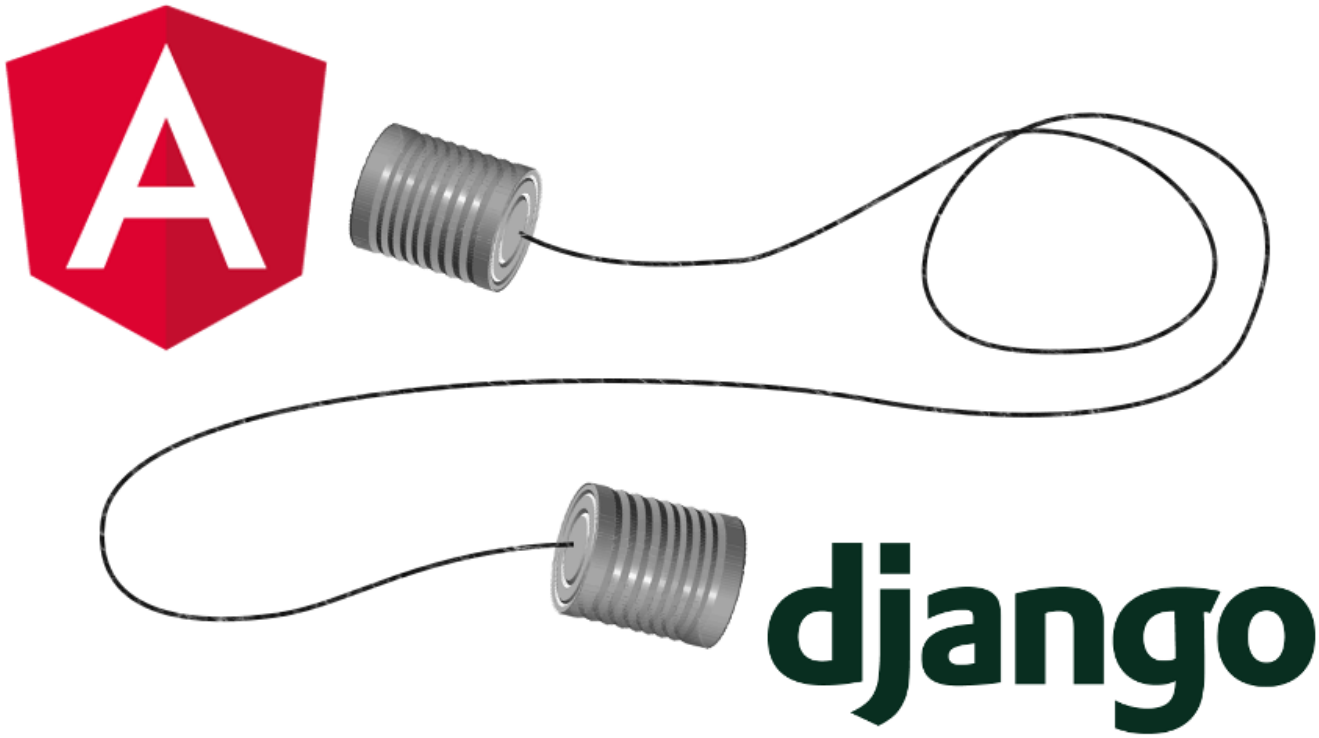



# Angular API Calls with Django: Authentication with JWT

*by Keith Dechant , Software Architect*



*Filed under: AngularJS (/help/angularjs) | Django (/help/django) | Python (/help/python)*

Metal Toad is an AWS Managed Services (/services/aws-managed-services) provider. In addition to Angular work we recommend checking out our article on how to host a website on AWS (/blog/how-host-website-aws-5-minutes) in 5 minutes.

Curious about how to make API calls with Angular 6 and the HttpClient service? This tutorial will show you some techniques for building a decoupled micro-blogging application using Angular 6 and the Django Rest Framework (DRF). Along the way, we will learn the following:

How to set up the back end of the app using Django and the Django Rest Framework API
Creating a simple Angular 6 single-page app which can query the API
Authenticating users with JSON Web Tokens (JWT)

Ready? Let's get started!

# Try this at home

The source code for this tutorial can be found on GitHub at https://github.com/kdechant/angular-django-example (https://github.com/kdechant/angular-django-example).

The example code requires Python 3 (Django 2.x requires Python 3.4 or higher) and uses a SQLite database, so install should be easy and painless.

# Background info

My previous tutorial (https://www.metaltoad.com/blog/angular-5-making-api-calls-httpclient-service) will give you a basic understanding of how to make API calls using Angular.

If you're new to Django and DRF, you can find some useful tutorials at the Django project (https://docs.djangoproject.com/en/2.0/intro/tutorial01/) and the Django Rest Framework site (http://www.django-rest-framework.org/tutorial/quickstart/). This example will build upon the basic knowledge from those tutorials.

Some knowledge of pip (https://packaging.python.org/tutorials/installing-packages/) and Python Virtual Environments (http://docs.python-guide.org/en/latest/dev/virtualenvs/) will also be useful here. Be aware that we're using classic virtualenv here and not the newer pipenv yet.

# Technology selection

The demo shown here uses the following technology:

Angular 6.1 - The latest as of the time of this writing
RxJS 6.0 - This is the version included with Angular 6.0. All API calls in this tutorial use the newer RxJS syntax introduced with this version. See my previous post about upgrading from RxJS 5.5 here (https://www.metaltoad.com/blog/angular-6-upgrading-api-calls-rxjs-6).
Angular CLI v6.x
Django 2.1 - The current release of Django
Django Rest Framework - The standard suite for generating a REST API in Django
Python 3.5 or higher - Django 2.x requires Python 3 and no longer supports Python 2.7
Node 8.x or higher

Why these platforms? Angular, because it's a full-featured front-end framework that has tremendous popularity. Django and Django Rest Framework, to provide the ORM and API layer. Python is growing in popularity and is an enjoyable language to develop applications.

# The Django app

The Django app is the back end of our decoupled application. It serves the API endpoints and it also renders the HTML container for the Angular front end app.

We will dive more deeply into DRF and its models, serializers, and viewsets in a later part of this tutorial. For now, we have a simple Django project with Django Rest Framework and the Django Rest Framework JWT packages installed.

```
pip install Django
pip install djangorestframework djangorestframework-jwt.
```

What these do:

the DJango package is the basic framework itself
djangorestframework is the core of DRF and provides the means to build API endpoints
djangorestframework-jwt (https://pypi.org/project/djangorestframework-jwt/) is an extension to DRF which provides an authentication layer using JSON Web Tokens

The vanilla install of Django provides a basic settings file for the application. To activate DRF and the JWT extension, we need to add DRF to our installed apps, and configure its settings:

angular_django_example/settings.py:

```
INSTALLED_APPS = [
    ...
    'rest_framework',
]

REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticatedOrReadOnly',
    ),
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_jwt.authentication.JSONWebTokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
        'rest_framework.authentication.BasicAuthentication',
    ),
}
```

By default, DRF allows Basic and Session authentication. The DEFAULT_AUTHENTICATION_CLASSES setting adds a third authentication mechanism, the JWT.

How do these work?

**Basic Auth** - a username and password are passed with each API request. This provides only a minimum level of security and user credentials are visible in the URLs
**Session Auth** - requires the user to log in through the server-side application before using the API. This is more secure than Basic Auth but is not convenient for working with single-page apps in a framework like Angular.
**JSON Web Tokens** are an industry standard mechanism for generating a token which can be passed in the HTTP headers of each request, authenticating the user. This is the mechanism we will use for authentication.

In addition to the Rest Framework configuration, the JWT package also has its own configuration settings. We will update two settings in particular for our app.

angular_django_example/settings.py

```
JWT_AUTH = {
    'JWT_ALLOW_REFRESH': True,
    'JWT_EXPIRATION_DELTA': datetime.timedelta(seconds=3600),
}
```

JWT tokens have a life span, after which they are no longer valid. The default is only 5 minutes, but we can set it to a longer time (say, 1 hour) using the JWT_EXPIRATION_DELTA setting. The JWT_ALLOW_REFRESH setting enables a feature of DRF-JWT where an application can request a refreshed token with a new expiration date.

## URLs

In addition to the settings, we need to add a few URLs to our API:

angular_django_example/urls.py:

```
from rest_framework_jwt.views import obtain_jwt_token, refresh_jwt_toke

urlpatterns = [
    ... other patterns here ...
    path(r'api-token-auth/', obtain_jwt_token),
    path(r'api-token-refresh/', refresh_jwt_token),
]
```

These endpoints provide us with a means to authenticate via the API and to request a new token.

## The Microblog app

Now that our application-wide settings are configured, we can create the Django app and the Angular app within it.

From the terminal, we can run `python manage.py startapp microblog` to create our new Microblog app. This gives us an empty Django app with the usual views.py, models.py, urls.py, and so on.

We now need to create a simple View and a template which will serve the single-page app.

microblog/views.py:

```
from django.shortcuts import render

def index(request, path=''):
    """
    The home page. This renders the container for the single-page app.
    """
    return render(request, 'index.html')
```

In Django fashion, we will use two template files, base.html, providing the outer HTML shell, and index.html, providing the content of the index page itself.

microblog/templates/base.html:

```
{% load staticfiles %}
<!DOCTYPE html>
<html (http://december.com/html/4/element/html.html)>
<head (http://december.com/html/4/element/head.html)>
  <meta (http://december.com/html/4/element/meta.html) charset="utf-8">
  <title (http://december.com/html/4/element/title.html)>Angular, Djang
  <base (http://december.com/html/4/element/base.html) href="/">
  <meta (http://december.com/html/4/element/meta.html) name="viewport"
  <link (http://december.com/html/4/element/link.html) rel="stylesheet"
</head (http://december.com/html/4/element/head.html)>
<body (http://december.com/html/4/element/body.html)>
  <div (http://december.com/html/4/element/div.html) class="container">
    {% block heading %}
      <h1 (http://december.com/html/4/element/h1.html)>Angular, Django
    {% endblock %}

    {% block content %}{% endblock %}
  </div (http://december.com/html/4/element/div.html)>
</body (http://december.com/html/4/element/body.html)>
</html (http://december.com/html/4/element/html.html)>
```

microblog/templates/index.html:

```
{% extends "base.html" %}
{% load staticfiles %}

{% block content %}
  <p (http://december.com/html/4/element/p.html)>This is a mini-blog ap

  <app-root>Loading the app...</app-root>

  <script (http://december.com/html/4/element/script.html) type="text/j
  <script (http://december.com/html/4/element/script.html) type="text/j
  <script (http://december.com/html/4/element/script.html) type="text/j
  <script (http://december.com/html/4/element/script.html) type="text/j
  <script (http://december.com/html/4/element/script.html) type="text/j

{% endblock %}
```

The `<script>` tags in index.html will load the compiled JavaScript files generated by webpack. We'll need to change a setting in angular.json to make this work, which we'll see momentarily.

For the first part of this tutorial, we are using only built-in models (like django.contrib.auth.models.User) and the Views that are provided by DRF and the JWT extension. In Part 2, we will delve into creating the rest of our micro-blogging application. There, you will see custom models, serializers, and views.

# The Angular App

To install an Angular app within a Django project, we just place the TypeScript source code inside of our "microblog" Django app.

```
cd microblog
ng new front-end
```

This will generate a starter Angular app in microblog/front-end, with the following interesting files:

app - The location of the Angular module, components, and services
angular.json - The configuration for the Angular CLI
dist - The destination where the Angular CLI will place the compiled files. We'll change this in just a moment to be compatible with Django.

Here we have our first conflict between the "Django way" and the "Angular way". Django's built-in "staticfiles" app won't know to look in microblog/front-end/dist to find the compiled JavaScript files and other assets. Django wants the static files for each app to live in a subdirectory called "static". So, we need to edit our angular.json file to instruct Angular CLI to place the files there.

microblog/front-end/angular.json:

```
{
  ...
  "projects": {
    "ng-demo": {
      ...
      "architect": {
        "build": {
          ...
          "options": {
            "outputPath": "../static/front-end",   <-- change this line
...
```

Now, when we run `ng build` it will place the files right where Django wants them.

## What's in our Angular app

The Angular app we're creating here will contain the following pieces:

microblog/front-end/src/app/app.component.html - a template that will contain the login form
microblog/front-end/src/app/app.component.ts - our main component
microblog/front-end/src/app/user.service.ts - a service that will manage the authentication API requests

## The Angular Module

Let's start building the app by configuring our module file.

microblog/front-end/src/app/app.module.ts:

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';    // add this
import { FormsModule } from '@angular/forms';    // add this
import { AppComponent } from './app.component';
import { UserService } from './user.service';    // add this

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, HttpClientModule],    // add th
  providers: [UserService],    // add this
  bootstrap: [AppComponent]
})
export class AppModule { }
```

This is very close to the default Angular module file. We have only added the built-in HttpClientModule and FormsModule, as well as our custom UserService, which we'll build in a minute.

Our app is very simple and has just one Component:

microblog/front-end/src/app/app.component.ts:

```
import {Component, OnInit} from '@angular/core';
import {UserService} from './user.service';
import {throwError} from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {

  /**
   * An object representing the user for the login form
   */
  public user: any;

  constructor(private _userService: UserService) { }

  ngOnInit() {
    this.user = {
      username: '',
      password: ''
    };
  }

  login() {
    this._userService.login({'username': this.user.username, 'password'
  }

  refreshToken() {
    this._userService.refreshToken();
  }

  logout() {
    this._userService.logout();
  }

}
```

Our template contains a login form, and a message to the user once they have logged in. We can also spruce it up a little with some Bootstrap classes.

microblog/front-end/src/app/app.component.html:

```
<h2 (http://december.com/html/4/element/h2.html)>Log In</h2 (http://dec
<div (http://december.com/html/4/element/div.html) class="row" *ngIf="!
  <div (http://december.com/html/4/element/div.html) class="col-sm-4">
    <label (http://december.com/html/4/element/label.html)>Username:</l
    <input (http://december.com/html/4/element/input.html) type="text"
    <span (http://december.com/html/4/element/span.html) *ngFor="let er
    {{ error }}</span (http://december.com/html/4/element/span.html)></
  <div (http://december.com/html/4/element/div.html) class="col-sm-4">
    <label (http://december.com/html/4/element/label.html)>Password:</l
    <input (http://december.com/html/4/element/input.html) type="passwo
    <span (http://december.com/html/4/element/span.html) *ngFor="let er
    {{ error }}</span (http://december.com/html/4/element/span.html)>
  </div (http://december.com/html/4/element/div.html)>
  <div (http://december.com/html/4/element/div.html) class="col-sm-4">
    <button (http://december.com/html/4/element/button.html) (click)="l
  </div (http://december.com/html/4/element/div.html)>
  <div (http://december.com/html/4/element/div.html) class="col-sm-12">
    <span (http://december.com/html/4/element/span.html) *ngFor="let er
  </div (http://december.com/html/4/element/div.html)>
</div (http://december.com/html/4/element/div.html)>
<div (http://december.com/html/4/element/div.html) class="row" *ngIf="_
  <div (http://december.com/html/4/element/div.html) class="col-sm-12">
    Token Expires: {{ _userService.token_expires }}<br (http://december
    <button (http://december.com/html/4/element/button.html) (click)="r
    <button (http://december.com/html/4/element/button.html) (click)="l
  </div (http://december.com/html/4/element/div.html)>
</div (http://december.com/html/4/element/div.html)>
```

When the user first arrives on the page, they will see the login form. Once they have successfully logged in, they will see a welcome message and their username. For the purposes of this demo app, we will also output the expiration time of their JWT authentication token.

Now for what you're been waiting for: the User Service, which handles all the Angular API calls we need to authenticate a user and manage their JWT tokens.

microblog/front-end/src/app/user.service.ts:

```
import {Injectable} from '@angular/core';
import {HttpClient, HttpHeaders} from '@angular/common/http';

@Injectable()
export class UserService {

  // http options used for making API calls
  private httpOptions: any;

  // the actual JWT token
  public token: string;

  // the token expiration date
  public token_expires: Date;

  // the username of the logged in user
  public username: string;

  // error messages received from the login attempt
  public errors: any = [];

  constructor(private http: HttpClient) {
    this.httpOptions = {
      headers: new HttpHeaders({'Content-Type': 'application/json'})
    };
  }

  // Uses http.post() to get an auth token from djangorestframework-jwt
  public login(user) {
    this.http.post('/api-token-auth/', JSON.stringify(user), this.httpO
      data => {
        this.updateData(data['token']);
      },
      err => {
        this.errors = err['error'];
      }
    );
  }

  // Refreshes the JWT token, to extend the time the user is logged in
  public refreshToken() {
    this.http.post('/api-token-refresh/', JSON.stringify({token: this.t
      data => {
        this.updateData(data['token']);
      },
      err => {
        this.errors = err['error'];
      }
    );
  }
```

```
  public logout() {
    this.token = null;
    this.token_expires = null;
    this.username = null;
  }

  private updateData(token) {
    this.token = token;
    this.errors = [];

    // decode the token to read the username and expiration timestamp
    const token_parts = this.token.split(/\./);
    const token_decoded = JSON.parse(window.atob(token_parts[1]));
    this.token_expires = new Date(token_decoded.exp * 1000);
    this.username = token_decoded.username;
  }

}
```

Remember when we installed djangorestframework-jwt that we added two URLs to our urls.py, "/api-token-auth" and "/api-token-refresh"? Here we use Angular's HttpClient service to send POST requests to them.

The UserService's login() method sends the username and password to "/api-token-auth" and receives a token in response. If the login is unsuccessful, we receive some error messages (this.errors) which will be shown to the user within the template.

The UserService's refreshToken() method sends the current token (not the username and password) to the "/api-token-refresh" endpoint. This retrieves a new token for the same user, with a new expiration time.

## Getting the expiration time from JWT tokens

So, what's in a JWT token? The API endpoints will return something like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwib3JpZ1
```

That looks frightening, but it is really just a series of base64-encoded strings glued together. The data payload is stored in JSON format within the second of these strings, between the first and second dot.

We can split by the dot, then run the built-in JavaScript method `window.atob()` and `JSON.parse()` on the second result, to get our token payload:

```
const token_parts = this.token.split(/\./);
const token_decoded = JSON.parse(window.atob(token_parts[1]));
console.log(token_decoded);

// output:
// {
//   "orig_iat": 1528071221,
//   "exp": 1528074821,
//   "username": "user1",
//   "email": "user1@example.com",
//   "user_id": 2
// }
```

This contains the username as well as the user's email address and numeric ID from the database. It also contains the expiration time as a Unix timestamp. In the code above, we used `this.token_expires = new Date(token_decoded.exp * 1000);` to convert this into a JavaScript Date object which provides a nicer display for the user.

## Using the tokens in subsequent API calls

Now that we have a means for our users to log in, it's time to build the rest of the Microblog app. The Django models and custom API endpoints will be covered in a future post, but for now, here's a teaser:

microblog/front-end/app/blog_post.service.ts:

```
import {Injectable} from '@angular/core';
import {HttpClient, HttpHeaders} from '@angular/common/http';
import {UserService} from './user.service';

@Injectable()
export class BlogPostService {

  constructor(private http: HttpClient, private _userService: UserServi
  }

  // send a POST request to the API to create a new blog post
  create(post, token) {
    let httpOptions = {
      headers: new HttpHeaders({
        'Content-Type': 'application/json',
        'Authorization': 'JWT ' + this._userService.token
      })
    };
    return this.http.post('/api/posts', JSON.stringify(post), httpOptio
  }

}
```

Notice how we send the JWT token in the Authorization header on our API call. This will authenticate the user with each request and will allow Django to know which user made the request.

# Further Reading

Continue to Part 2 of this series to see the rest of the micro-blogging app (https://www.metaltoad.com/blog/angular-api-calls-django-part-2-building-micro-blog-app).

Looking for more examples of Angular API calls using HttpClient and ForkJoin? See how I used Angular, Django Rest Framework, HttpClient, and ForkJoin to rebuild a classic text adventure game (https://www.kdechant.com/blog/angular-text-adventure-part-3-the-game-data-models).

For the front-end developers, you may be interested in some examples of Angular animations (https://www.kdechant.com/blog/angular-animations-fade-in-and-fade-out).

Happy coding!

Date posted: June 4, 2018

# Comments

Thu, 06/14/2018 - 09:20

Very Good read

Sun, 08/12/2018 - 10:38

Hey,
thanks for your post its very helpfull,
How I can use expiration date to automatically disconnect user after non use and navigate him to login screen again?
thanks for your help

Fri, 09/07/2018 - 16:44

Great work,That helps me a lot

Sun, 09/16/2018 - 17:46

that's in depth tuto
i find it very helful
thank you so much

Fri, 09/21/2018 - 05:50

Just what I was looking for.
Cheers !!

Sat, 09/22/2018 - 17:19

> Hey thanks but it didn't work for me, everything is ok with django rest auth jwt it works great. when i implement like you didn't with angular, it's seems like it checked if the user exist, show that i'm logged in but when i check django i'm not, and also when i refresh the page i'm not logged in? How can i solve that?

> Sun, 12/30/2018 - 14:51
>
> Thank you very much. This is a nice Tut. I am a student and I have problem. How does it work with only Django server? I meant, without starting Angular server? I have tried similar exericse by starting both Django and Angular server and cross origin policy. But here, how it is working with just Django server?

> Sun, 09/22/2019 - 10:27
>
> Thanks for your article. I am new to Django. I have successfully installed angular project inside django. I run the django server and no errors there. However I see only index.html of Django and do not see my app.component.html. Any ideas, why it might happen? Thanks in advance!

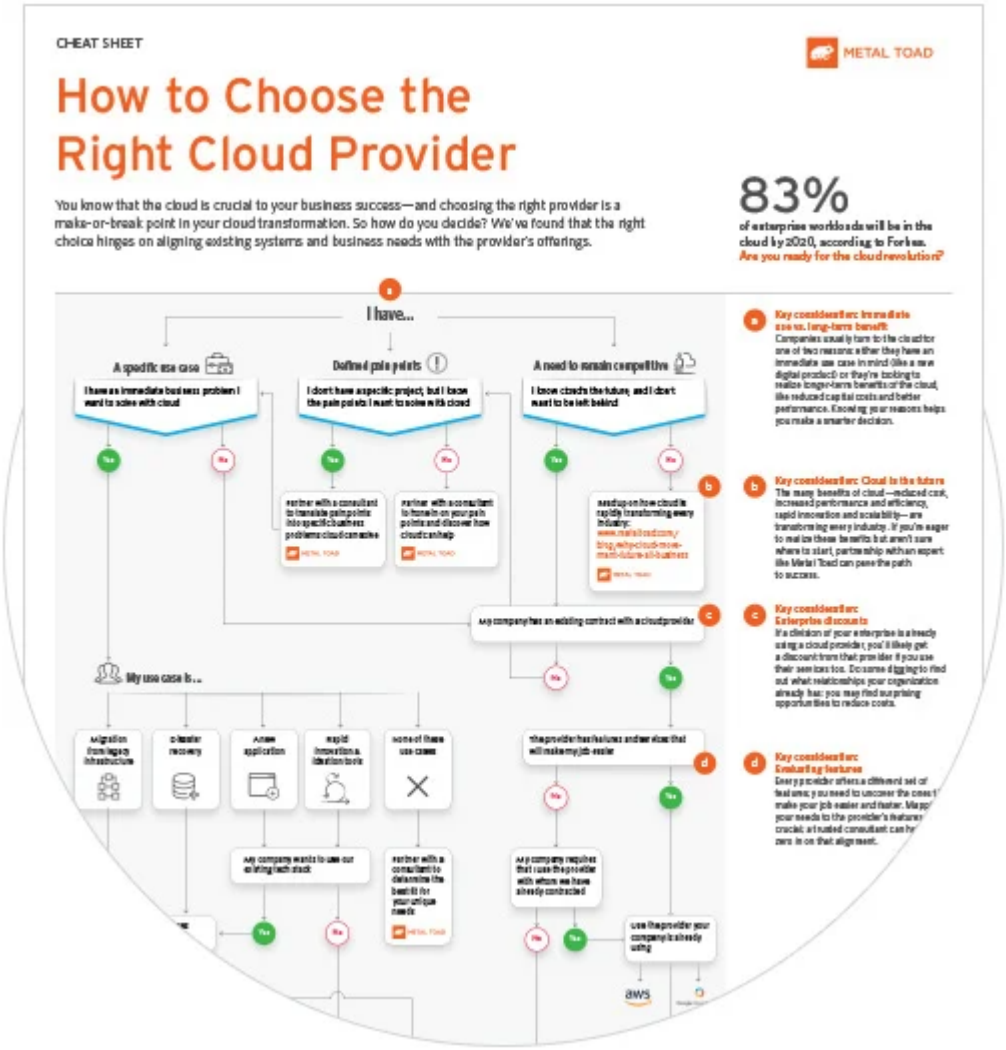
# Add new comment

Your name

Subject

Comment *

Save Preview




(https://metaltoad-6104926.hs-sites.com/landing-page/choose-cloud-cheat-sheet)

# Have questions?

GET IN TOUCH (/CONTACT)

- Values (/blog/metal-toad-corporate-values) • Privacy Policy (/privacy-policy)
- Careers (/careers-metal-toad) • AWS Lambda (/services/aws-lambda)
- AWS Los Angeles (/los-angeles-aws-consultant)
- AWS Media & Entertainment (/industries/media-and-entertainment)
- AWS Oregon (/oregon-aws-consultant) • Cloud Hosting (/cloud-hosting-solutions)
- Technical Help (/help) • Top 10 React JS Tips (/help/top-10-reactjs-tips)
- Top 20 Drupal Tips (/help/top-20-drupal-tips)