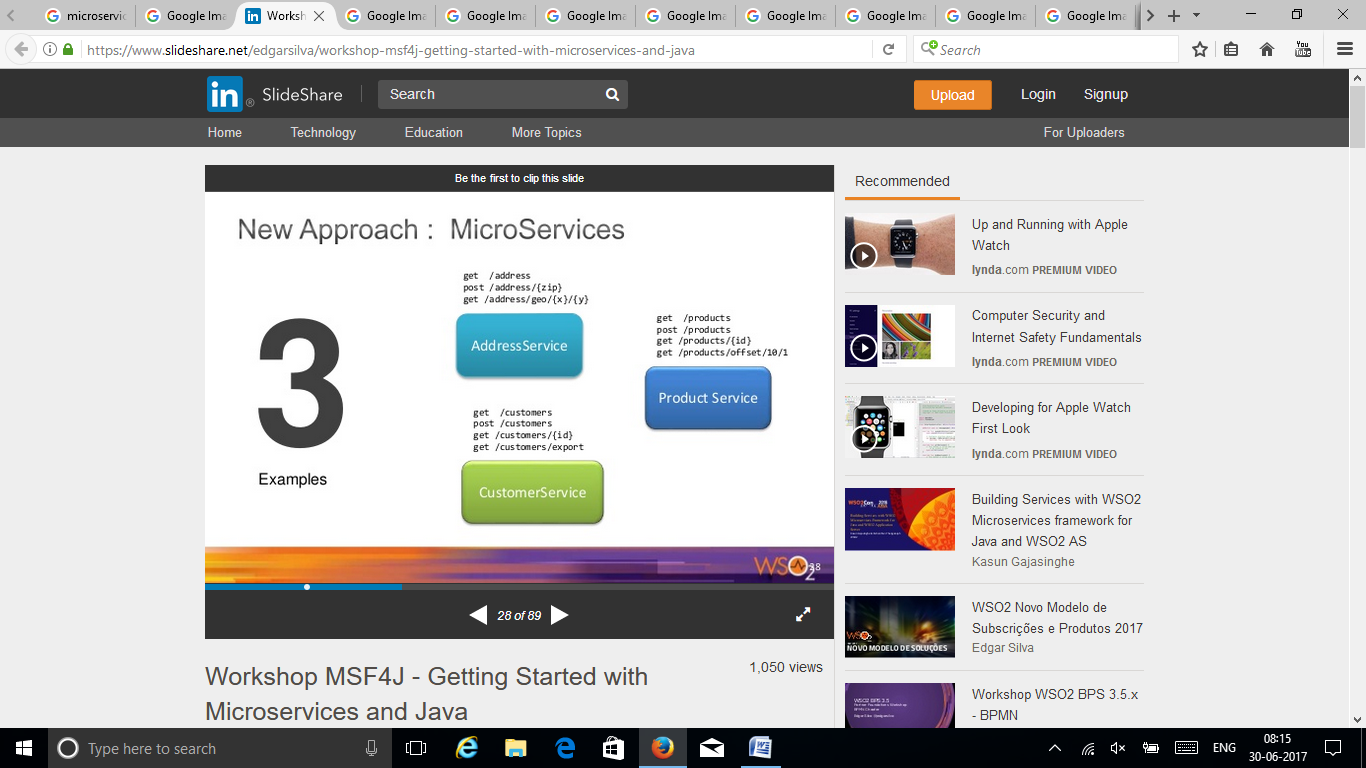
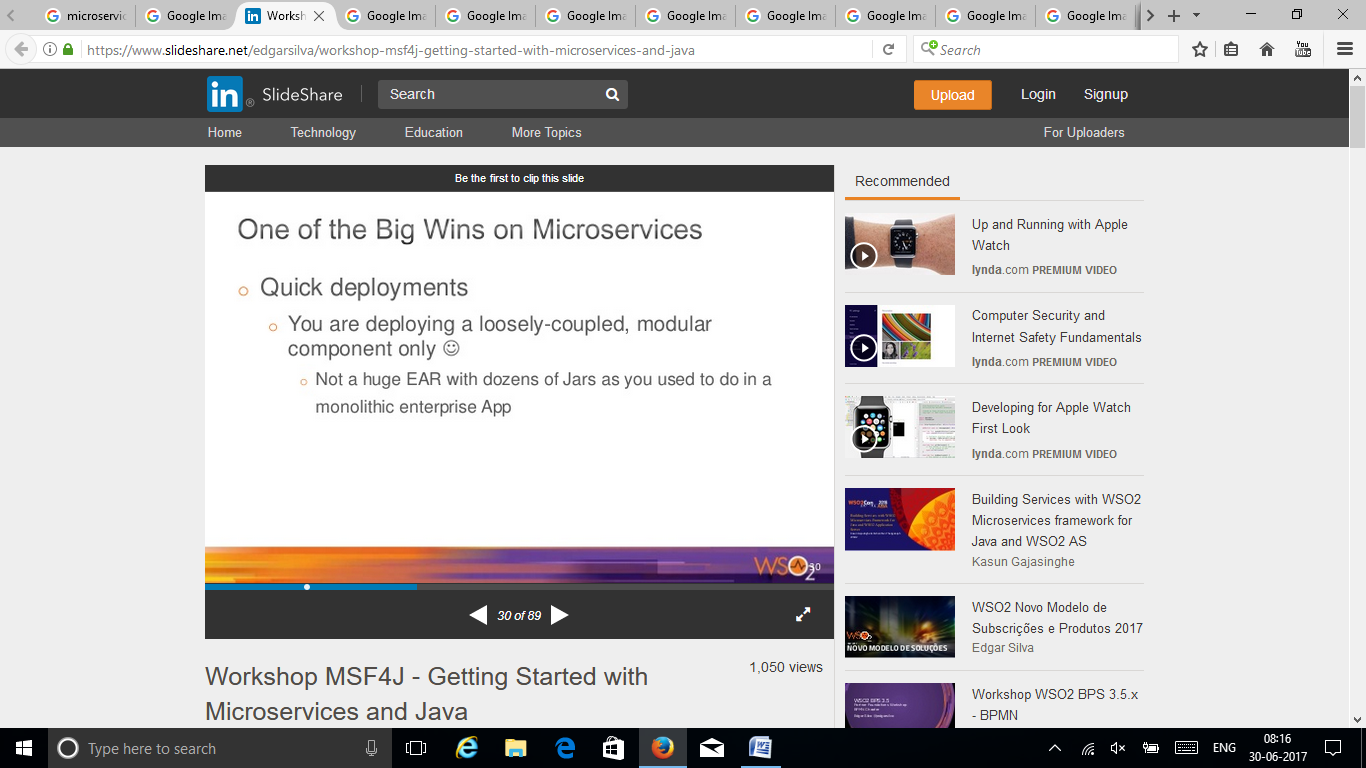
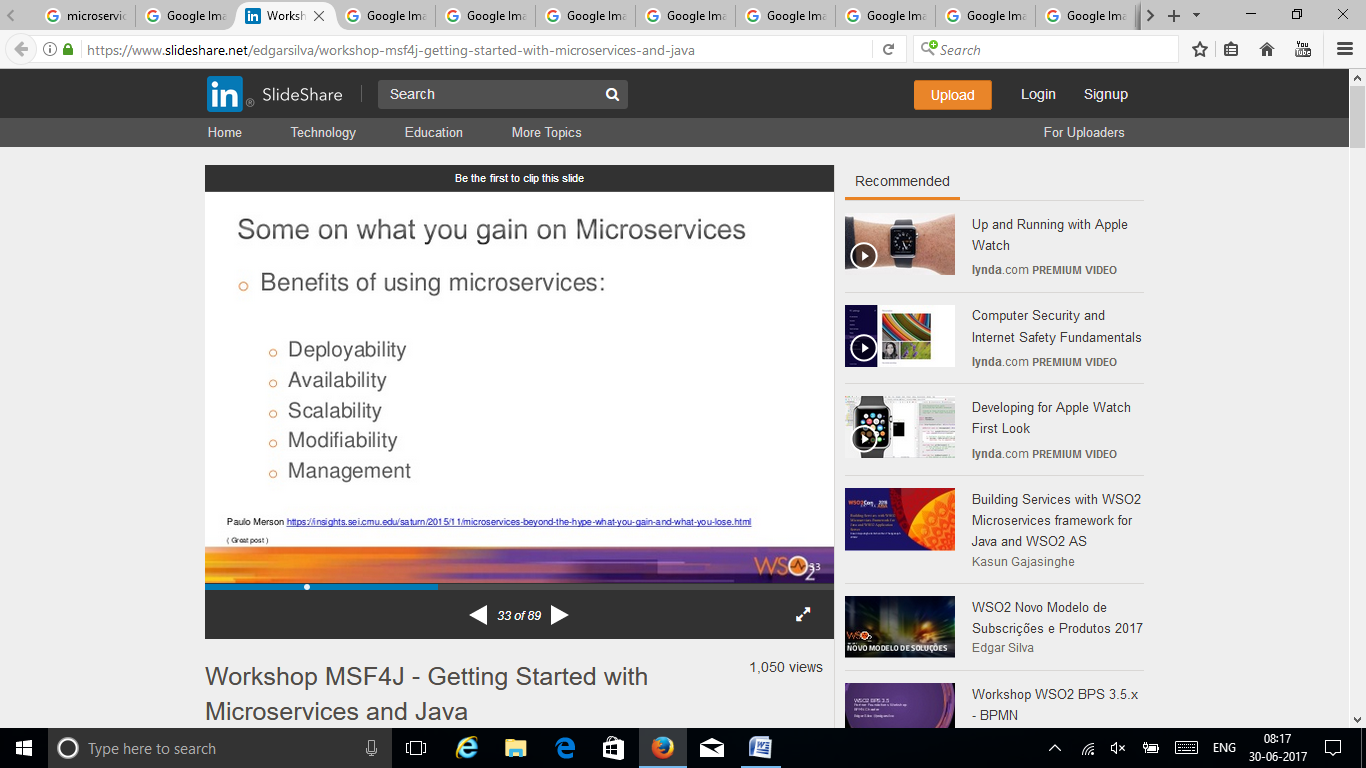


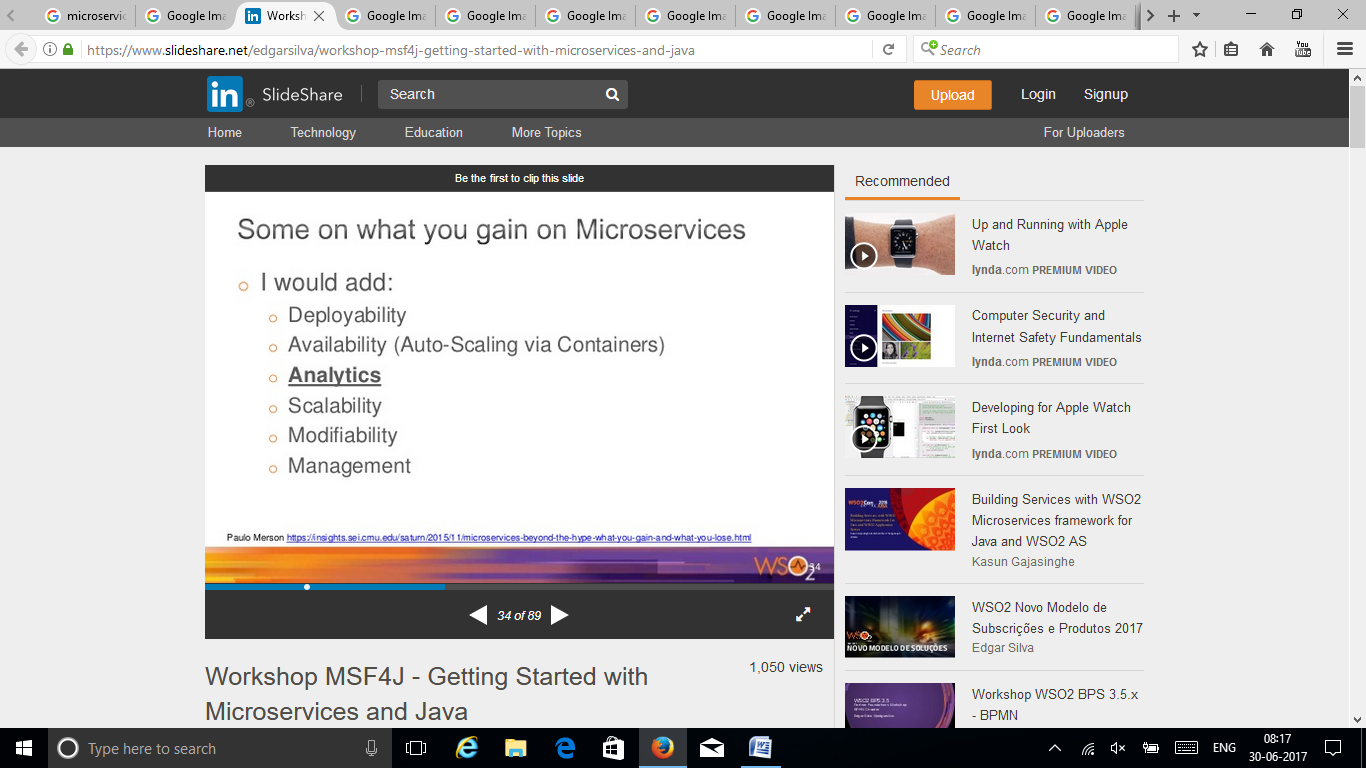
http://modeling-languages.com/automatic-discovery-web-api-specifications/



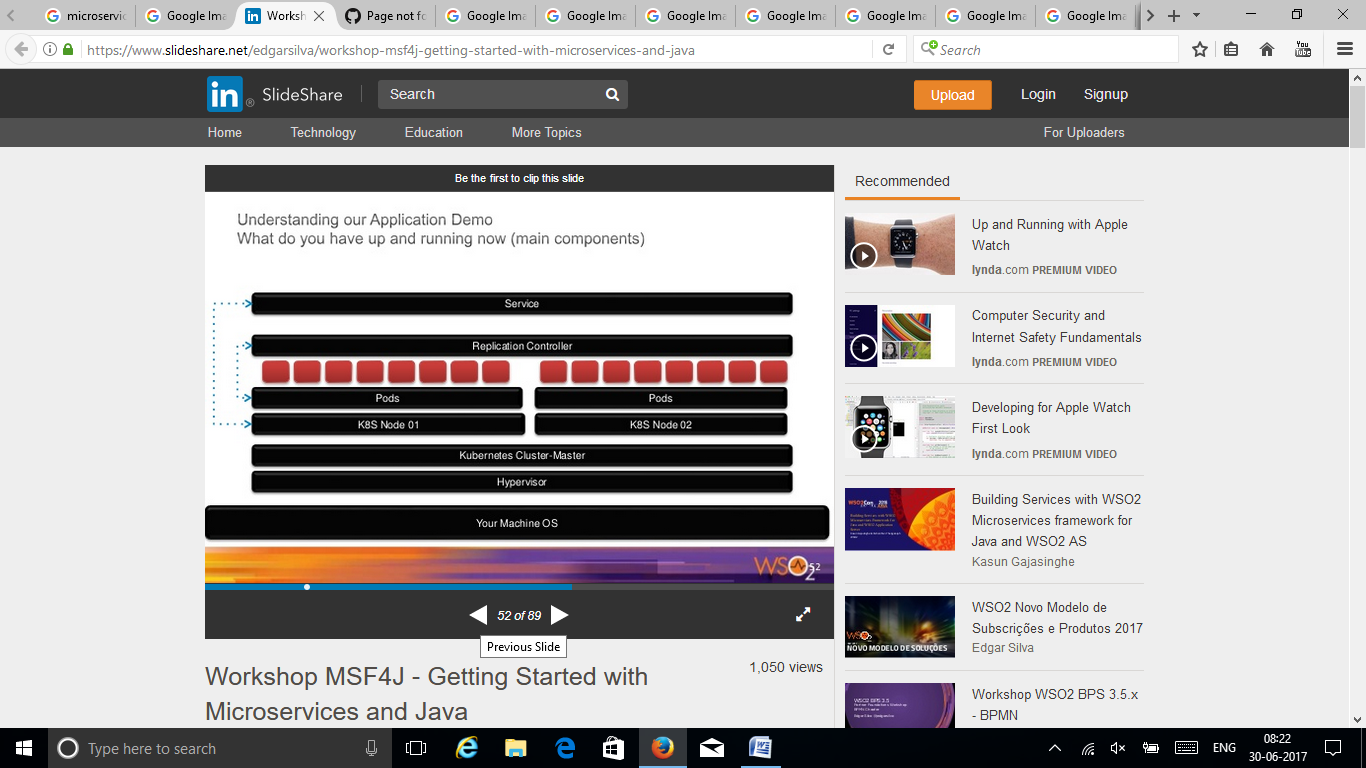


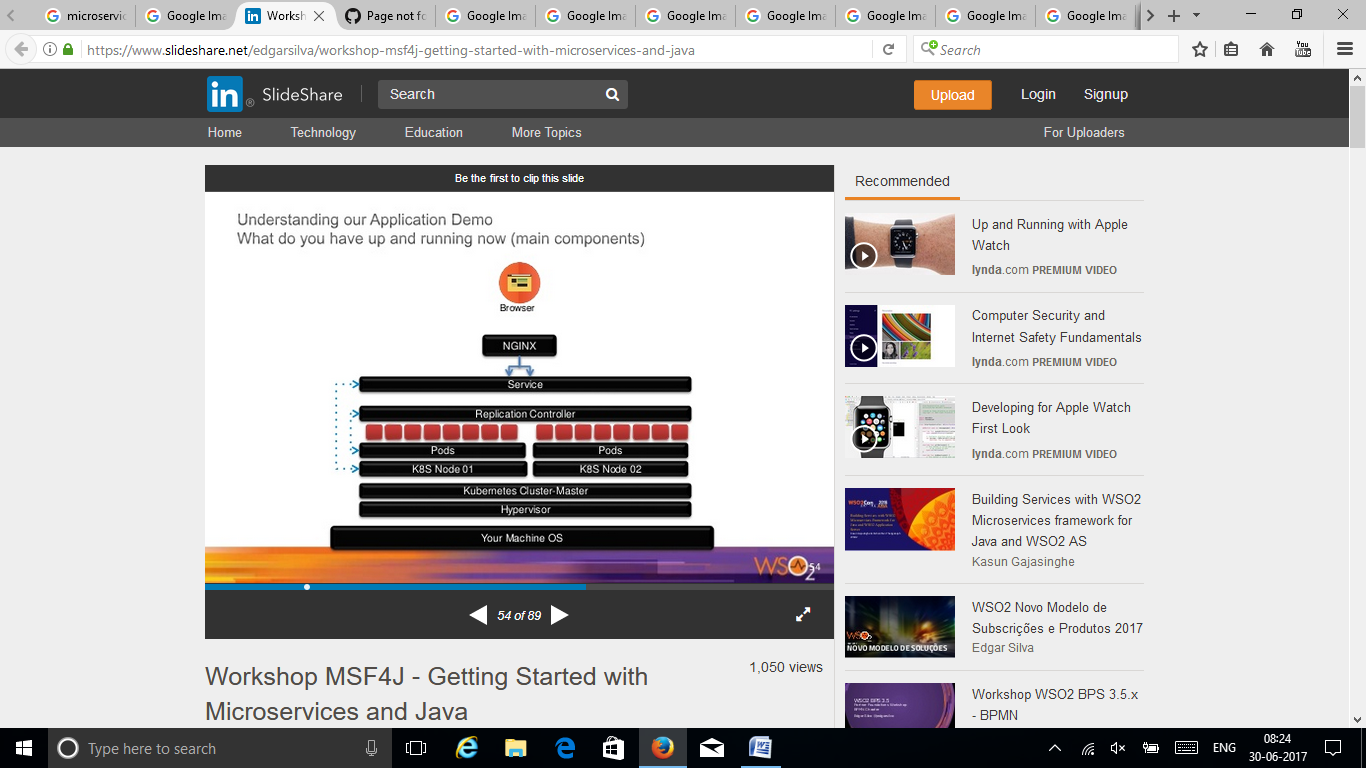


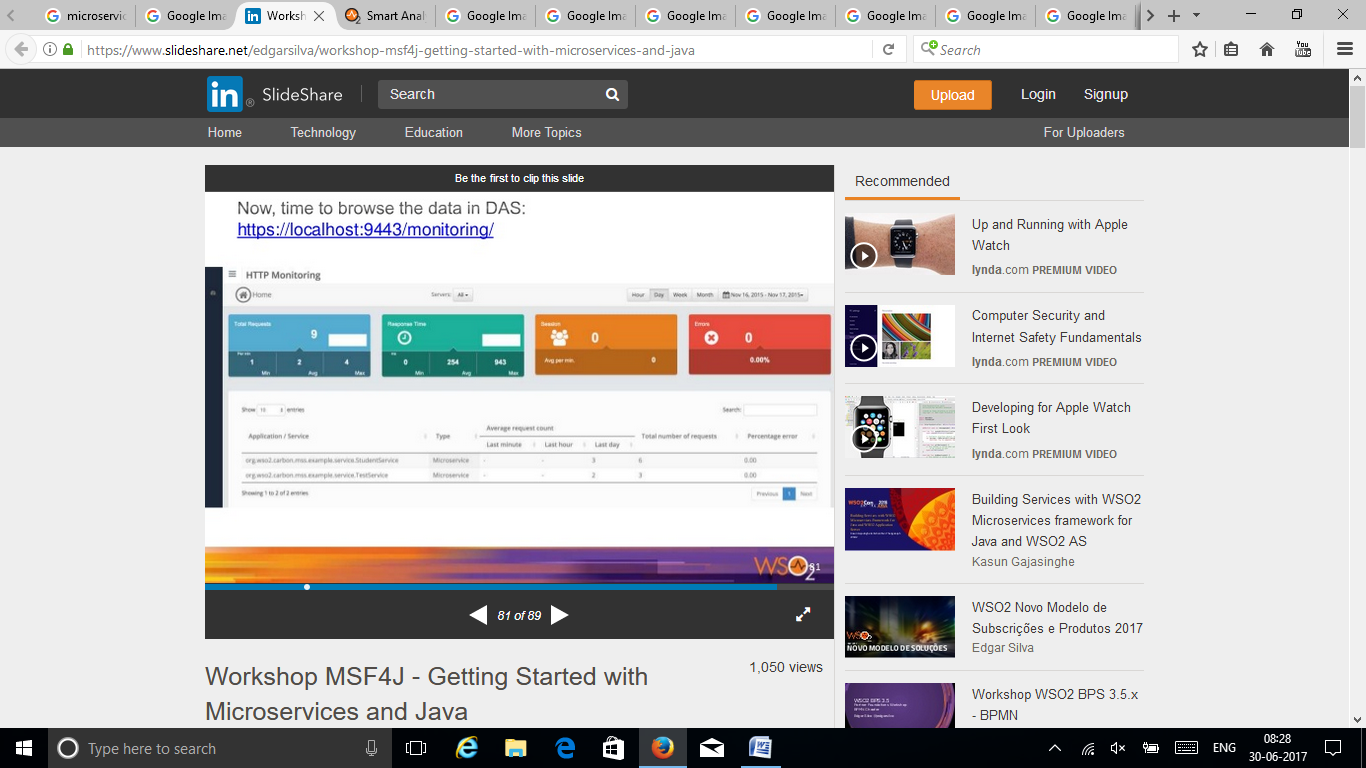


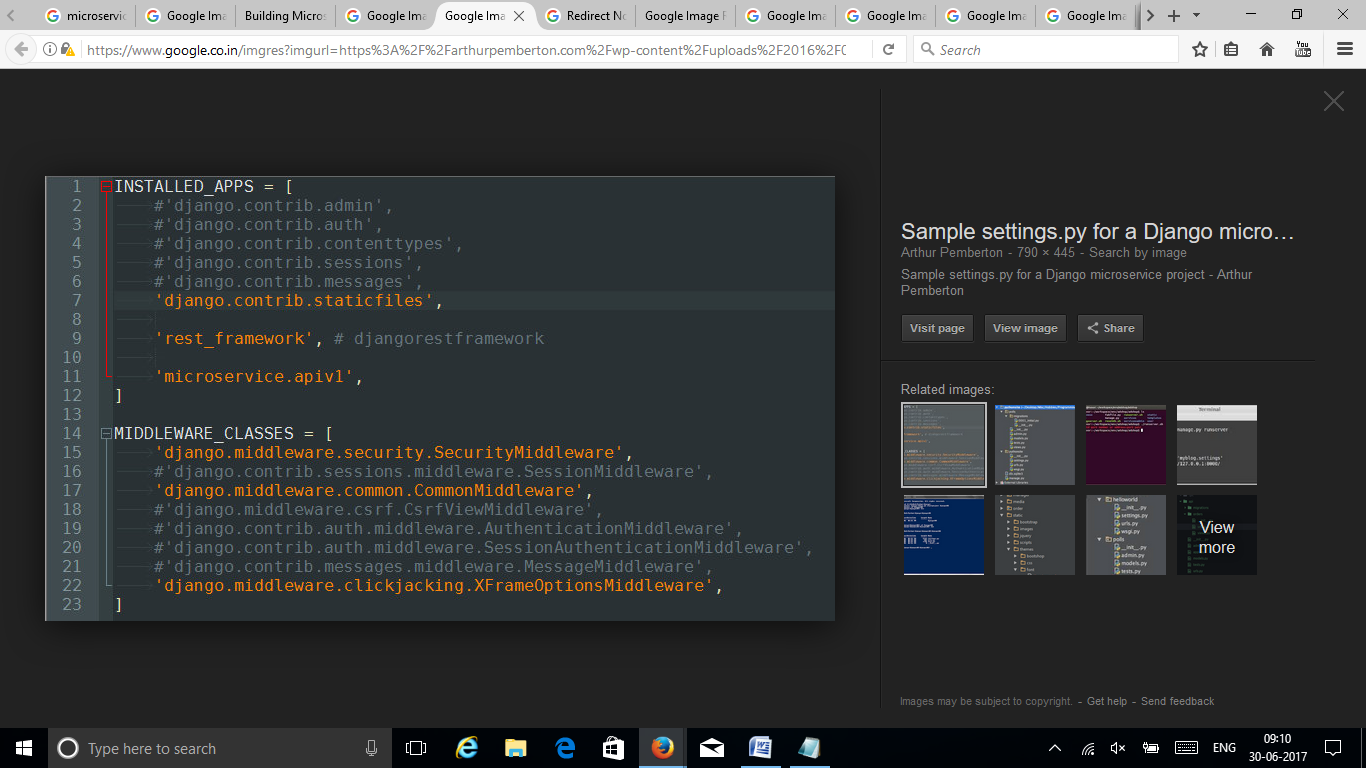


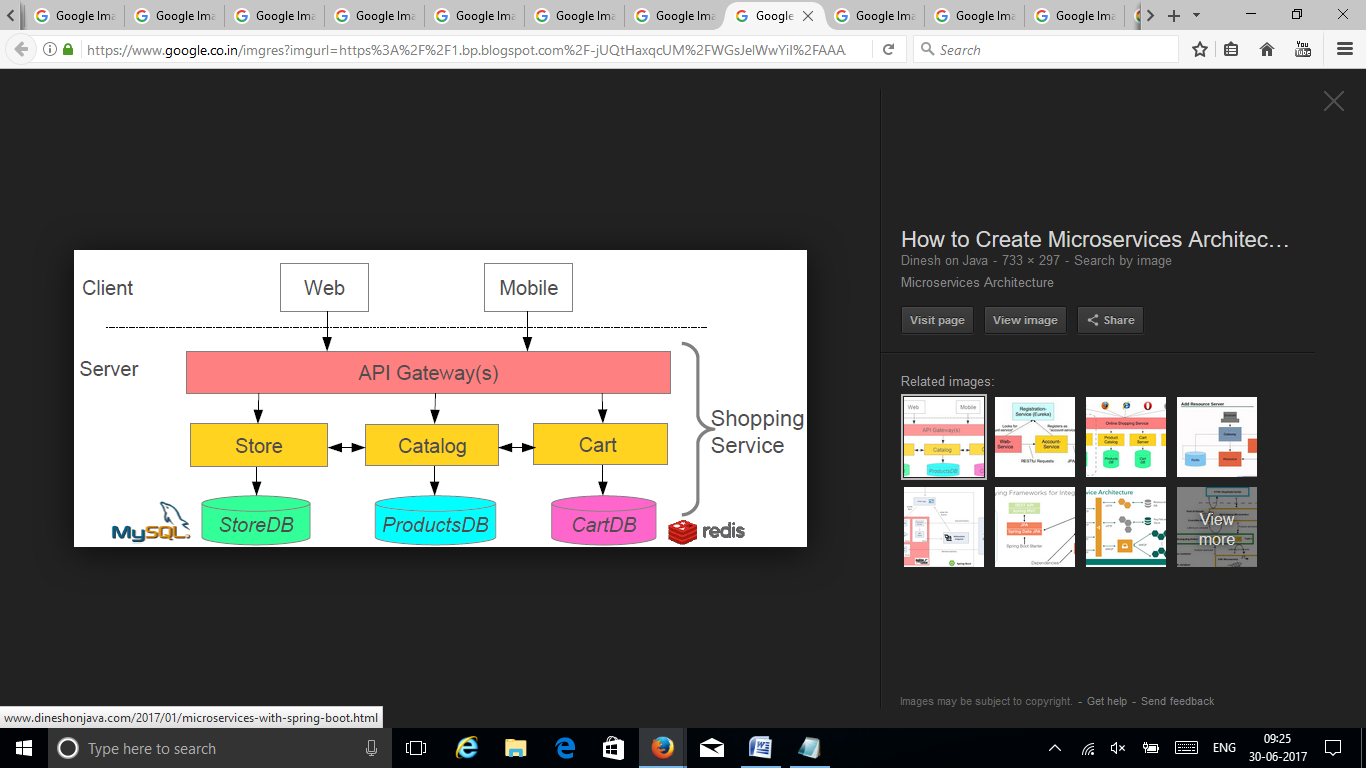


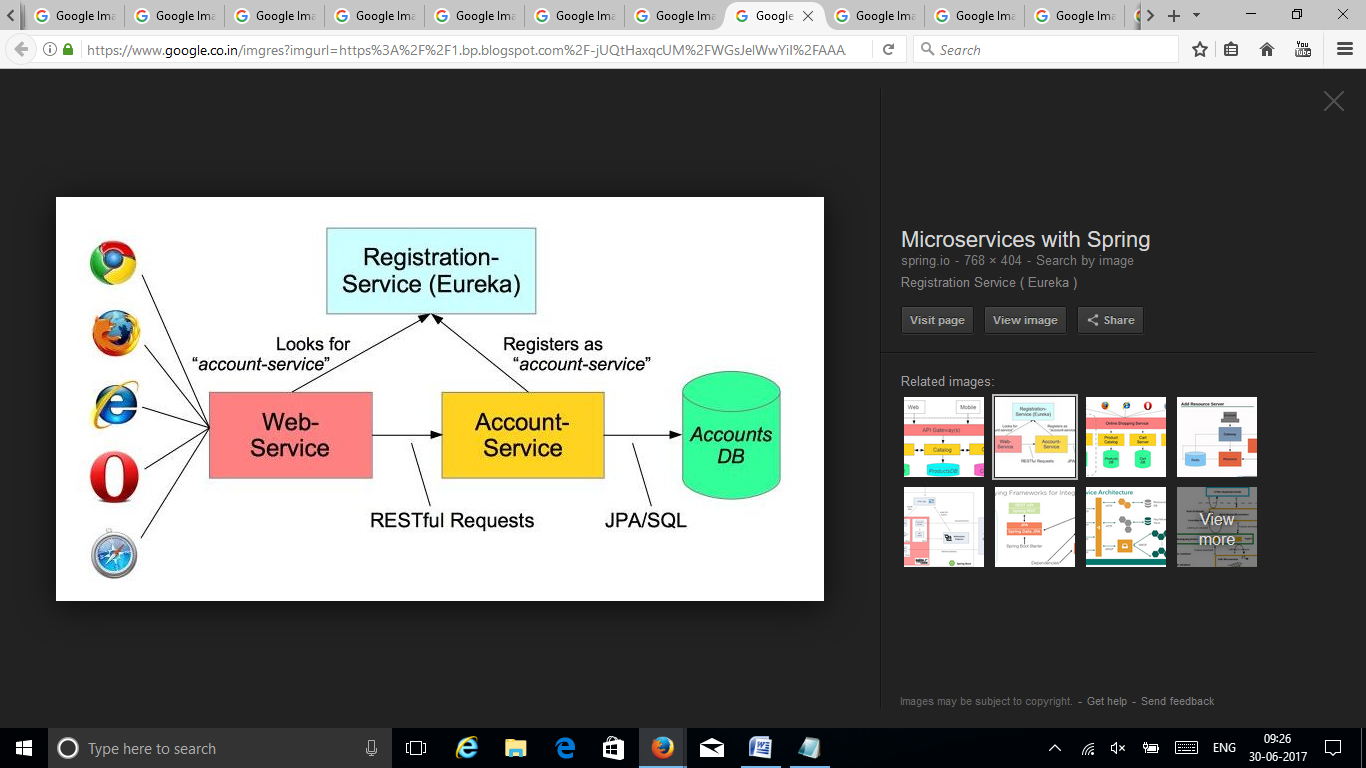


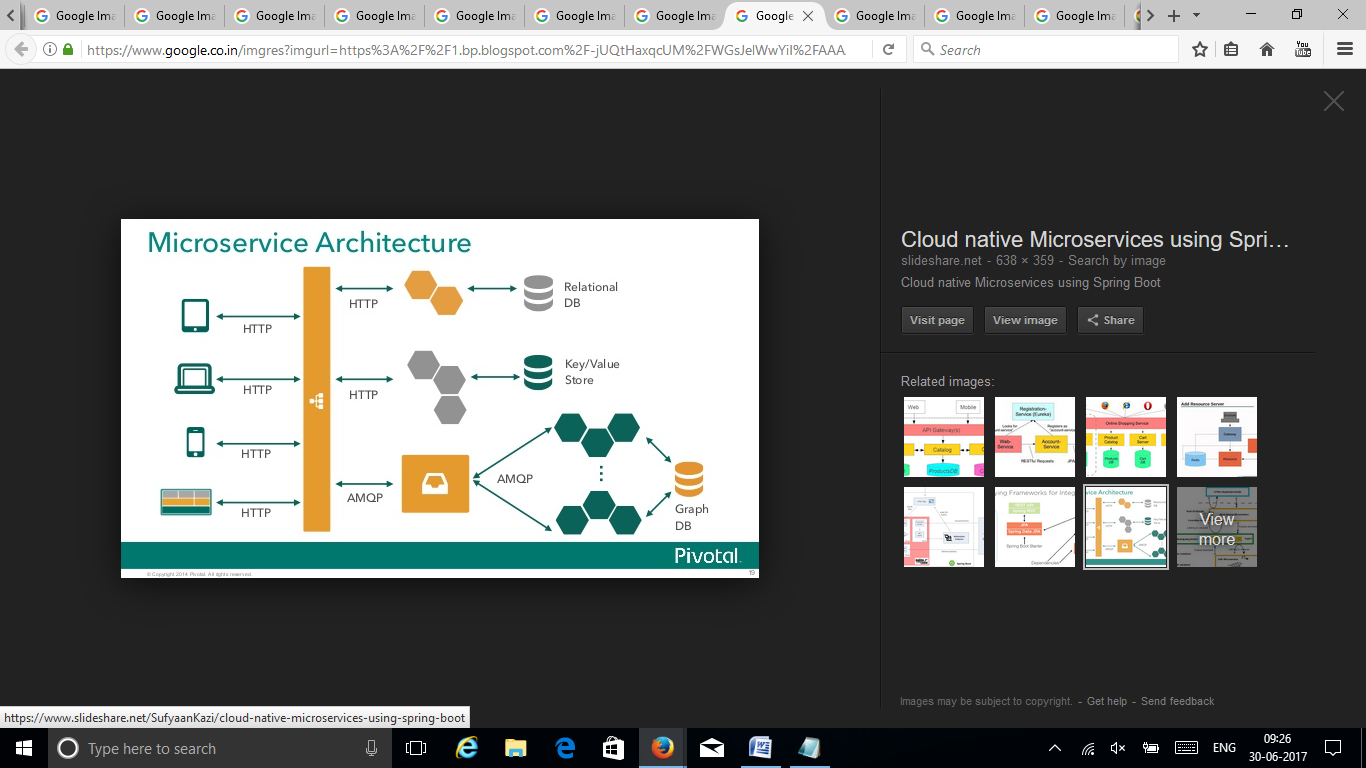


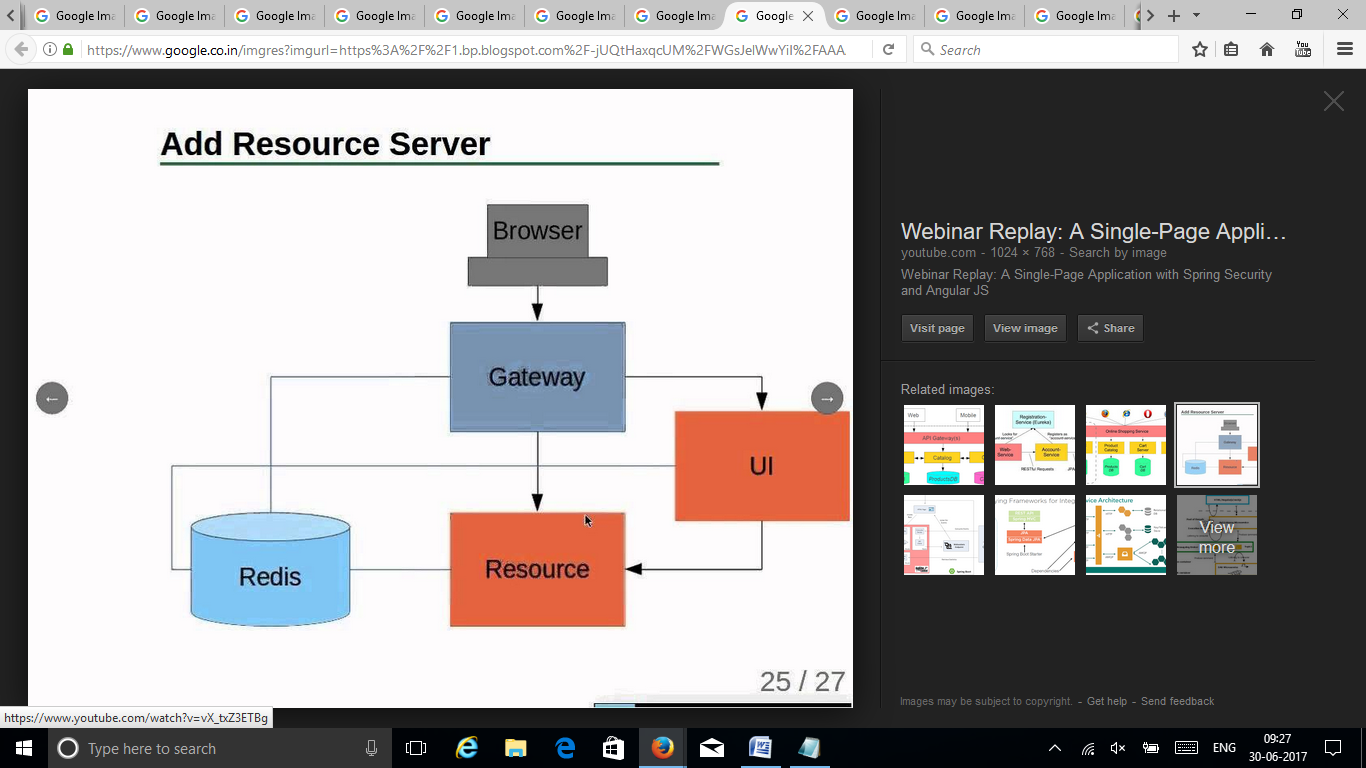


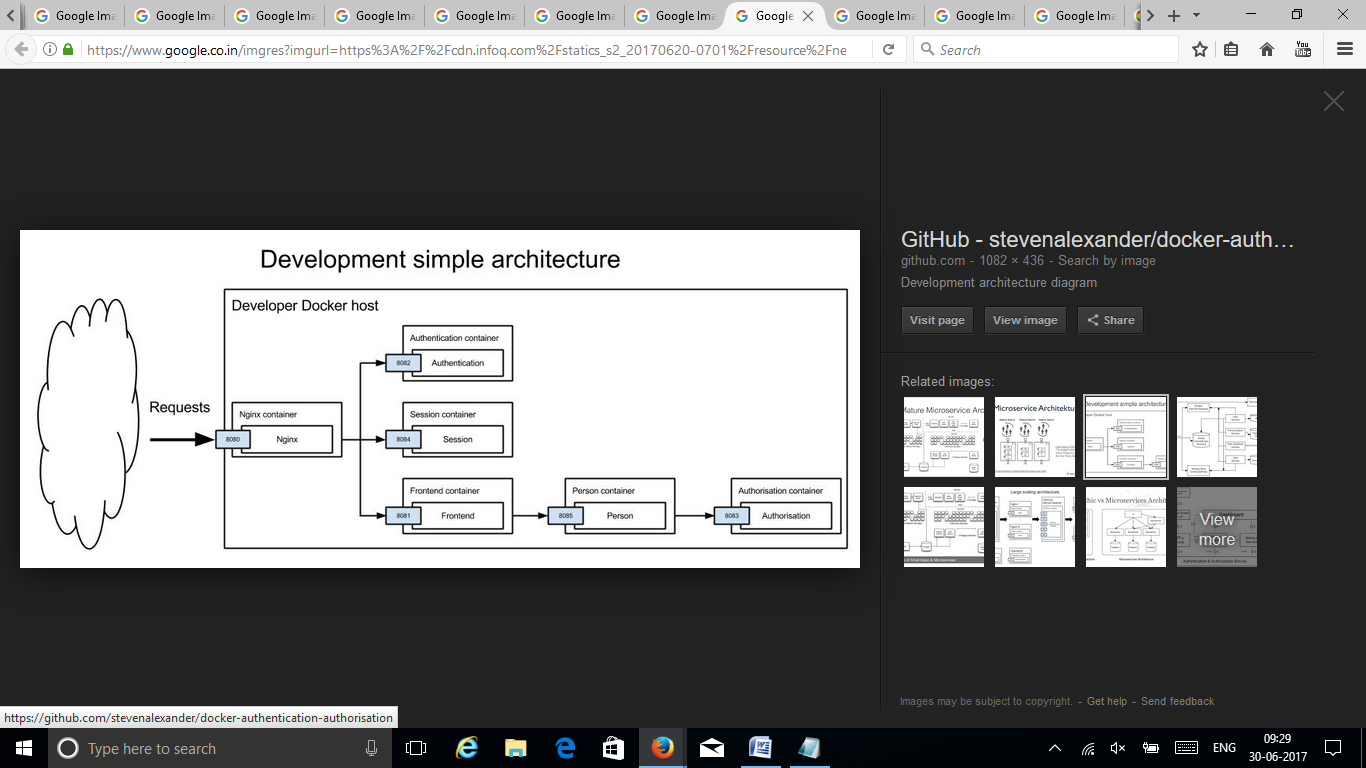




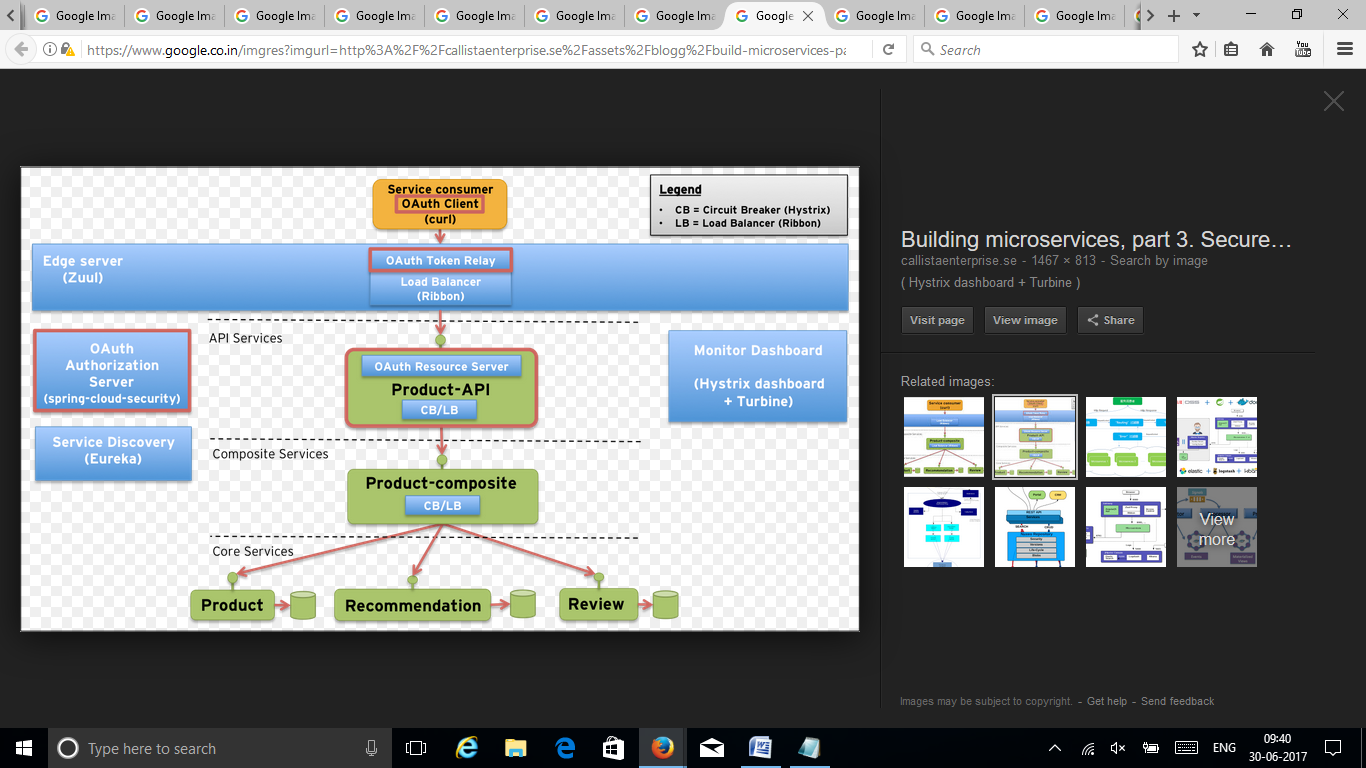


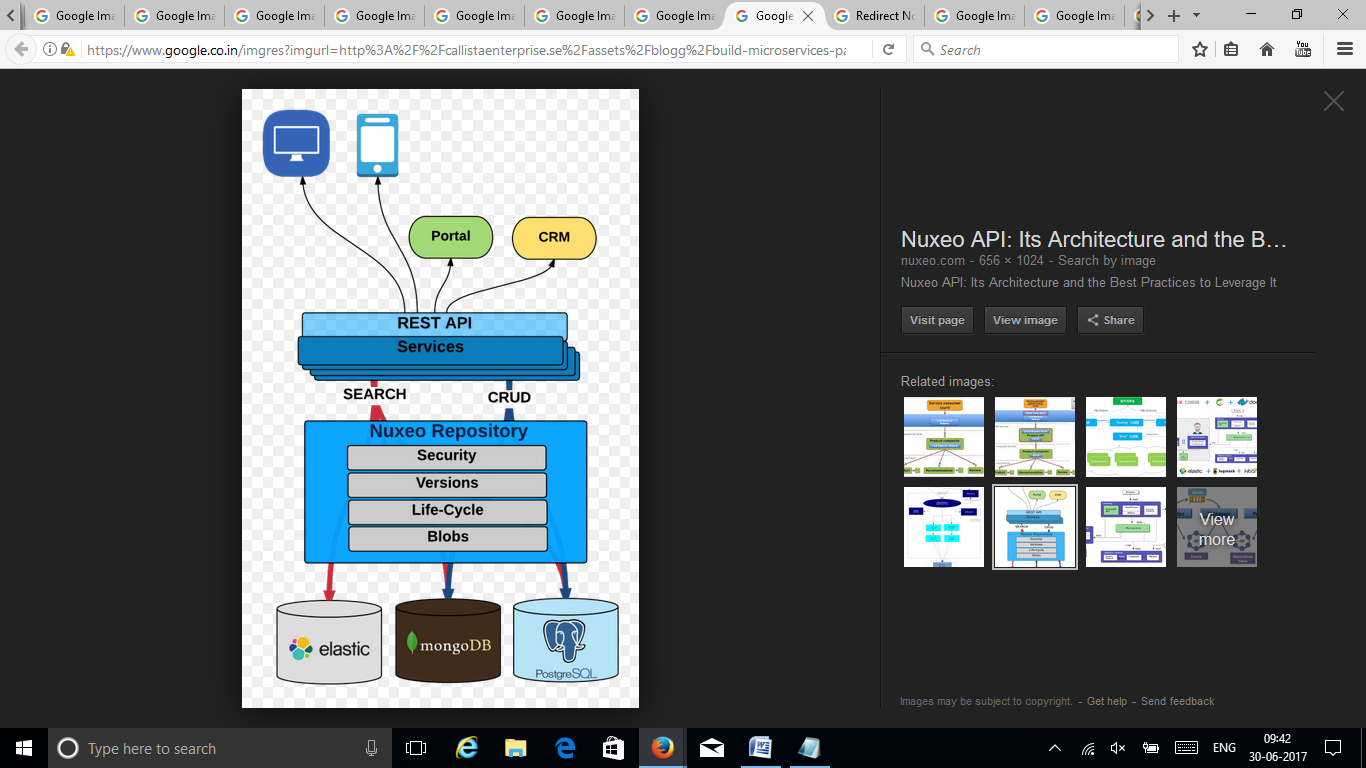


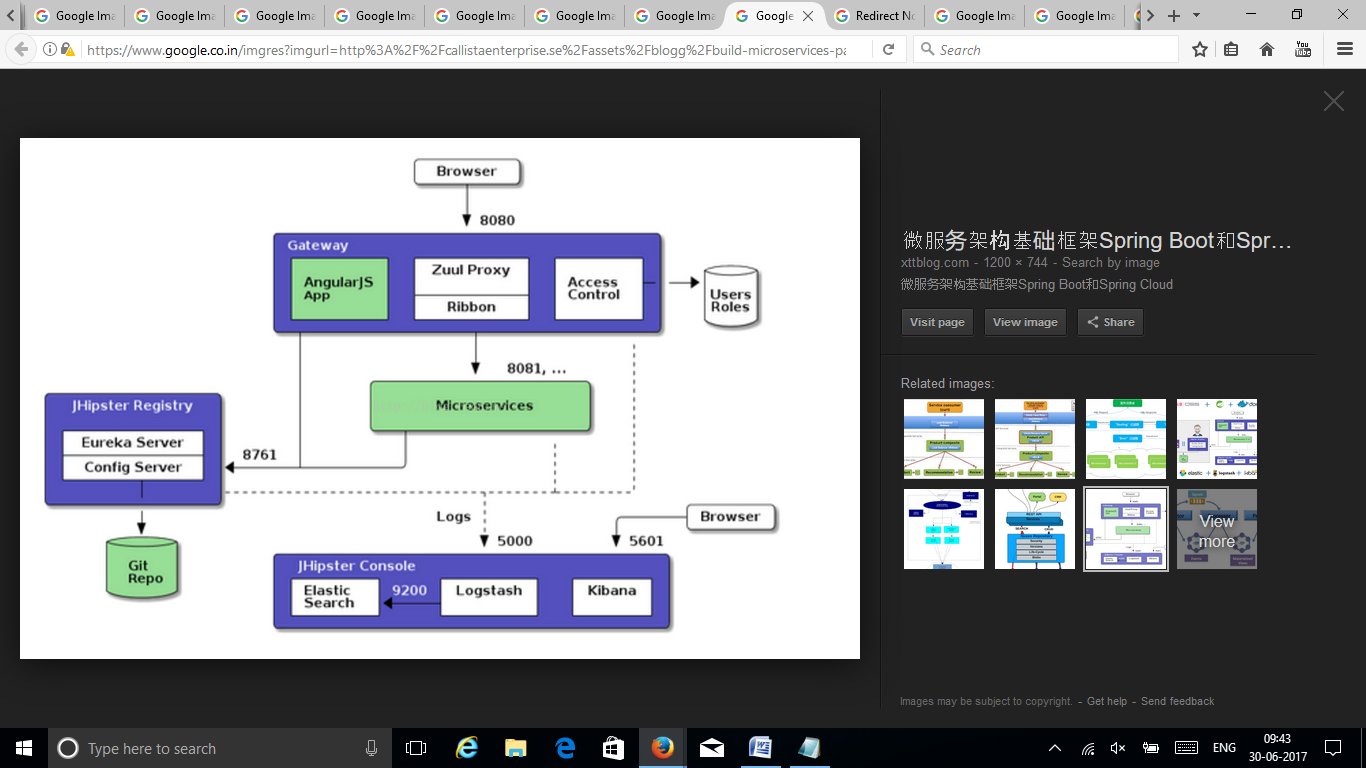


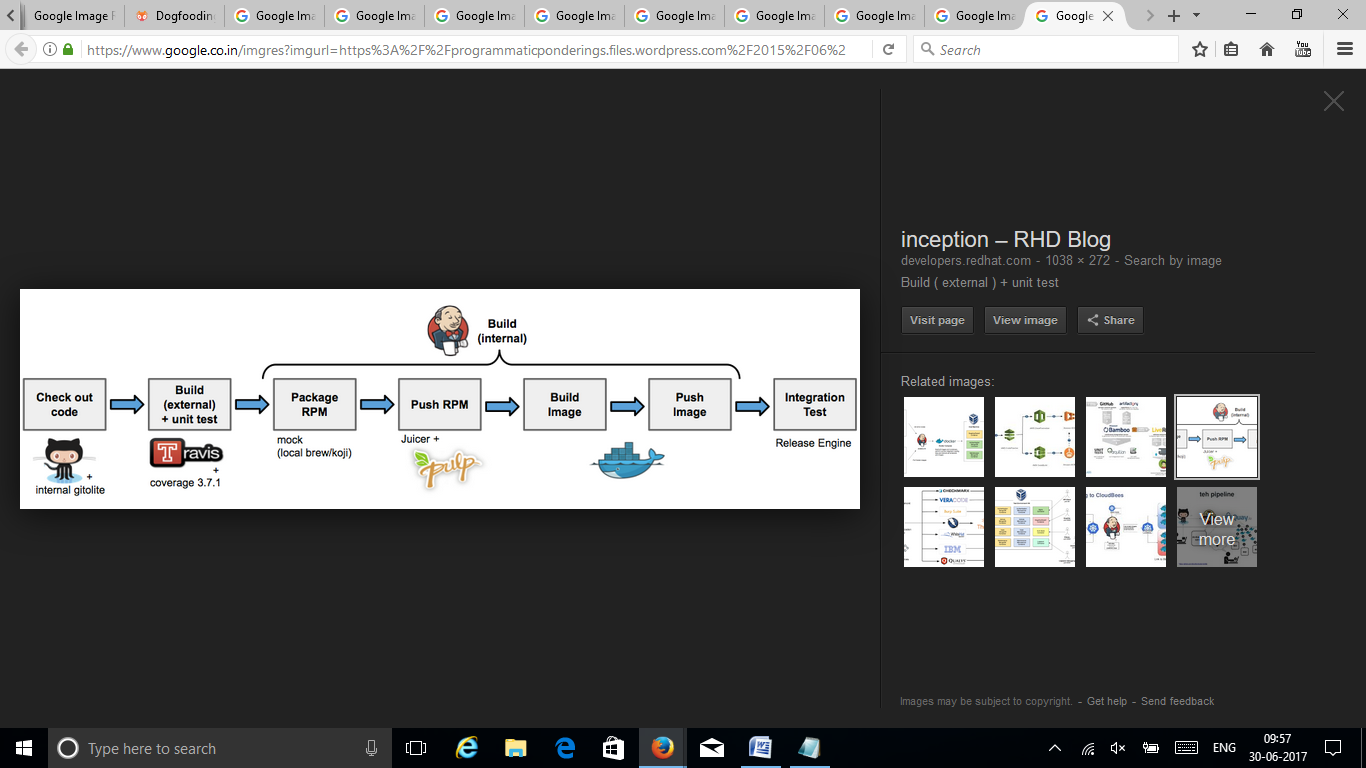


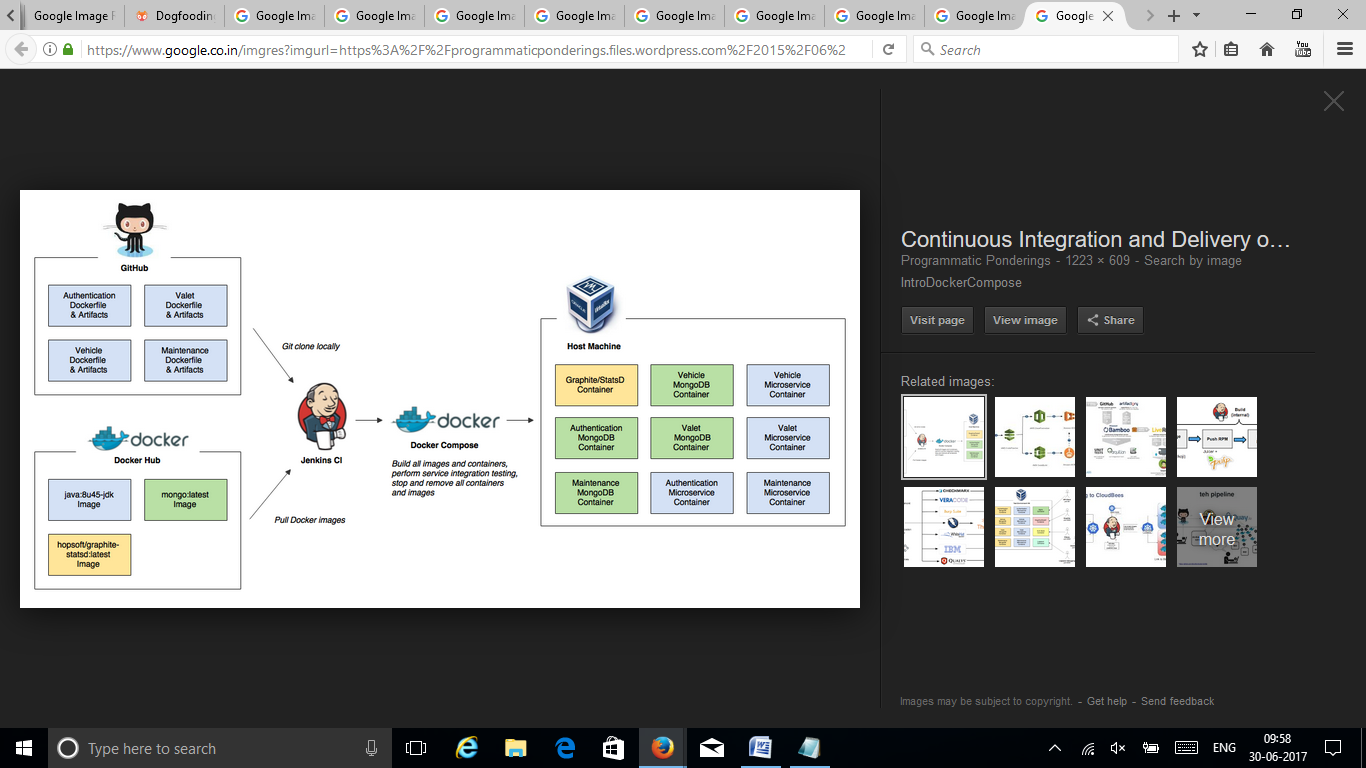












http://michaco.net/blog

http://michaco.net/blog/TipsForUsingSwaggerAndAutorestInAspNetCoreMvcServices

http://michaco.net/blog/ServiceDiscoveryAndHealthChecksInAspNetCoreWithConsul

http://dotnetthoughts.net/creating-your-first-aspnet-core-web-api-with-swashbuckle/

services.AddMvc()

.AddJsonOptions(options =>

{

options.SerializerSettings.ContractResolver = new Newtonsoft.Json.Serialization.DefaultContractResolver();

});

this keeps json object name the same as .net class property.

https://seroter.wordpress.com/2017/03/27/yes-you-can-use-a-single-service-registry-for-net-and-java-microservices/

Performing Discovery From a .NET App

Finally, let’s see how a .NET app would pull a reference from the Eureka registry and use it.

I created a new project based on the ASP.NET Core MVC template. And then I added the Steeltoe package for service discovery.

dotnet new mvc

dotnet add package Steeltoe.Discovery.Client -v 1.0.0-rc2

dotnet restore

With this MVC template, I got some basic scaffolding for a sample website. I just extended this by adding a new view (called “Demo”) and controller method. No content in the method right away.

Just like before, I updated the Startup.cs class by first adding a reference to “Steeltoe.Discovery.Client” and updating the “ConfigureServices” and “Configure” methods.

ASP.NET Core offers some nice dependency injection stuff. So with the code update above, I now had a “DiscoveryClient” object available for any controller or service to use. So, back in the controller, I added a variable for DiscoveryHttpClientHandler. Then I instantiated that object in the controller constructor, and used it in the new controller method to call a Eureka-registered Java service. Note once again that I only needed the registered service name, and the client libraries flipped this to the address/port of my actual service.

public class HomeController : Controller

{

//added for demonstration

DiscoveryHttpClientHandler \_handler;

public HomeController(IDiscoveryClient client) {

\_handler = new DiscoveryHttpClientHandler(client);

}

public IActionResult Demo()

{

HttpClient c = new HttpClient(\_handler, false);

//call service using registered alias

string s = c.GetStringAsync("http://boot-customer-service").Result;

ViewData["Message"] = "Service result is: " + s;

return View();

}

}

Finally, I added a few things to my appsettings.json file so that the Steeltoe client library knew how to behave. I gave the application a name, and told it to \*not\* register itself with Eureka, but only to fetch the registry and cache it locally.

{

"Logging": {

"IncludeScopes": false,

"LogLevel": {

"Default": "Warning"

}

},

"spring": {

"application": {

"name": "dotnet-demo-service-client"

}

},

"eureka": {

"client": {

"serviceUrl": "http://localhost:8761/eureka/",

"shouldRegisterWithEureka": false,

"shouldFetchRegistry": true

},

"instance": {

"hostname": "localhost",

"port": 5001

}

}

}

After that, I started up by ASP.NET Core app, hit the webpage, and saw a result from my Spring Boot service.

https://seroter.wordpress.com/2017/03/27/yes-you-can-use-a-single-service-registry-for-net-and-java-microservices/

That was fun! Some sort of service registry is extremely helpful when adopting a microservices architecture. Instead of using hard-coding references or stale data stores, an always-accurate registry gives you the best chance of surviving in a fluid microservices environment. Now, thanks to Steeltoe, you can use the same registry for your Java, .NET (and even Node.js) services.

Thanks Richard, another great article and that we can now easily register/integrate .Net services is very attractive. I had a quick search but couldnt see anyone offering a SaaS Eureka service which might be nice rather than self-hosted/containered etc. Anyone thought of provisioning a hyper-scale Eureka for public API endpoints, and a general pattern of self-registration plus a standard security/usage contract for clients..? That would be funky… Will stop getting ahead of myself and see if we can adopt this for our internal endpoints

Steeltoe brings the power of resilient, scalable microservices to .NET developers, without all the headache.

https://steeltoe.io/

What is Steeltoe?

Pivotal Cloud Foundry Steeltoe logo v2

Steeltoe was built to simplify the adoption of microservices for .NET developers. It provides the access to the pool of Spring Cloud and NetflixOSS tools that power cloud-native Java apps. What is more, the new toolkit promises first-class integration with Pivotal Cloud Foundry (PCF).

It also simplifies step-by-step porting of a .NET 4.x app to .NET Core microservices by making decomposition of monolithic architectures much easier. The transition results in a heterogeneous environment for the app: some of its parts may run on the Windows platform, while others can be deployed to Linux containers running on Cloud Foundry.

This post from Pivotal sheds more light on the topic, while this one demonstrates how to use Steeltoe for ASP.NET 4.x apps.

https://www.altoros.com/blog/enabling-dotnet-core-microservices-with-steeltoe-and-pivotal-cloud-foundry/

So, we have seen how .NET developers can leverage Steeltoe to enable .NET Core apps to consume Spring Cloud services right from Pivotal Cloud Foundry. With thoroughly documented components and a relatively flat learning curve, Steeltoe makes a good job of facilitating development and deployment of microservices-based .NET apps.

https://www.soapui.org/rest-testing-discovery/rest-discovery---api-with-internal-browser.html

https://www.simple-talk.com/dotnet/net-development/visual-studio-2017-swagger-building-documenting-web-apis/

Change the ‘ValuesController’ class for the code below. We are completing the PUT action and storing the list inside Application memory area.

public class ValuesController : ApiController

{

private string[] lista

{

get

{

if (System.Web.HttpContext.Current.Application[<em>"lista"</em>]==null)

{

System.Web.HttpContext.Current.Application[<em>"lista"</em>] =

new string[] { <em>"value1"</em>, <em>"value2"</em> };

}

return

(string[])System.Web.HttpContext.Current.Application[<em>"lista"</em>];

}

}

public IEnumerable<string> Get()

{

return lista;

}

public string Get(int id)

{

return lista[id];

}

public void Post([FromBody]string value)

{

}

public void Put(int id, [FromBody]string value)

{

lista[id] = value;

}

public void Delete(int id)

{

}

}

Execute the application and analyze the result

Click ‘Start’ button in the toolbar to execute the application

On the web page just opened, click ‘API’ link in the top menu. You will notice the API operations, basically a CRUD with two GET’s, one POST, PUT and DELETE

Include SwashBuckle nugget package in the web API project

https://blog.giantswarm.io/dogfooding-our-own-website-on-giant-swarm/

http://www.developer.com/open/building-microservices-with-open-source-technologies.html

To get a list of controllers in RC2, just get ApplicationPartManager from DependencyInjection and do this:

ApplicationPartManager appManager = <FROM DI>;

var controllerFeature = new ControllerFeature();

appManager.PopulateFeature(controllerFeature);

foreach(var controller in controllerFeature.Controllers)

{

...

}

https://stackoverflow.com/questions/36680933/discovering-generic-controllers-in-asp-net-core

https://blogs.msdn.microsoft.com/cesardelatorre/2016/12/05/generating-swagger-description-metadata-from-your-asp-net-core-web-apis-with-swashbuckle/

https://www.codeproject.com/Articles/1078249/RESTful-Web-API-Help-Documentation-using-Swagger-U

https://www.slideshare.net/SteveSfartz/emn-workshop-restfull-web-apis-build-document-manage-slideshare

npm install consul