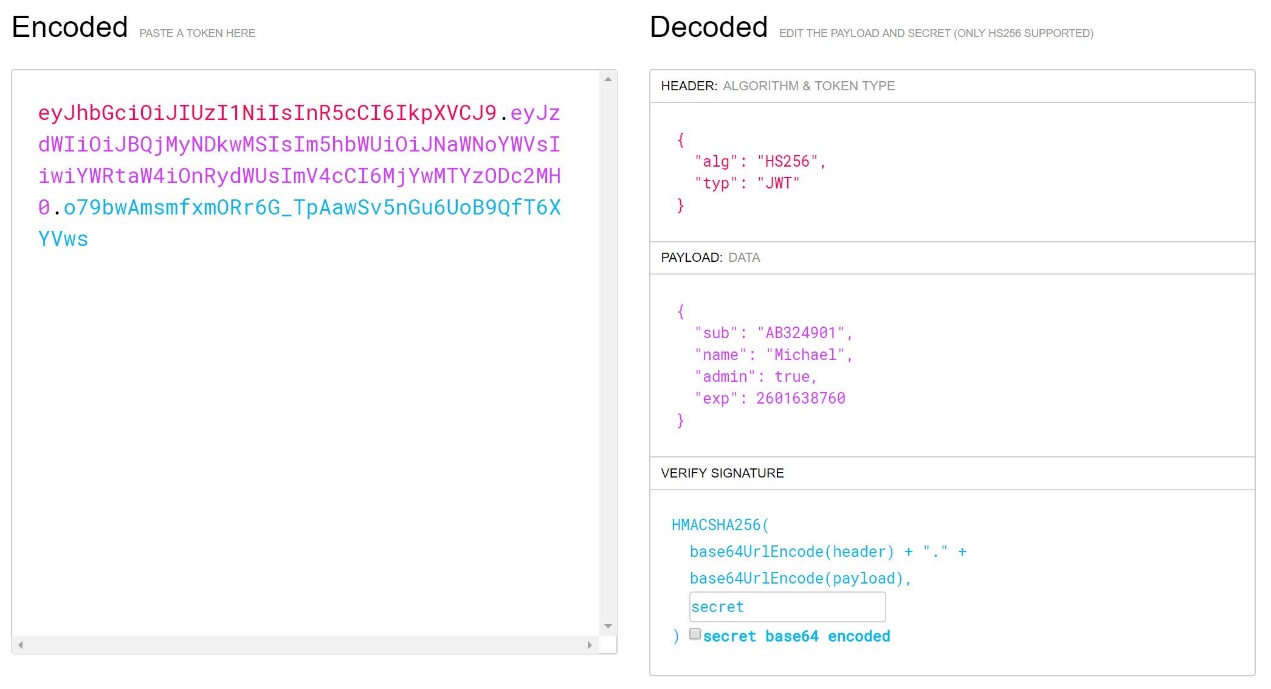
The signature is used to verify that the sender of the JWT is who they say they are, and to ensure that the message wasn't changed along the way.

**Putting It All Together**

The output is three Base64 strings separated by dots that can be easily passed in HTML and HTTP environments, while being more compact when compared to XML-based standards such as SAML.

The following shows a JWT that has the previous header and payload encoded, and it is signed with a secret.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJBQjMyNDkwMSIsIm5hbWUiOiJNaWNoYWVsIiwiYWRtaW4iOnRydWUsImV4cCI6MjYwMTYzODc2MH0.o79bwAmsmfxmORr6G\_TpAawSv5nGu6UoB9QfT6XYVws



You can create your own JWT here - <https://jwt.io/>

**When Should You Use JSON Web Tokens?**

Here are some scenarios where JSON Web Tokens are useful:

* **Authentication**: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single sign-on is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
* **Information Exchange**: JSON Web Tokens are a good way of securely transmitting information between parties. Because JWTs can be signed—for example, using public/private key pairs—you can be sure the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

## How Do JSON Web Tokens Work?

In authentication, when the user successfully logs in using their credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage, but cookies can be also used), instead of the traditional approach of creating a session in the server and returning a cookie.

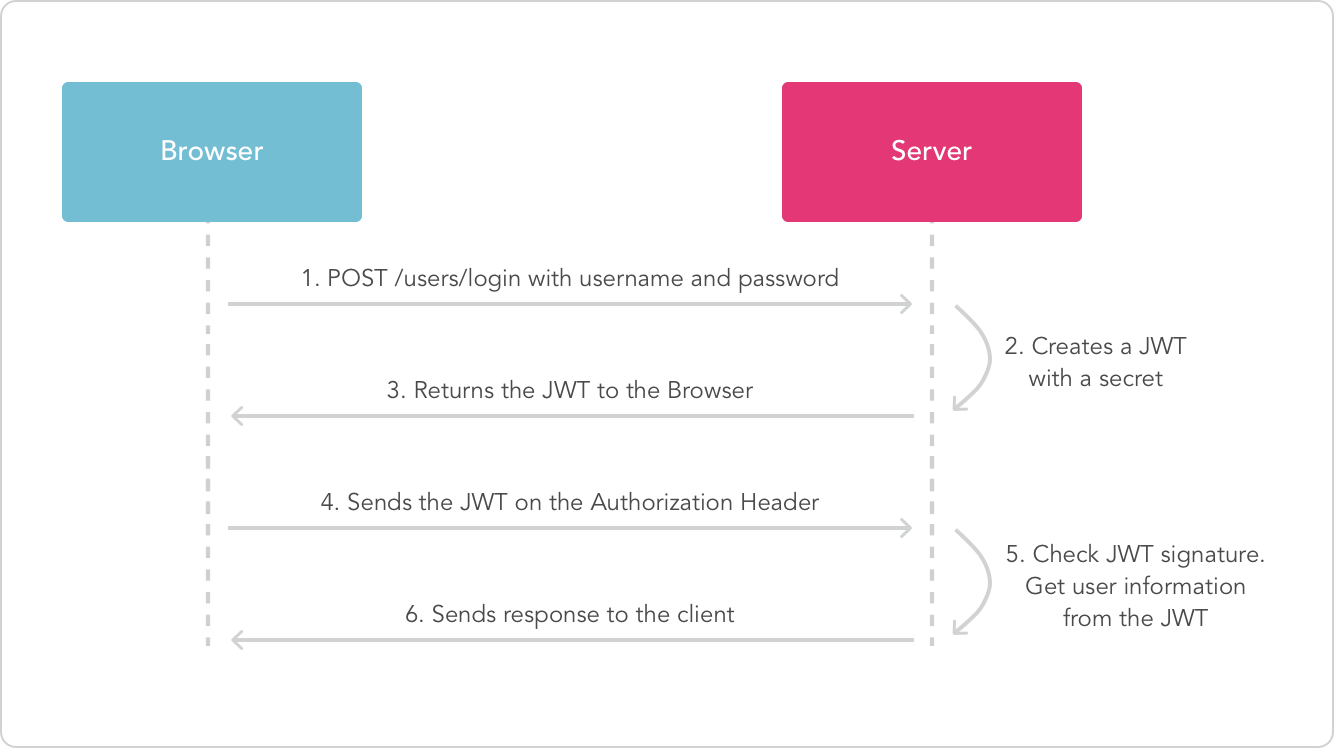
Whenever the user wants to access a protected route or resource, the user agent should send the JWT, typically in the Authorization header using the Bearer schema. The content of the header should look like the following:

Authorization: Bearer <token>

This is a stateless authentication mechanism as the user state is never saved in the server’s memory. The server's protected routes will check for a valid JWT in the Authorization header, and if it's present, the user will be allowed to access protected resources. As JWTs are self-contained, all the necessary information is there, reducing the need to query the database multiple times.

This allows you to fully rely on data APIs that are stateless, and even make requests to downstream services. It doesn't matter which domains are serving your APIs, so Cross-Origin Resource Sharing (CORS) won't be an issue as it doesn't use cookies.

The following diagram shows this process:



### In Conclusion

We went over what JWT is, how the tokens are created and validated, and how they can be used to ensure trust between an application and its users. This is a starting point for understanding the fundamentals of JWTs and why they are useful. JWTs are just one piece of the puzzle in ensuring trust and security in your application.

## Secure Your REST API: Best Practices

### Rate Limiting

An API key is a valuable strategy to provide a level of identity to consumers of a RESTful API. In addition to providing tiered services, another benefit of using API keys is the ability to throttle usage of the API, and the logging of all API calls made allows API providers to limit the rate of consumption for all API users. Putting caps on the number of API calls that can be made for any single API resource, dictating consumption by the second, minute, day, other relevant constraint.

### Authorization

#### Protect HTTP Methods

RESTful APIs often use GET (read), POST (create), PUT (replace/update) and DELETE (to delete a record).

Not all of these are valid choices for every single resource collection, user, or action. Make sure the incoming HTTP method is valid for the session token/API key and associated resource collection, action, and record.

#### Whitelist Allowable Methods

It is common with RESTful services to allow multiple methods for a given URL for different operations on that entity.

For example, a GET request might read the entity, while PUT would update an existing entity, POST would create a new entity, and DELETE would delete an existing entity.

It is important for the service to properly restrict the allowable verbs such that only the allowed verbs would work, while all others would return a proper response code (for example, a 403 Forbidden).

#### Protect Privileged Actions and Sensitive Resource Collections

Not every user has a right to every web service. This is vital, as you don't want administrative web services to be misused:  https://example.com/admin/exportAllData

The session token or API key should be sent along as a cookie or body parameter to ensure that privileged collections or actions are properly protected from unauthorized use.

#### Protect Against Cross-Site Request Forgery

For resources exposed by RESTful web services, it's important to make sure any PUT, POST, and DELETE request is protected from Cross-Site Request Forgery. Typically, one would use a token-based approach.

CSRF is easily achieved — even using random tokens — if any XSS exists within your application, so please make sure you understand how to prevent XSS.

### Input Validation

Everything you know about input validation applies to RESTful web services, but add 10%, because automated tools can easily fuzz your interfaces for hours on end at high velocity. So:

**Assist the user > Reject input > Sanitize (filtering) > No input validation**

Assisting the user makes the most sense, as the most common scenario is "problem exists between keyboard and chair" (PEBKAC).

#### URL Validations

Web applications/web services use input from HTTP requests (and occasionally files) to determine how to respond.

Attackers can tamper with any part of an HTTP request, including the URL, query string, headers, cookies, form fields, and hidden fields, to try to bypass the site’s security mechanisms.

Common names for common input tampering attacks include: forced browsing, command insertion, cross site scripting, buffer overflows, format string attacks, SQL injection, cookie poisoning, and hidden field manipulation.

#### XML Input Validation

XML-based services must ensure that they are protected against common XML-based attacks by using secure XML-parsing.

This typically means protecting against XML External Entity attacks, XML-signature wrapping, etc.

See [http://ws-attacks.org](http://ws-attacks.org/) for examples of such attacks.

### Output Encoding

#### Security Headers

To make sure the content of a given resource is interpreted correctly by the browser, the server should always send the Content-Type header with the correct Content-Type, and the Content-Type header should preferably include a charset.

The server should also send an X-Content-Type-Options: nosniff to make sure the browser does not try to detect a different Content-Type than what is actually sent (as this can lead to XSS).

Additionally, the client should send an X-Frame-Options: deny to protect against drag-and-drop clickjacking attacks in older browsers.

#### JSON Encoding

A key concern with JSON encoders is preventing arbitrary JavaScript remote code execution within the browser... or, if you're using Node.js, on the server. It's vital that you use a proper JSON serializer to encode user-supplied data properly to prevent the execution of user-supplied input on the browser.

#### XML Encoding

XML should never be built by string concatenation. It should always be constructed using an XML serializer. This ensures that the XML content sent to the browser is parseable and does not contain XML injection. For more information, please see the Web Service Security Cheat Sheet.

### HTTP Status Codes

HTTP defines status codes. When you design your REST API, don't just use 200 for success or 404 for error.

Here are some guidelines to consider for each REST API status return code. Proper error handling may help to validate the incoming requests and better identify the potential security risks.

| **RETURN CODE** | **MESSAGE** | **DESCRIPTION** |
| --- | --- | --- |
| **200** | OK | Response to a successful REST API action. The HTTP method  can be GET, POST, PUT, PATCH or DELETE. |
| **201** | Created | The request has been fulfilled and the resource created.  A URI for the created resource is returned in the Location header. |
| **400** | Bad Request | The request is malformed, such as a message body format error,  missing headers, etc. |
| **401** | Unauthorized | Wrong or no authentication ID/password provided. |
| **403** | Forbidden | Used when the authentication succeeded but the authenticated user  doesn't have permission to the requested resource. |
| **404** | Not Found | When a non-existent resource is requested. |
| **406** | Unacceptable | The client presented a content type in the Accept header which is  not supported by the server API. |
| **405** | Method Not Allowed | The error for an unexpected HTTP method. For example,  the REST API is expecting HTTP GET, but HTTP PUT is used. |
| **413** | Payload Too Large | Used to signal that the request size exceeded the given limit, e.g. regarding file uploads and to ensure that the requests have reasonable sizes. |
| **415** | Unsupported Media Type | The requested content type is not supported by the REST service.  This is especially effective when you are working primary with JSON  or XML media types. |
| **429** | Too Many Requests | The error is used when there may be a DOS attack detected or the  request is rejected due to rate limiting. |