**Using Graph Analysis to Design Microservice Architectures in the Cloud**

This interactive Neo4j graph tutorial covers how to use graph analysis to find software modules that are highly centralized, making good candidates to be decomposed into microservices.

**Table of Contents**

* **Introduction**
  + [Introduction to Problem](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_introduction_to_problem)
* **Scenario**
  + [Explanation of Scenario](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_explanation of scenario)
* **Data Model**
  + [Online Storefront Domain Model](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_online_storefront_domain_model)
  + [Application Deployment Model](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_application_deployment_model)
  + [Dependency Graph](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_dependency_graph)
* **Database Setup**
  + [Sample Dataset](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_sample_dataset)
* **Cypher Queries**
  + [Entity Link Analysis](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#entity_link_analysis)
  + [Which resources have the highest story centrality?](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_which_resources_have_the_highest_story_centrality)
  + [Which service components are most depended upon?](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_which_service_components_are_most_depended_upon)
    - **Scenarios**
      * [Accounting Service](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_accounting_service)
      * [Products Microservice](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_products_microservice)
      * [Orders Microservice](https://gist.github.com/kbastani/4f1e5fe25088209dcc26#_orders_microservice) '''

**Introduction to Problem**

Jumping head first into microservices is a major commitment. Monolithic platforms will have highly centralized components that will gain more mass as new microservices are born. It is important to first map and analyze these connections to understand which services in an SOA are becoming more depended upon. These services with the highest centrality need to be decomposed.

**Explanation of Scenario**

Microservices are an extension of SOA principles that are better suited for agile software development. A microservice architecture usually starts from the decomposing of a monolithic application into services that are cheaper to evolve and easier to throw away. The guiding theme behind this movement is to decentralize change management and reduce conflicts that tend to cause roadblocks for agile teams in an SOA-based platform.

When companies on an SOA add new features to their platform, there tends to be a fair amount of conflicts between service teams. Certain services in the SOA become more relied upon by other services or applications in the platform.

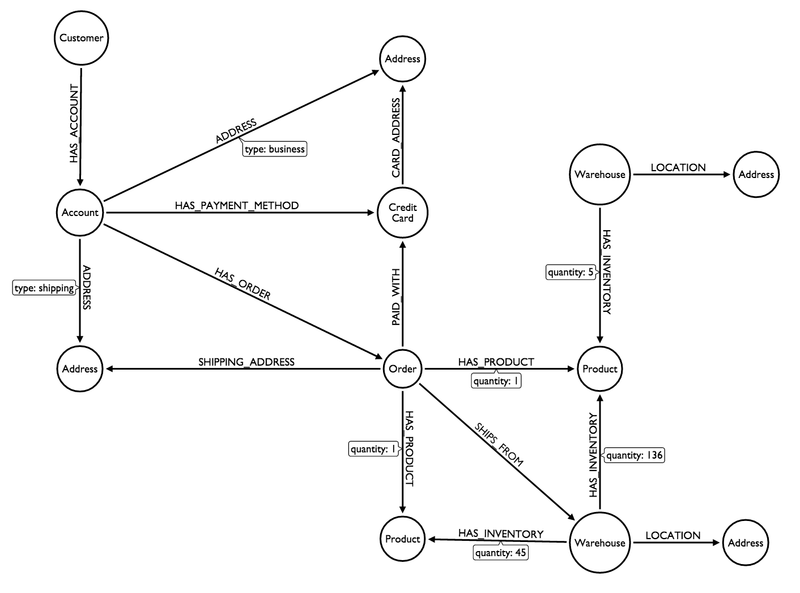
Ideally, microservices should both be liberal in the resources they request of others and conservative in the resources they choose to expose. By striking this balance, services will strive to become more modular as they decompose and evolve.

**Data Models**

Below is a set of models that describe the online storefront of a company named Acmezon, Inc.

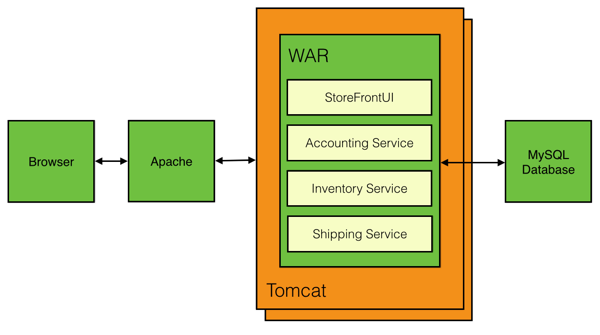
**Online Storefront Domain Model**

The graph data model below shows the domain model of Acmezon, Inc. as a labeled property graph. The data model contains a set of resources, indicated as node labels, that will be owned by a set of service modules within Acmezon’s monolithic application.

[](https://camo.githubusercontent.com/7ad3af716a7211da0f239c7c37b8531437f798ea/687474703a2f2f692e696d6775722e636f6d2f3477347845674e2e706e67)

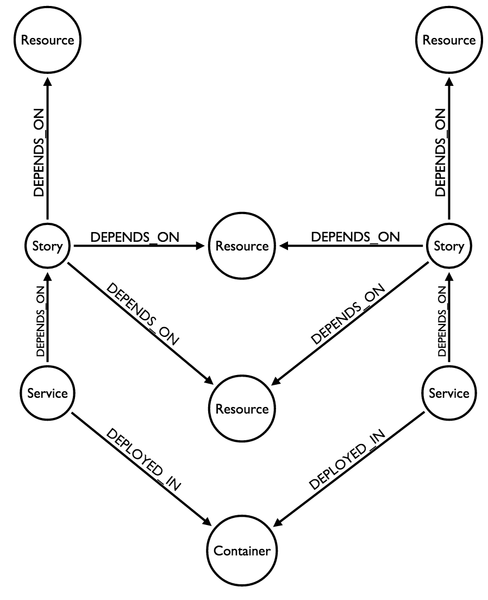
**Application Deployment Model**

The diagram below is Acmezon’s application deployment model, as described by microservice guru [Chris Richardson](http://microservices.io/patterns/monolithic.html).

[](https://camo.githubusercontent.com/cc88bac823c3f3876710ee519d42fbe9c7ed1d27/687474703a2f2f692e696d6775722e636f6d2f58645732645a462e706e67)

**Dependency Graph**

Below you will find an example graph data model of the service dependencies shared between containers, services, resources, and user stories that describe product features.

[](https://camo.githubusercontent.com/72cbef4831e93bba769876df817396e474b542ee/687474703a2f2f692e696d6775722e636f6d2f4b6661384a666e2e706e67)

In order to generate a rich dataset, I’m introducing the non-functional concept of a user story. User stories do well to group together a set of features. These feature groups act as a good boundary criteria for determining how to make components more modular.

The relationships between concepts in our dependency graph are driven by the following rules:

* User stories depend on domain resources
* Domain resources are owned by services
* Services are managed by teams
* A service belongs to a deployment container

**Sample Dataset**

The sample dataset below is an example dependency graph for Acmezon’s monolithic application.

// Teams

CREATE (servicesTeam:Team { name: "Services team" })

// Containers

CREATE (container:Container { name: "Tomcat WAR" })

// Services

CREATE (accountingService:Service { name: "Accounting Service"})

CREATE (inventoryService:Service { name: "Inventory Service"})

CREATE (shippingService:Service { name: "Shipping Service"})

// Resources

CREATE (customers:Resource { name: "Customers"})

CREATE (accounts:Resource { name: "Accounts"})

CREATE (warehouseAddresses:Resource { name: "Warehouse Addresses"})

CREATE (shippingAddresses:Resource { name: "Shipping Addresses"})

CREATE (billingAddresses:Resource { name: "Billing Addresses"})

CREATE (orders:Resource { name: "Orders"})

CREATE (products:Resource { name: "Products"})

CREATE (warehouses:Resource { name: "Warehouses"})

CREATE (creditCards:Resource { name: "Credit Cards"})

// Stories

CREATE (story1:Story { name: "Customer registration"})

CREATE (story2:Story { name: "Account details"})

CREATE (story3:Story { name: "Payment details"})

CREATE (story4:Story { name: "Order history"})

CREATE (story5:Story { name: "Product catalog"})

CREATE (story6:Story { name: "Warehouse information"})

CREATE (story7:Story { name: "Inventory management"})

CREATE (story8:Story { name: "Supply chain management"})

CREATE (story9:Story { name: "Shipping logistics"})

CREATE (story10:Story { name: "Shipping history"})

CREATE (story11:Story { name: "Shipping notifications"})

CREATE (story12:Story { name: "Backordering"})

CREATE (story13:Story { name: "Account management"})

CREATE (story14:Story { name: "Shopping cart"})

CREATE (story15:Story { name: "Product search"})

CREATE (story16:Story { name: "Checkout"})

// Connect customer registration story to resources

CREATE (story1)-[:DEPENDS\_ON]->(customers)

CREATE (story1)-[:DEPENDS\_ON]->(accounts)

// Connect account details story to resources

CREATE (story2)-[:DEPENDS\_ON]->(accounts)

CREATE (story2)-[:DEPENDS\_ON]->(customers)

CREATE (story2)-[:DEPENDS\_ON]->(shippingAddresses)

CREATE (story2)-[:DEPENDS\_ON]->(billingAddresses)

CREATE (story2)-[:DEPENDS\_ON]->(orders)

CREATE (story2)-[:DEPENDS\_ON]->(creditCards)

// Connect payment details story to resources

CREATE (story3)-[:DEPENDS\_ON]->(accounts)

CREATE (story3)-[:DEPENDS\_ON]->(billingAddresses)

CREATE (story3)-[:DEPENDS\_ON]->(creditCards)

// Connect order history story to resources

CREATE (story4)-[:DEPENDS\_ON]->(orders)

CREATE (story4)-[:DEPENDS\_ON]->(products)

CREATE (story4)-[:DEPENDS\_ON]->(shippingAddresses)

CREATE (story4)-[:DEPENDS\_ON]->(billingAddresses)

CREATE (story4)-[:DEPENDS\_ON]->(creditCards)

CREATE (story4)-[:DEPENDS\_ON]->(accounts)

CREATE (story4)-[:DEPENDS\_ON]->(warehouses)

// Connect product catalog story to resources

CREATE (story5)-[:DEPENDS\_ON]->(products)

// Connect warehousing information story to resources

CREATE (story6)-[:DEPENDS\_ON]->(warehouses)

CREATE (story6)-[:DEPENDS\_ON]->(orders)

CREATE (story6)-[:DEPENDS\_ON]->(products)

CREATE (story6)-[:DEPENDS\_ON]->(warehouseAddresses)

// Connect inventory management information to resources

CREATE (story7)-[:DEPENDS\_ON]->(products)

CREATE (story7)-[:DEPENDS\_ON]->(warehouses)

// Connect supply chain management to resources

CREATE (story8)-[:DEPENDS\_ON]->(warehouses)

CREATE (story8)-[:DEPENDS\_ON]->(products)

CREATE (story8)-[:DEPENDS\_ON]->(warehouseAddresses)

// Connect shipping logistics story to resources

CREATE (story9)-[:DEPENDS\_ON]->(warehouseAddresses)

CREATE (story9)-[:DEPENDS\_ON]->(shippingAddresses)

CREATE (story9)-[:DEPENDS\_ON]->(orders)

CREATE (story9)-[:DEPENDS\_ON]->(products)

CREATE (story9)-[:DEPENDS\_ON]->(warehouses)

// Connect shipping history story to resources

CREATE (story10)-[:DEPENDS\_ON]->(shippingAddresses)

CREATE (story10)-[:DEPENDS\_ON]->(orders)

CREATE (story10)-[:DEPENDS\_ON]->(accounts)

// Connect shipping notifications to resources

CREATE (story11)-[:DEPENDS\_ON]->(orders)

CREATE (story11)-[:DEPENDS\_ON]->(shippingAddresses)

// Connect backordering story to resources

CREATE (story12)-[:DEPENDS\_ON]->(warehouses)

CREATE (story12)-[:DEPENDS\_ON]->(products)

// Connect account management story to resources

CREATE (story13)-[:DEPENDS\_ON]->(accounts)

CREATE (story13)-[:DEPENDS\_ON]->(customers)

CREATE (story13)-[:DEPENDS\_ON]->(shippingAddresses)

CREATE (story13)-[:DEPENDS\_ON]->(billingAddresses)

CREATE (story13)-[:DEPENDS\_ON]->(creditCards)

// Connect shopping cart story to resources

CREATE (story14)-[:DEPENDS\_ON]->(products)

CREATE (story14)-[:DEPENDS\_ON]->(orders)

CREATE (story14)-[:DEPENDS\_ON]->(accounts)

CREATE (story14)-[:DEPENDS\_ON]->(customers)

CREATE (story14)-[:DEPENDS\_ON]->(warehouses)

// Connect product search story to resources

CREATE (story15)-[:DEPENDS\_ON]->(products)

// Connect checkout story to resources

CREATE (story16)-[:DEPENDS\_ON]->(products)

CREATE (story16)-[:DEPENDS\_ON]->(orders)

CREATE (story16)-[:DEPENDS\_ON]->(warehouses)

CREATE (story16)-[:DEPENDS\_ON]->(accounts)

CREATE (story16)-[:DEPENDS\_ON]->(creditCards)

CREATE (story16)-[:DEPENDS\_ON]->(billingAddresses)

CREATE (story16)-[:DEPENDS\_ON]->(shippingAddresses)

CREATE (story16)-[:DEPENDS\_ON]->(customers)

// Connect teams to services

CREATE (servicesTeam)-[:OWNS]->(accountingService)

CREATE (servicesTeam)-[:OWNS]->(shippingService)

CREATE (servicesTeam)-[:OWNS]->(inventoryService)

// Connect services to resources

CREATE (customers)<-[:DEPENDS\_ON]-(accountingService)

CREATE (accounts)<-[:DEPENDS\_ON]-(accountingService)

CREATE (billingAddresses)<-[:DEPENDS\_ON]-(accountingService)

CREATE (shippingAddresses)<-[:DEPENDS\_ON]-(accountingService)

CREATE (warehouseAddresses)<-[:DEPENDS\_ON]-(inventoryService)

CREATE (shippingAddresses)<-[:DEPENDS\_ON]-(shippingService)

CREATE (orders)<-[:DEPENDS\_ON]-(accountingService)

CREATE (orders)<-[:DEPENDS\_ON]-(inventoryService)

CREATE (orders)<-[:DEPENDS\_ON]-(shippingService)

CREATE (products)<-[:DEPENDS\_ON]-(inventoryService)

CREATE (products)<-[:DEPENDS\_ON]-(shippingService)

CREATE (warehouses)<-[:DEPENDS\_ON]-(inventoryService)

CREATE (warehouses)<-[:DEPENDS\_ON]-(shippingService)

CREATE (creditCards)<-[:DEPENDS\_ON]-(accountingService)

// Connect service components to the resources they own

CREATE (accountingService)-[:OWNS]->(customers)

CREATE (accountingService)-[:OWNS]->(accounts)

CREATE (accountingService)-[:OWNS]->(billingAddresses)

CREATE (accountingService)-[:OWNS]->(shippingAddresses)

CREATE (accountingService)-[:OWNS]->(creditCards)

CREATE (shippingService)-[:OWNS]->(orders)

CREATE (inventoryService)-[:OWNS]->(products)

CREATE (inventoryService)-[:OWNS]->(warehouses)

CREATE (inventoryService)-[:OWNS]->(warehouseAddresses)

// Connect services to containers

CREATE (accountingService)-[:DEPLOYED\_IN]->(container)

CREATE (shippingService)-[:DEPLOYED\_IN]->(container)

CREATE (inventoryService)-[:DEPLOYED\_IN]->(container)

**Entity Link Analysis**

Performing entity link analysis on the above dataset will reveal how modular our service components are within our application’s deployment container. We can use these metrics to determine which module is the best candidate for becoming a microservice.

The metric we need to work towards is:

Which service component will have the highest contribution to decreasing the overall centrality of the monolith if removed?

**Which resources have the highest story centrality?**

We can start by measuring the highest resource centrality from the user story perspective. This tells us which of our domain resources are most depended upon by our user stories.

// What resources are most depended upon by the user stories?

MATCH (resource:Resource)<-[:DEPENDS\_ON]-(story:Story)

RETURN resource.name as resource, count(resource) as dependencies

ORDER BY dependencies DESC

**Which service components are most depended upon?**

MATCH (resource:Resource)<-[:DEPENDS\_ON]-(story:Story),

(resource)<-[:OWNS]-(service:Service)

RETURN service.name as service, count(story) as dependencies

ORDER BY dependencies DESC

**Scenarios**

Having a data source that maps the dependencies between services is a tremendously useful thing to have when working on services in an SOA. When making any change to a service you need to understand what other services are consuming your application. Changes to a service could potentially result in a breaking change for other services, causing features on the platform to stop working for users.

**Accounting Service**

Let’s first take a look at the accounting service and answer a few simple questions using Cypher.

Which services depend on the accounting service?

MATCH (services:Service)-[:DEPENDS\_ON]->(resource:Resource),

(resource)<-[:OWNS]-(accountingService:Service { name: "Accounting Service" })

WHERE accountingService <> services

RETURN DISTINCT services.name as service

We can see that the accounting service is depended upon by the shipping service. If we want to understand why, we can query to see which resources are needed by the shipping service from the accounting service.

MATCH (services:Service)-[:DEPENDS\_ON]->(resource:Resource),

(resource)<-[:OWNS]-(accountingService:Service { name: "Accounting Service" })

WHERE accountingService <> services

RETURN services.name as service, collect(resource.name)

Since the accounting service owns the shipping and billing information of customers, it is likely that the service will become more depended upon as new features are added to the platform. But what else does the accounting service do? The following query returns the user stories that depend on all of the resources owned by the accounting service.

MATCH (story:Story)-[:DEPENDS\_ON]->(resource:Resource)

MATCH (service:Service)-[:OWNS]->(resource)

WITH story.name AS stories, collect(DISTINCT service.name) AS services

WHERE length(services)= 1 UNWIND services AS service

WITH service, stories WHERE service = "Accounting Service"

RETURN service, collect(stories)

This tells us that we have three stories that are completely contained within the accounting service. This might make it an excellent candidate to become a microservice. Now we can apply the same question to the other two services to see how they compare.

MATCH (story:Story)-[:DEPENDS\_ON]->(resource:Resource)

MATCH (service:Service)-[:OWNS]->(resource)

WITH story.name AS stories, collect(DISTINCT service.name) AS services

WHERE length(services)= 1 UNWIND services AS service

RETURN service, collect(stories)

The results for this query tells us a lot. We now know that the shipping service is completely reliant on the other services for all the product’s user stories. We also now know that the accounting service, while it might have the highest centrality, it is more tightly coupled to other services than the inventory service. The inventory service has five stories that depend on resources that it owns. It makes a lot of sense to start decomposing our monolith by starting first with the inventory service.

Now that we’ve decided that the inventory service will be functionally decomposed and deployed into its own service container, can we decompose it further to better decrease coupling?

It might make sense to turn the inventory service into two microservices. One of the new microservices would handle those stories that we discovered are completely decoupled. Another microservice could be created to take on any other stories that depends on the inventory service that isn’t in our list of decoupled stories. Let’s expand the previous query to include resources and visualize the results.

MATCH (story:Story)-[:DEPENDS\_ON]->(resource:Resource)

MATCH (service:Service)-[:OWNS]->(resource)

WITH story AS stories, collect(DISTINCT service.name) AS services

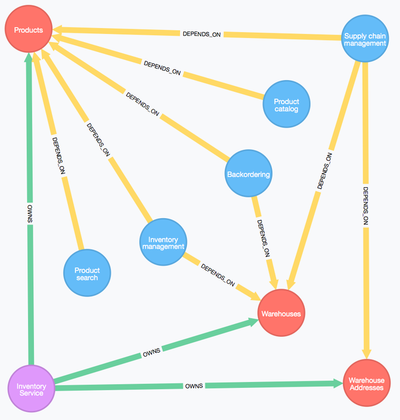
WHERE length(services)= 1 UNWIND services AS service

WITH service, stories

WHERE service = "Inventory Service"

MATCH (stories)-[:DEPENDS\_ON]->(resource)

RETURN DISTINCT resource.name, collect(stories.name)

[](https://camo.githubusercontent.com/2c5b7adbb092f0a46f6d9385b09f7b1573990f9d/687474703a2f2f692e696d6775722e636f6d2f374450527941682e706e67)

We’ve arrived at an excellent candidate for a new microservice. We’ll name the new service the **Products Microservice**, which will now own the resources: **Products**, **Warehouses**, and **Warehouse Addresses**.

The inventory service will become a microservice as well. If we run another query to determine what the left over dependencies are from the decomposition of the inventory service into the product service, we see the following.

MATCH (story:Story)-[:DEPENDS\_ON]->(resource:Resource)<-[:DEPENDS\_ON]-(service:Service { name: "Inventory Service" })

WHERE NOT resource.name IN ["Warehouse Addresses", "Warehouses", "Products"]

MATCH (services:Service)-[:OWNS]->(resource)

RETURN services.name as owner, resource.name as resource, collect(story.name) as stories

The shipping service owns the orders resource. We can decompose the shipping service by turning it into a new microservice and combine its dependencies with the leftovers from the inventory service. Let’s call this new microservice the **Orders Microservice**.

Now that we’ve functionally decomposed our monolith into two new microservices, let’s see what the new dependency graph looks like.

**Products Microservice**

Create the new products microservice and connect it to the resources it owns.

MATCH (resource:Resource)

WHERE resource.name IN ["Warehouse Addresses", "Warehouses", "Products"]

WITH collect(resource) AS resources

CREATE (productsService:Service { name: "Products Microservice" })

FOREACH (x IN resources | MERGE (productsService)-[:OWNS]->(x))

Connect the products microservice to its deployment container.

MATCH (productsService:Service { name: "Products Microservice" })

CREATE (productsServiceContainer:Container { name: "Docker Container" })

CREATE (productsServiceContainer)<-[:DEPLOYED\_IN]-(productsService)

Connect the products microservice to its service team.

MATCH (productsService:Service { name: "Products Microservice" })

MATCH (team:Team { name: "Services team" })

CREATE (team)-[:OWNS]->(productsService)

Finally, delete the inventory service and its relationships.

MATCH (inventoryService:Service { name: "Inventory Service" })

OPTIONAL MATCH (inventoryService)-[r]-()

DELETE r, inventoryService

**Orders Microservice**

Create the new orders microservice and connect it to the resources it owns.

MATCH (resource:Resource)

WHERE resource.name = "Orders"

WITH collect(resource) AS resources

CREATE (ordersService:Service { name: "Orders Microservice" })

FOREACH (x IN resources | MERGE (ordersService)-[:OWNS]->(x))

Connect the orders microservice to its deployment container.

MATCH (ordersService:Service { name: "Orders Microservice" })

CREATE (ordersServiceContainer:Container { name: "Docker Container" })

CREATE (ordersServiceContainer)<-[:DEPLOYED\_IN]-(ordersService)

Connect the orders microservice to its service team.

MATCH (ordersService:Service { name: "Orders Microservice" })

MATCH (team:Team { name: "Services team" })

CREATE (team)-[:OWNS]->(ordersService)

Finally, delete the shipping service and its relationships.

MATCH (shippingService:Service { name: "Shipping Service" })

OPTIONAL MATCH (shippingService)-[r]-()

DELETE r, shippingServic