SPRINGONE2GX
WASHINGTON, DC

Securing Microservices with Spring Cloud Security

By Will Tran
twitter.com/fivetenwill




Will > About

About Me

- Spring user since Spring 2.0
- Works for Pivotal
  - Currently on Spring Cloud Services
  - Formerly on Pivotal SSO, CF UAA, PCF Mobile Services
- 2nd time speaker at SpringOne 2GX
- Based in Toronto, Canada

Microservices?

What are Microservices?

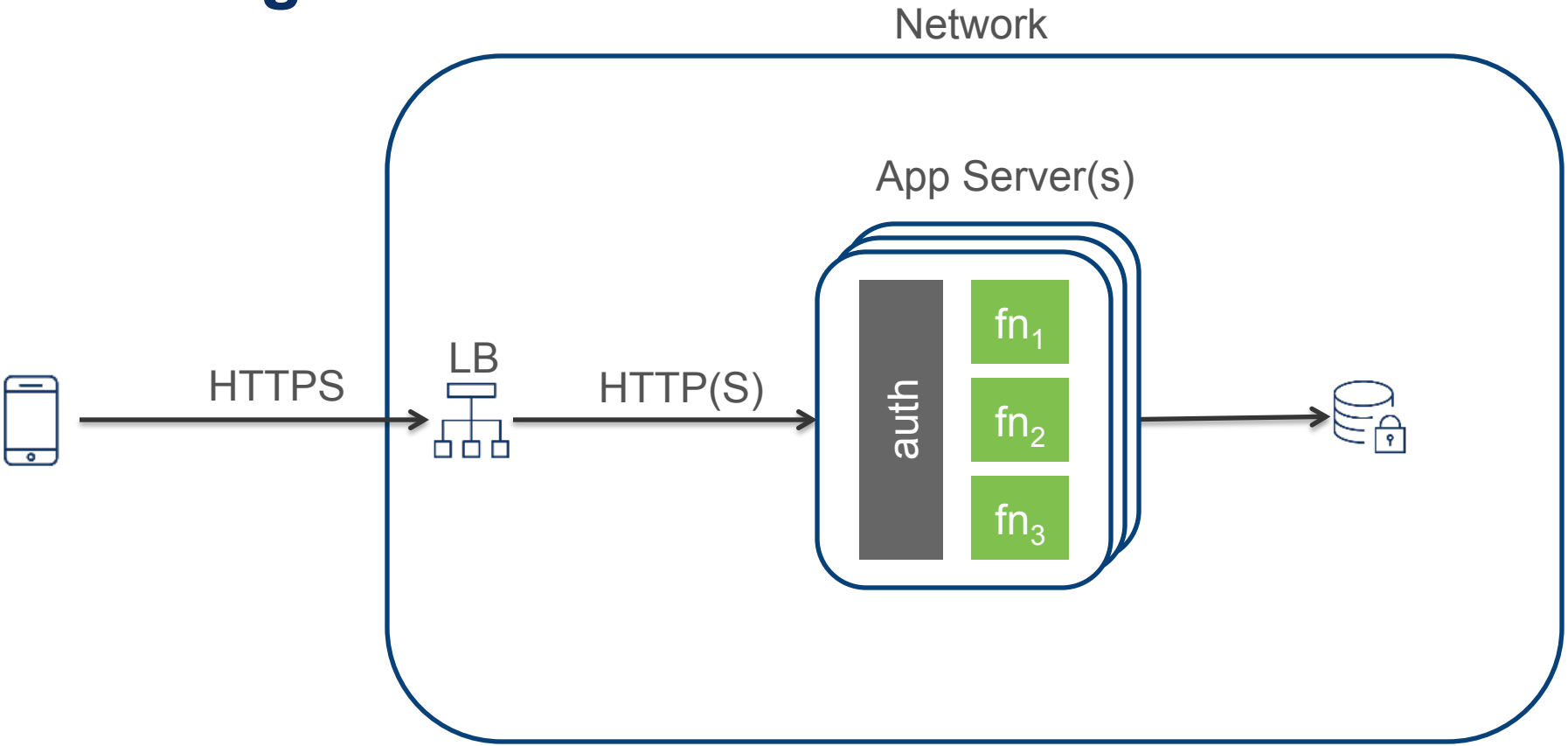
- “A loosely coupled service oriented architecture with bounded contexts” – Adrian Cockroft
- Loosely coupled
  - Services can be updated independently
- Bounded context
  - Services are responsible for a well defined business function
  - And care little about the services that surround it
  - ie. “Do one thing and do it well”

What are Microservices?

- Microservice Architectures are
  - HTTP based (or communicate via other open standards)
  - Containerized
  - Independently deployable and scalable
  - Self-sufficient
    - Makes as little assumptions as possible on the external environment

What about security?

Securing the monolith



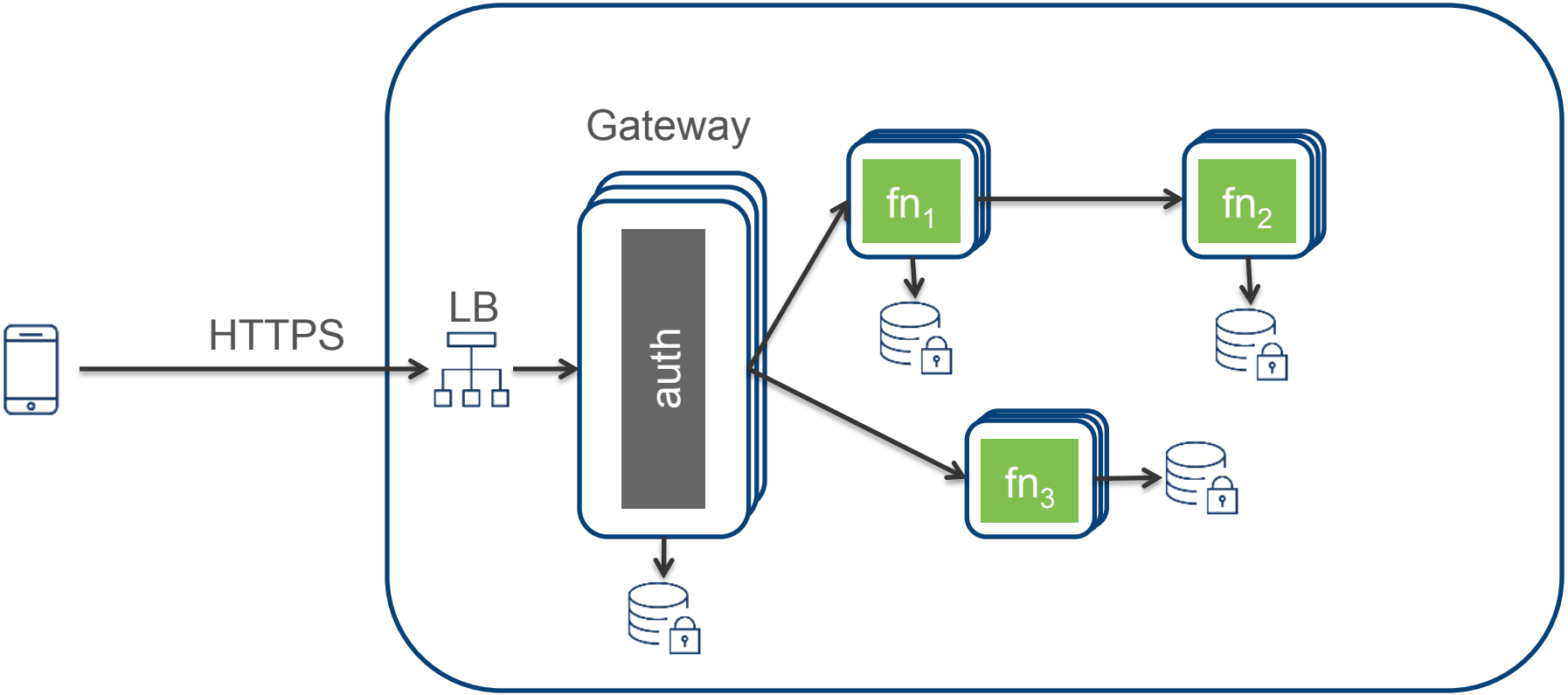
Securing the monolith is EASY(er)!

- You only need to auth the request once per user request
  - No session?
    - Verify user credentials
    - Get the users roles
    - Start a user session
  - Yes session?
    - Verify session not expired
- Request/response is handled in process
  - You can trust method calls

Securing the monolith is EASY(er)!

- Pros
  - Limited attack surface
- Cons
  - The app has all the credentials it needs to do anything it wants to the DB
  - Break the process and you get it all

Securing a Microservice Architecture



Microservice Security Is Harder

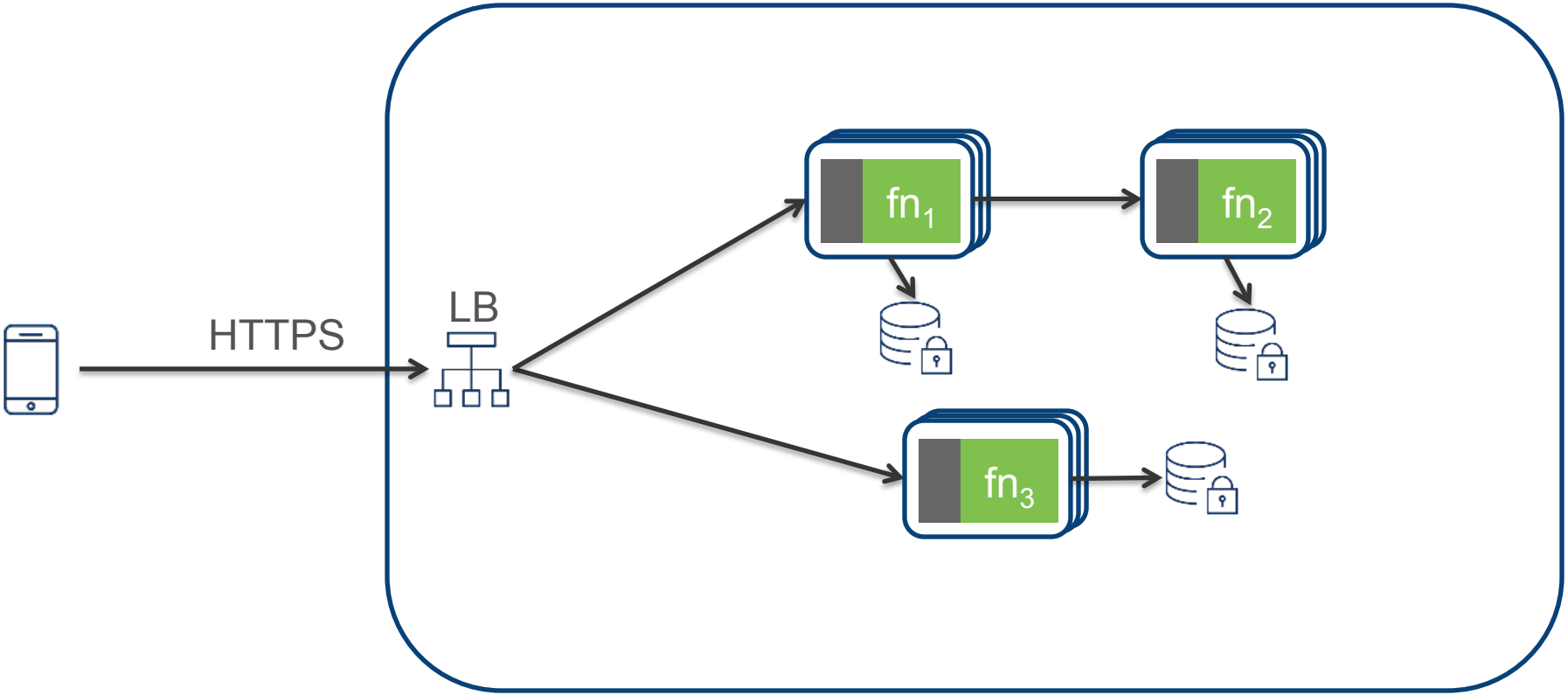
- Win!
  - Principal of least privilege
    - Every component only has access to what it needs to perform its function
- Lose
  - Much larger attack surface (especially for internal threats)
  - How do other services know who's accessing them?
  - How can other services trust each other?

Microservice Security Implementations

API Gateway / Perimeter security

- Requests are authenticated and authorized by the gateway
- The public LB cannot send requests to apps directly
- Apps trust all traffic they receive by assumption
- Pros
  - Network setup can virtually guarantee assumptions
  - Apps have stateless security (assumption is stateless)
- Cons
  - Does nothing for internal threats

Securing a Microservice Architecture

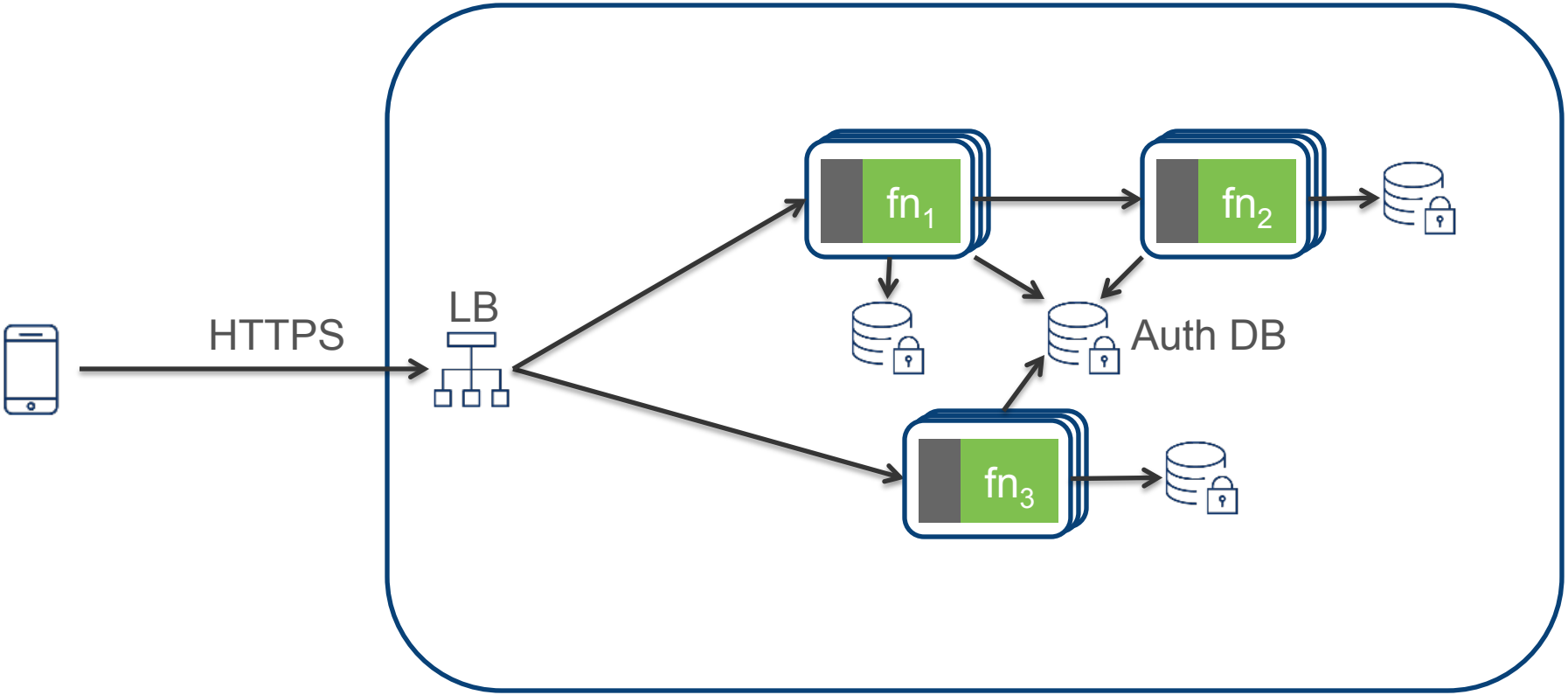


Microservice Security Implementations

Everybody can auth (with HTTP Basic)

- All apps get to do authentication and authorization themselves
- Basic credentials are passed along in every request
- Pros:
  - Stateless (authenticate every time)
  - Easy
- Cons:
  - How do you store and lookup the credentials?
  - How do you manage authorization?
  - User's credentials can unlock all functionality (until user updates password)

Securing a Microservice Architecture



Microservice Security Implementations

Basic + Central Auth DB

- All apps get to do authentication and authorization themselves
- Basic credentials are passed along in every request
- Credentials are verified against a central DB
- Pros:
  - Central user store
  - Stateless (authenticate every time)
- Cons:
  - Auth DB is hit every request
  - DB lookup logic needs to be implemented everywhere
  - User's credentials can unlock all functionality

Microservice Security Implementations

Sessions Everywhere

- Same as before but each app gets to maintain a session with the client device
- Pros:
  - Auth DB is hit once per session
- Cons:
  - Hard to manage all the sessions
  - No single sign on
  - DB lookup logic needs to be implemented everywhere
  - User's credentials can unlock all functionality

Microservice Security Implementations

API Tokens

- Username and password is exchanged for a token at a centralized auth server
- Apps validate the token for each request by hitting the auth server
- Pros:
  - Apps don’t see user credentials
- Cons:
  - Auth server bottleneck
  - Token provides all or nothing access

Microservice Security Implementations

SAML

- Identity provider provides signed assertions to apps
- Apps can trust the assertions because they're signed
- Pros:
  - Standard trust model
  - Self verification of assertions
- Cons:
  - XML is big and stinky
  - Difficult for non-browser (eg mobile) clients

Microservice Security Concerns

Common concerns

- Central user store bottleneck
- Single sign on
- Statelessness
- User credentials == pure pwnage
- Fine grained authorization
- Interoperability with non browser clients

Enter OAuth2 + OpenID Connect

What is OAuth2?

Delegated Authorization

- A protocol for conveying authorization decisions (via a token)
- Standard means of obtaining a token (aka the 4 OAuth2 grant types)
  - Authorization Code
  - Resource Owner Password Grant
  - Implicit
  - Client Credentials
- Users and Clients are separate entities
  - “I am authorizing this app to preform these actions on my behalf”

What is OAuth2 Not?

OAuth2 is not Authentication

- The user must be authenticated to obtain a token
- How the user is authenticated is outside of the spec
- How the token is validated is outside the spec
- What the token contains is outside the spec
- Read more: http://oauth.net/articles/authentication/

What is OpenID Connect?

Delegated Authentication

- A protocol for conveying user identity (via a signed JWT)
- Built on top of OAuth2
- Standard means of obtaining an ID token
  - The same 4 OAuth2 grant types are supported
- Standard means of verifying ID tokens
- “Will is authorizing this app to preform these actions on his behalf”
  - And here’s his email address in case you need it

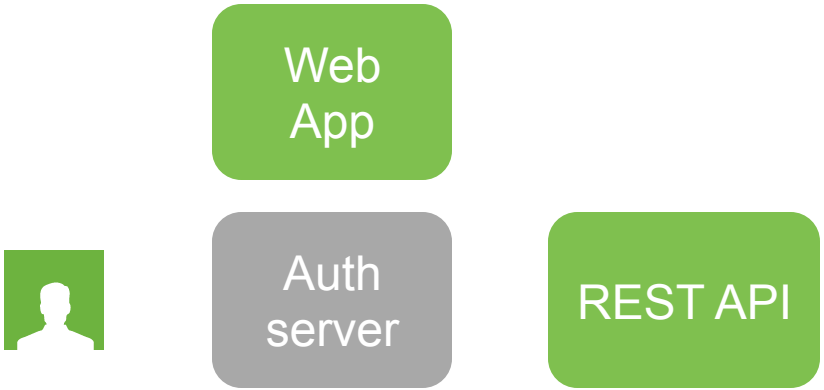
What is OpenID Connect Not?

Authentication

- Still doesn't say how users are to be authenticated
- This is good: there's lots of ways to authenticate users
  - Internal DB
  - Another Identity Provider
    - SAML
    - LDAP
  - Multi-factor

How to get tokens

Authorization Code Flow



Actors:

- User - Resource Owner
- Web App - Client
- REST API - Resource Server
- Auth server – OpenID Connect Provider (eg Google)

Setup:

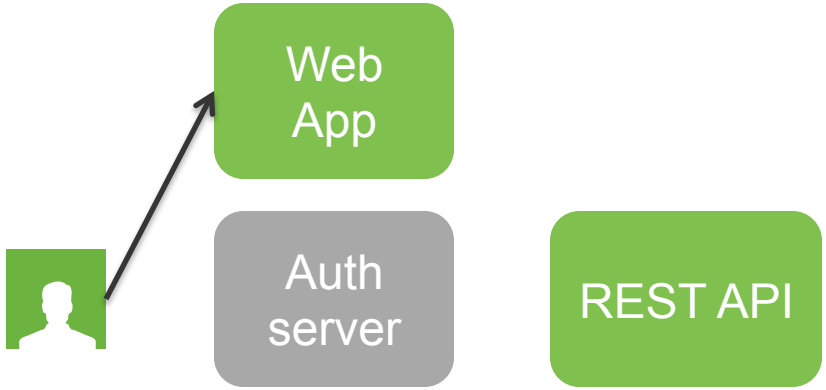
- User has no session with the auth server or web app

Use case:

- User wants to place an order on the REST API using the web app

How to get tokens

Authorization Code Flow

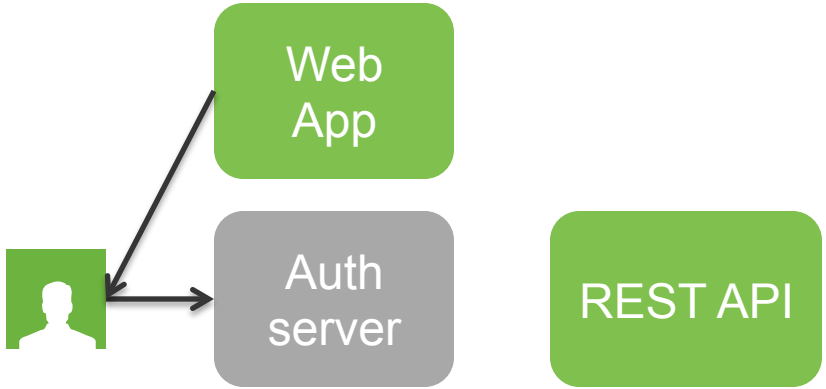


Step 1:

User accesses web app and does not have a session with it.

How to get tokens

Authorization Code Flow



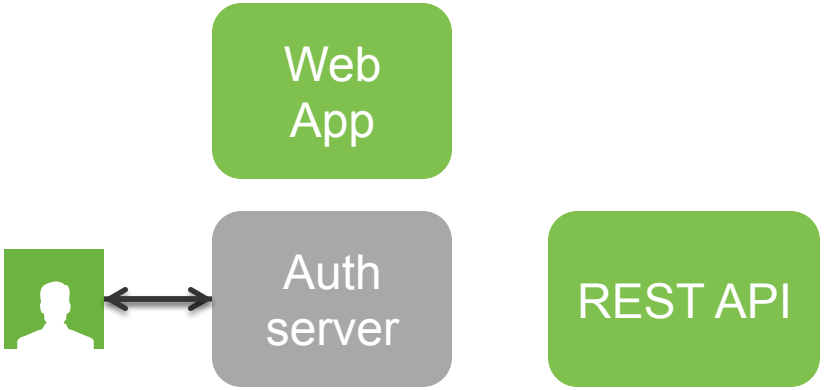
Step 2:

Web app redirects user to the authorize endpoint on the auth server. The redirect URL contains the scopes openid and order.me

This means that the web app is requesting a token that allows apps to view the users identity (openid) and place orders on the user's behalf (order.me).

How to get tokens

Authorization Code Flow

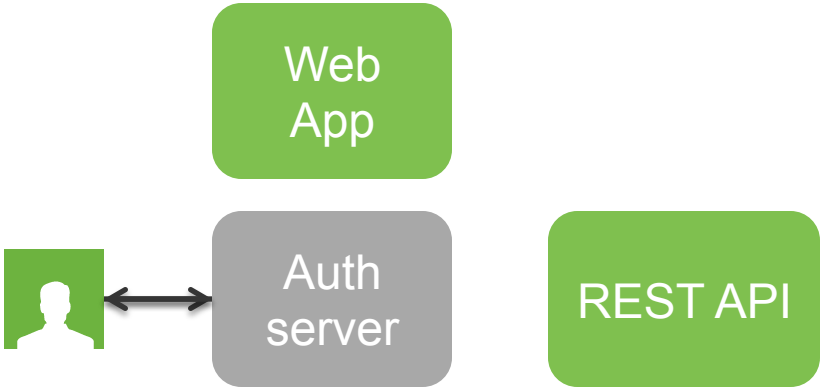


Step 3:

Auth server redirects user to its login page because the user isn't logged in

How to get tokens

Authorization Code Flow



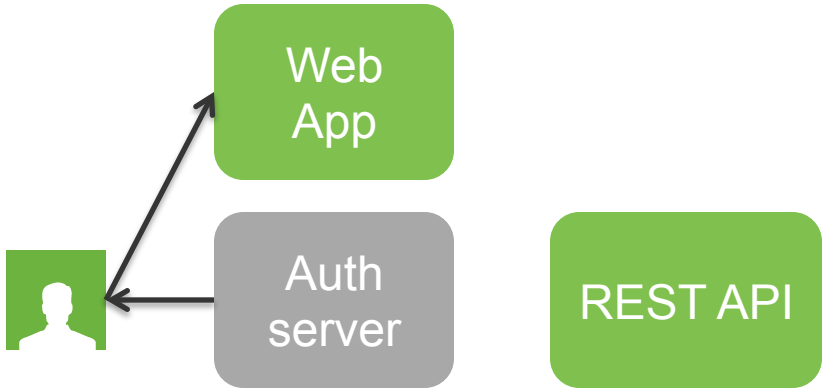
Step 4:

User logs in, starts a session with the auth server, and is redirected back to the authorize endpoint.

Control is given back to the user, who sees a page asking if the user permits the web app to access the their identity and manage their orders on their behalf.

How to get tokens

Authorization Code Flow

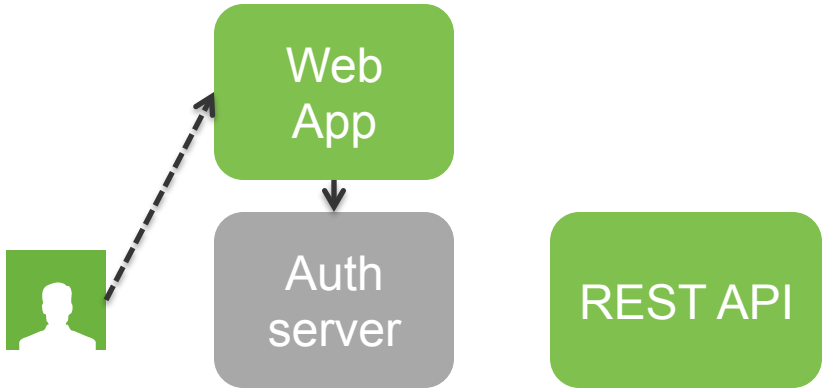


Step 5:

User authorizes access. Auth server redirects the user back to the web app with a one time code in the query params of the redirect

How to get tokens

Authorization Code Flow



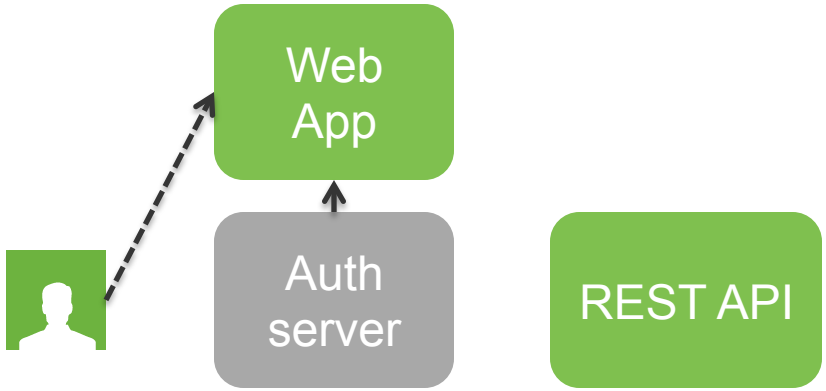
Step 6:

Web App hits the token endpoint with the one time code in the query params.

Auth server validates the code.

How to get tokens

Authorization Code Flow



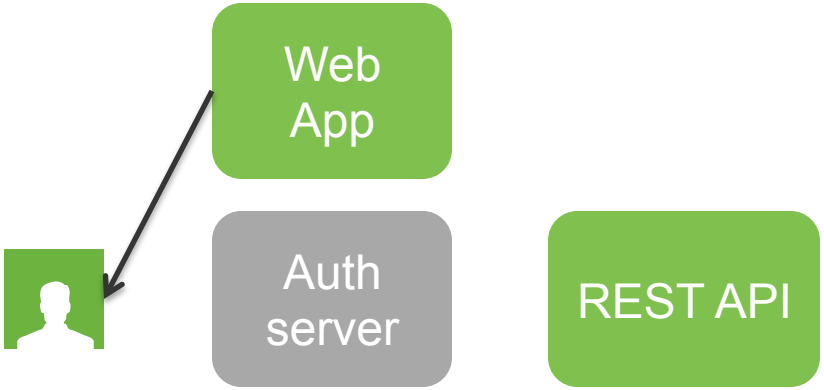
Step 7:

Auth server responds with an access token (random string), and ID token (signed JWT).

Web app verifies the ID token, consumes its contents, and starts an authenticated session, and saves the access token in session

How to use tokens

The Resource Server

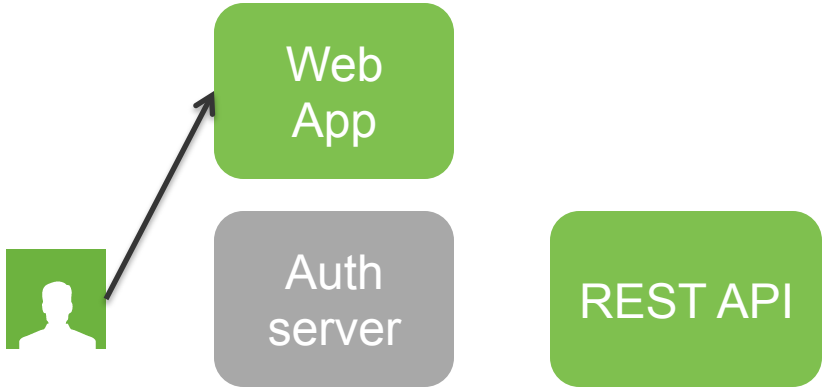


Step 8:

Web app now gives control back to the user and responds with an order form.

How to use tokens

The Resource Server

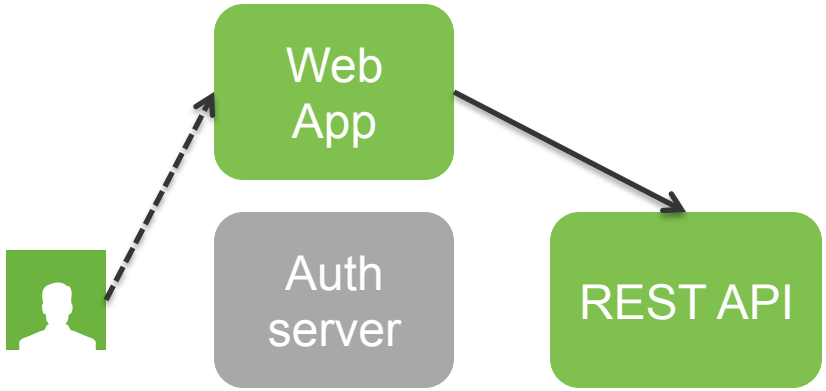


Step 9:

User fills out and submits the order form

How to use tokens

The Resource Server

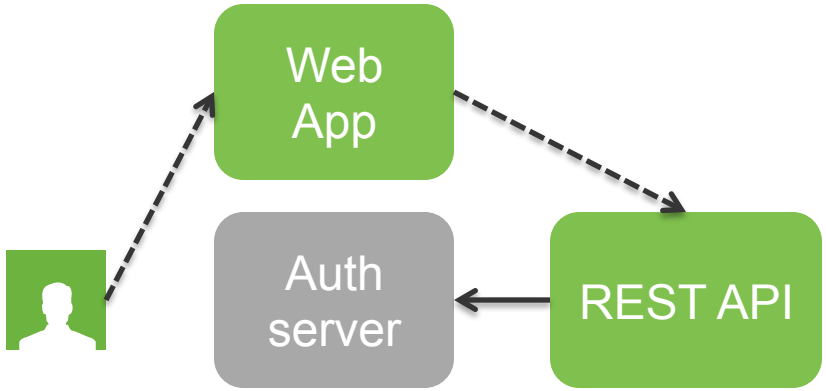


Step 10:

The web app submits the order to the REST API with the access token that was stored in session.

How to use tokens

The Resource Server

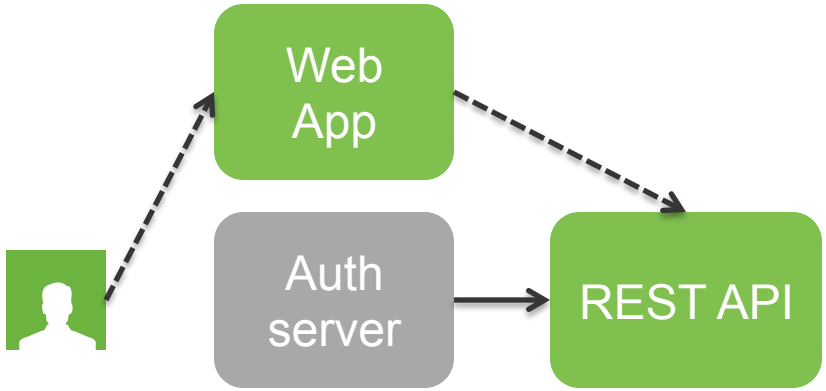


Step 11:

The REST API needs validate the token. It sends the token to the Auth server's token verification endpoint.

How to use tokens

The Resource Server

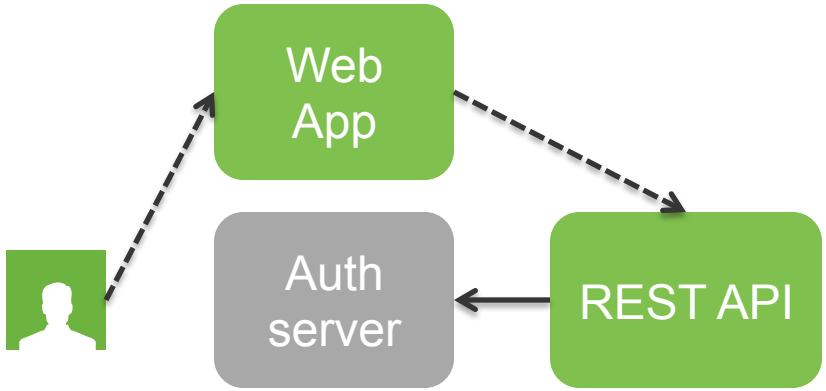


Step 12:

The Auth server responds with the permissions (scopes) that the token grants. The REST API now knows that the request is authorized.

How to use tokens

The Resource Server

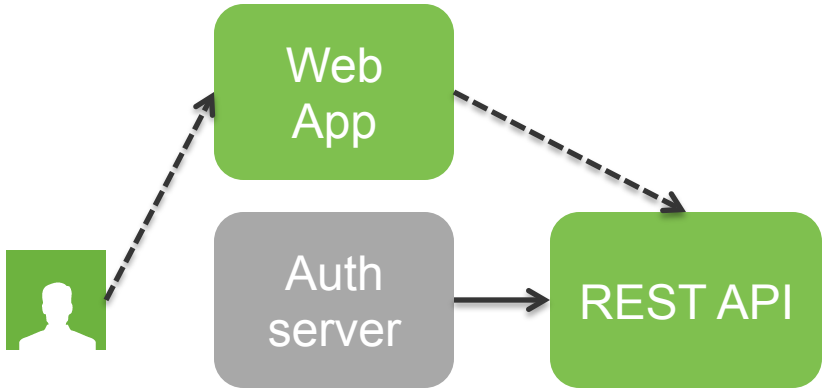


Step 13:

But wait! Before saving the order, the REST API wants to populate it with other user information not contained in the request, eg address, phone number. The REST API make a request with that same token to the /userinfo endpoint

How to use tokens

The Resource Server

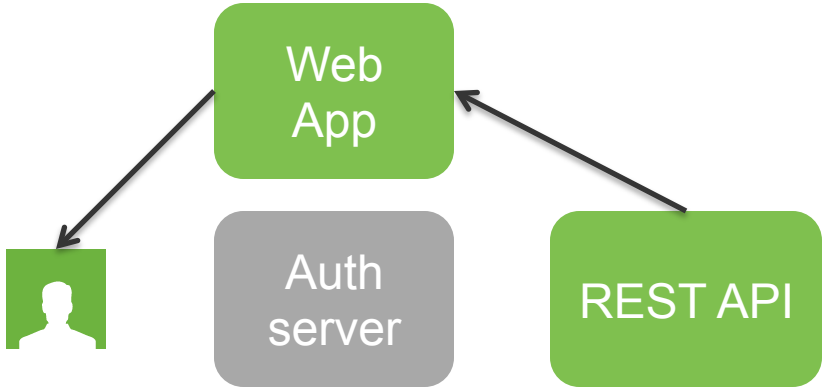


Step 14:

The Auth server responds with the user's information. The REST API can now save the order.

How to use tokens

The Resource Server

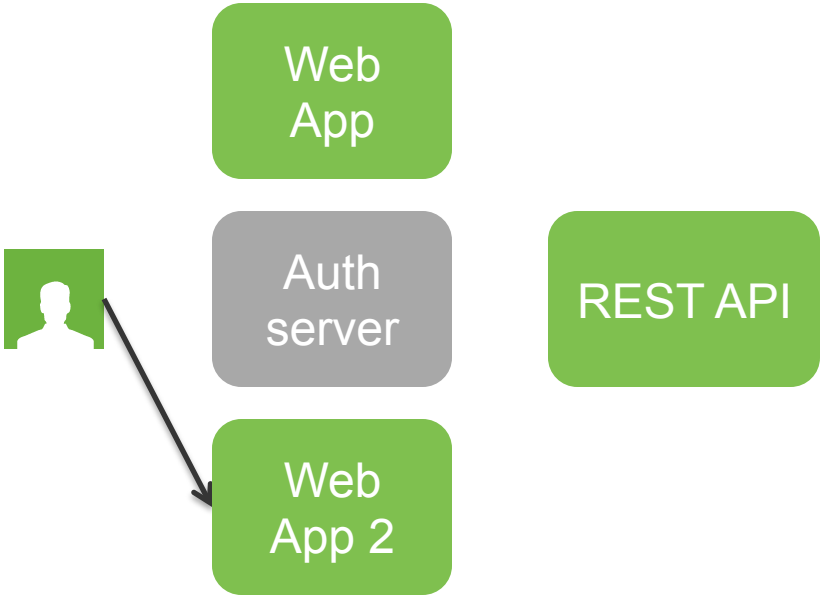


Step 15:

Control is now given back to the user.

How to SSO

Single Sign On

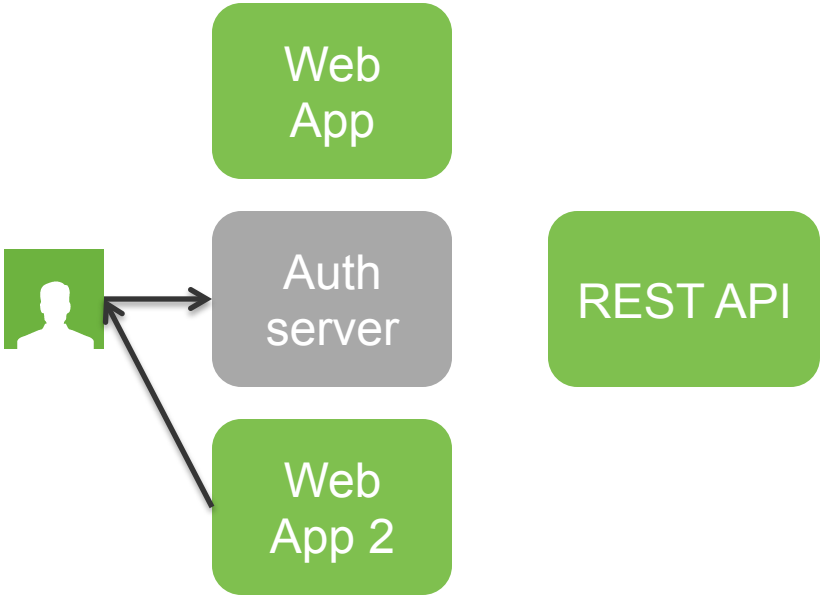


Step 16:

User wants to use Web App 2 to track their order and is not authenticated with it.

How to SSO

Single Sign On

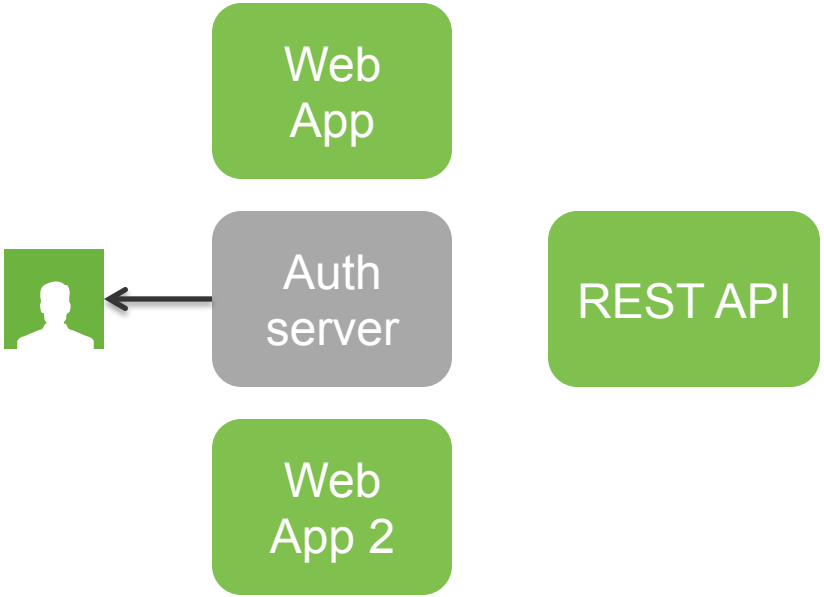


Step 17:

Web App 2 redirects the user to the Auth server's /authorize endpoint

How to SSO

Single Sign On



Step 18:

The user already has an authenticated session with the Auth server, so the server responds to the /authorize request with a page that asks if the user permits the web app to access the their identity and manage their orders on their behalf. (And the flow continues as before)

Enter Spring Cloud Security

Spring Cloud Security

Features

- SSO with OAuth2 and OpenID Connect servers
  - With a single annotation (and some config)
- Secure Resource Servers with tokens
  - With a single annotation (and some config)
- Relay tokens between SSO enabled webapps and resource servers
  - With an autoconfigured OAuth2RestTemplate

Spring Cloud Security

Caveats

- OpenID Connect ID tokens aren't directly consumed
  - But you can use /userinfo instead
  - But if the access token is a JWT containing identity claims you're in luck

But I thought Access Tokens were opaque!

RFC 6749 OAuth 2.0 October 2012

1.4. Access Token

Access tokens are credentials used to access protected resources. An access token is a string representing an authorization issued to the client. The string is usually opaque to the client.

If Access Tokens carried information…

With scopes in the token

- You can authorize the request yourself

With identity claims in the token

- You know who the originator of the request is

With a signed token

- You can validate the token's authenticity yourself
- Your auth server won't become the bottleneck

Enter UAA

UAA to the rescue

What is UAA

- The User Account and Authorization server
- Core component of Cloud Foundry, battle tested in production
- Apache 2 License, download the WAR and run it for free
- OAuth2 compliant, almost OpenID Connect compliant
  - Supports /userinfo
- Multitenant
- Spring Security OAuth2 is based on UAA
- Spring Cloud Security is a great fit with UAA
  - Because UAA produces JWT containing both scopes and identity

JWT to the rescue

What is JWT

- JSON Web Token (RFC7519), standardized May 2015
- Header, payload, signature
- Base64 encoded form is easy to transmit in headers
- Standardized generation and verification of signatures
- Can encapsulate any claim (scopes, identity)
- Can expire
- Enables scalable, stateless authentication and authorization
  - Clients can verify tokens themselves
  - With the tradeoff of losing token revocation

Microservice Security Concerns

Common concerns

- Central user store bottleneck ✔
- Single sign on ✔
- Statelessness ✔
- User credentials == pure pwnage ✔
- Fine grained authorization ✔
- Interoperability with non browser (mobile) clients ✔

Let’s See It Work

Demo Time

Follow along on GitHub:
github.com/william-tran/microservice-security

Demo 1: SSO

SSO with Google OpenID Connect and UAA

- The only difference is configuration

Show Me How

Demo 2: Freddy's BBQ Joint

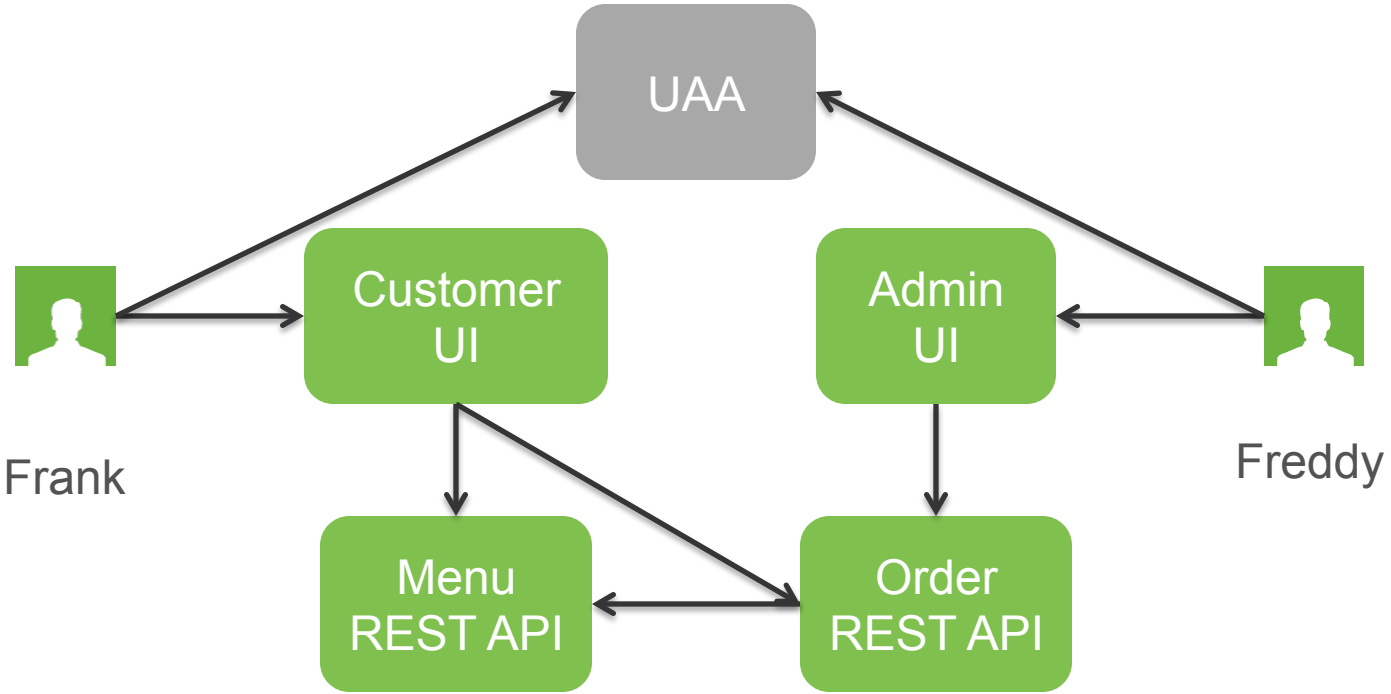
Actors

- Freddy, owner of Freddy's BBQ Joint, the best ribs in DC
- Frank, Freddy's most important customer (and the most powerful man in the world)
- The Developer, works for Frank and wants to impress him with a side project

Use Case

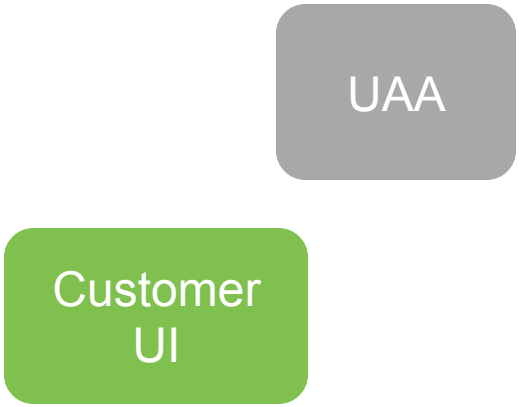
- Give Frank the ability to see the menu online and place orders
- Give Freddy the ability to manage the menu and close orders

Demo 2: Freddy's BBQ Joint

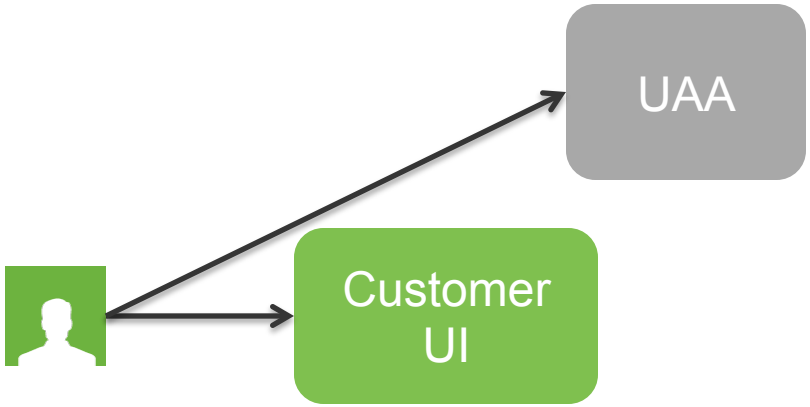


Demo 2: Freddy's BBQ Joint

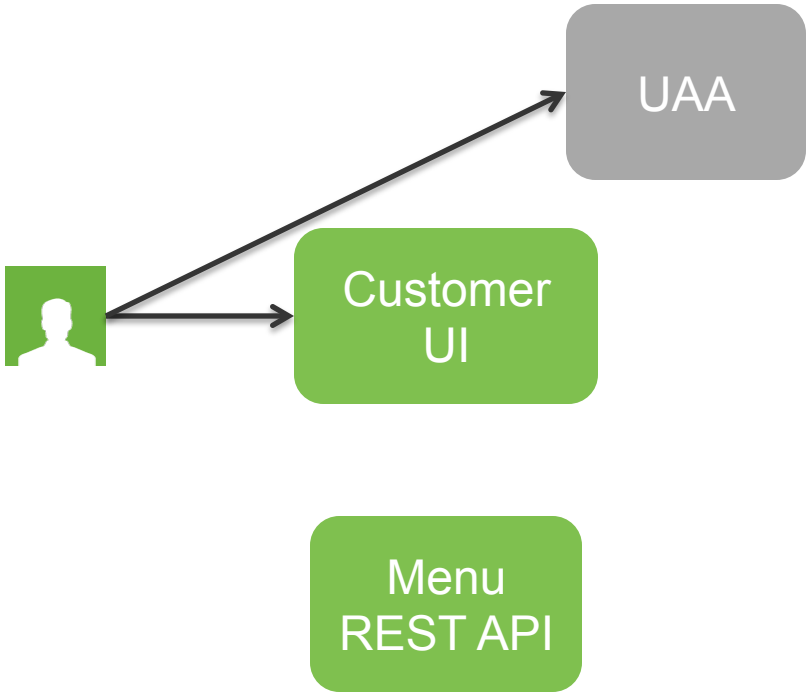
Demo 2: Freddy's BBQ Joint



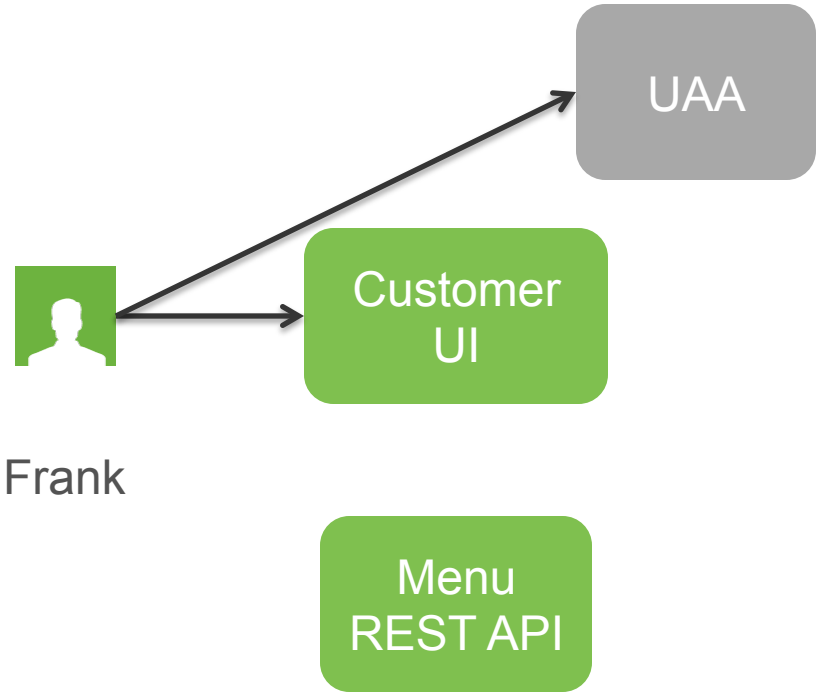
Demo 2: Freddy's BBQ Joint



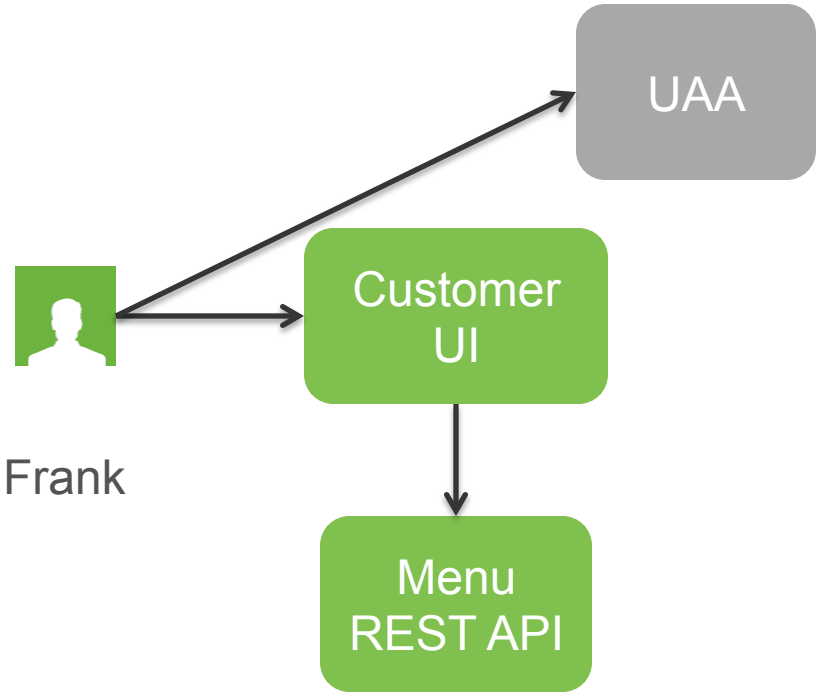
Demo 2: Freddy's BBQ Joint



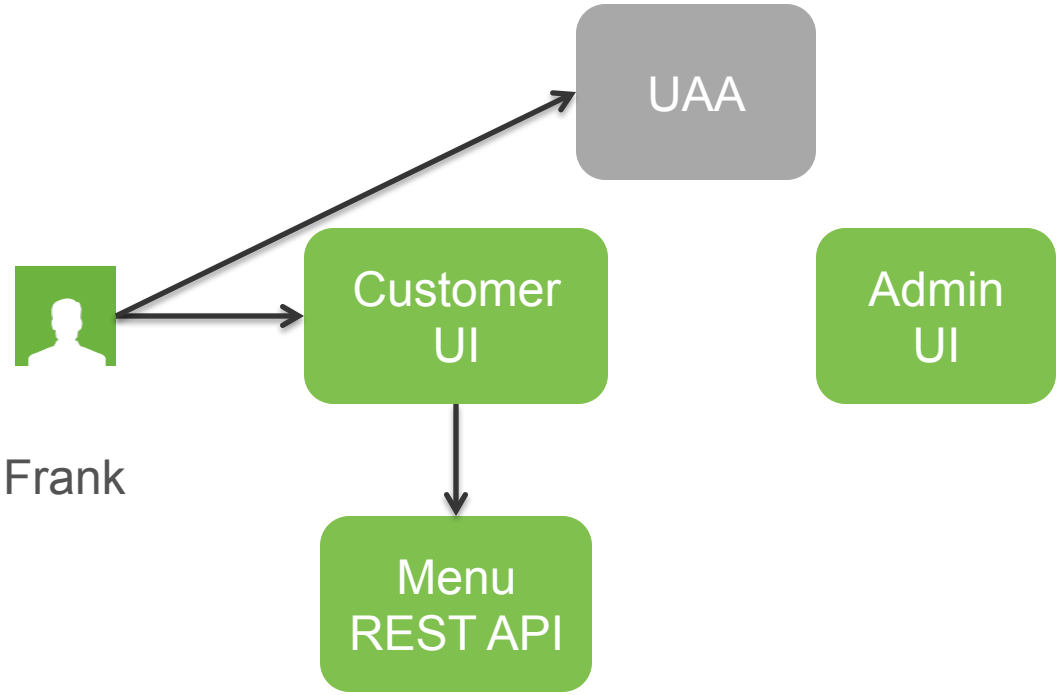
Demo 2: Freddy's BBQ Joint



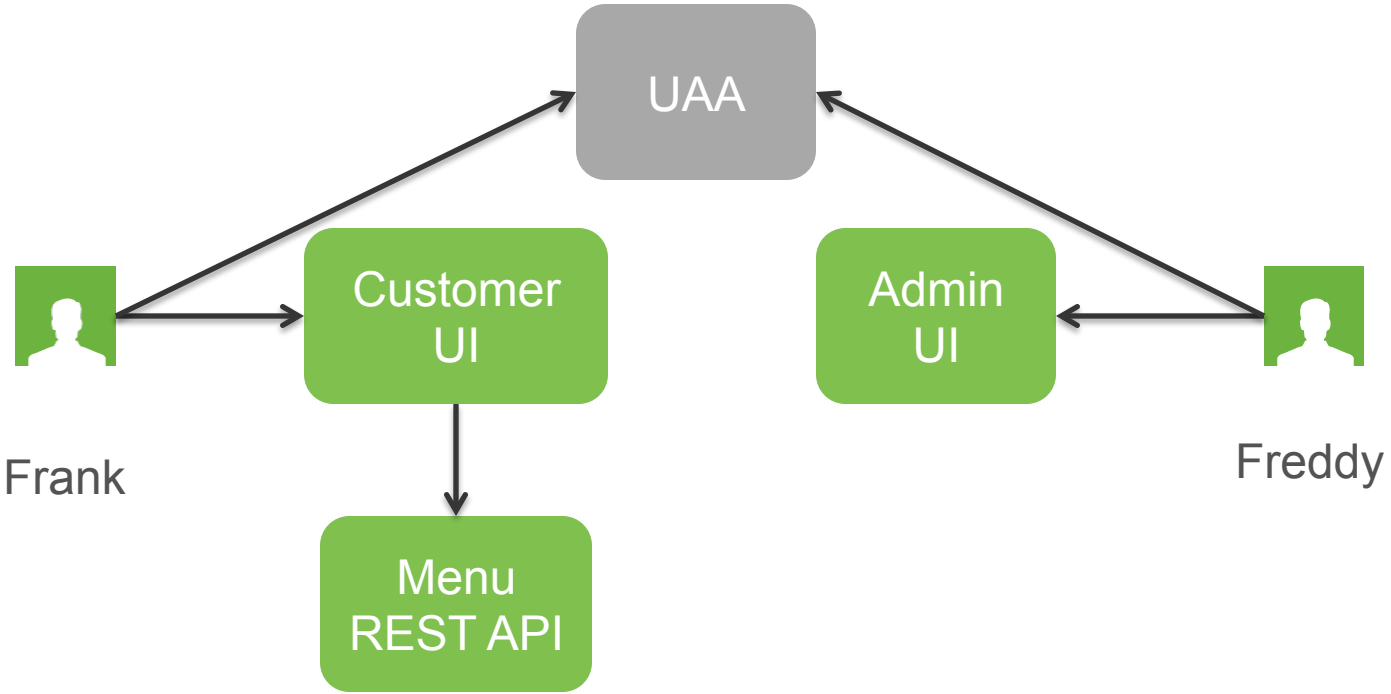
Demo 2: Freddy's BBQ Joint



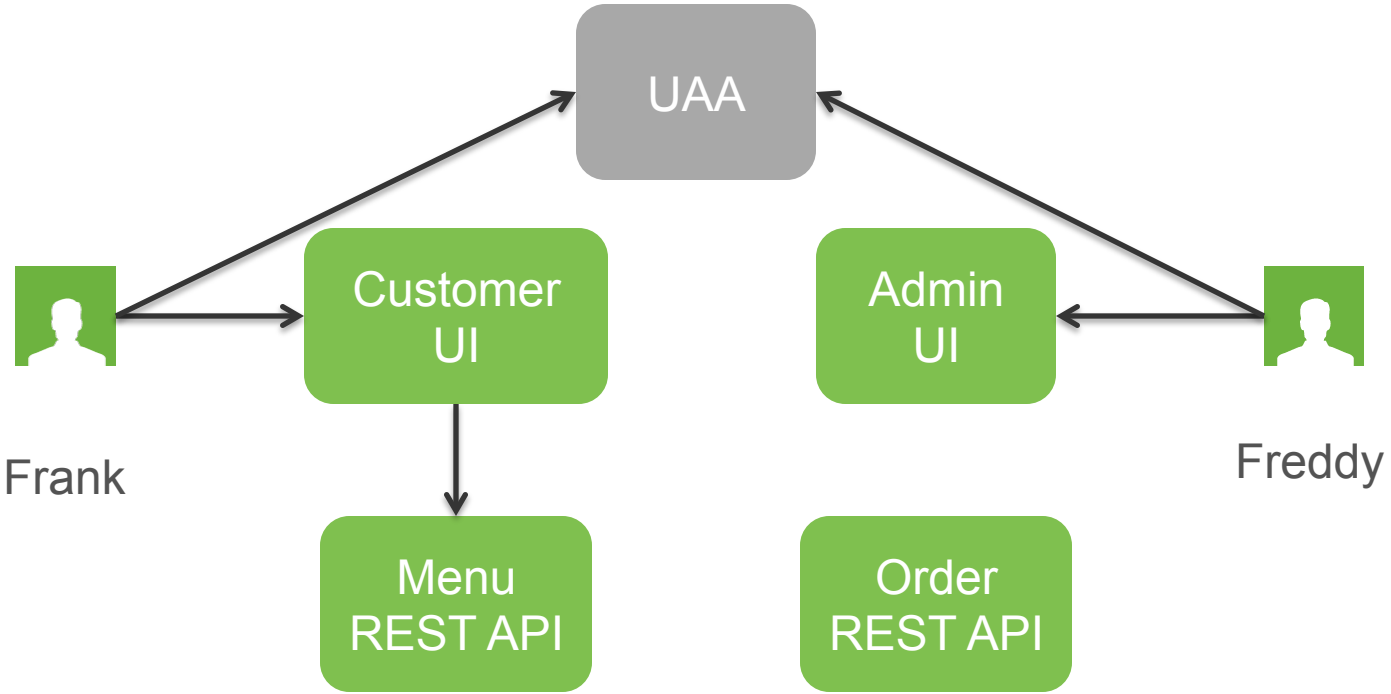
Demo 2: Freddy's BBQ Joint



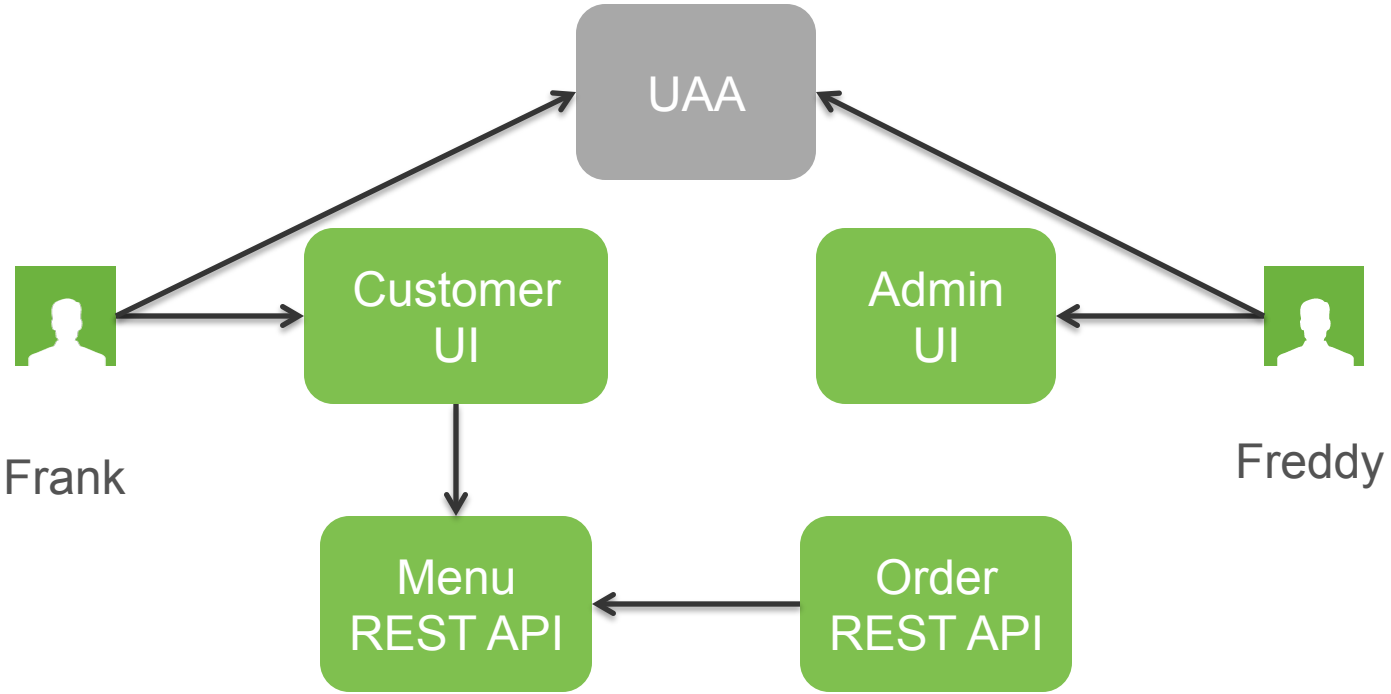
Demo 2: Freddy's BBQ Joint



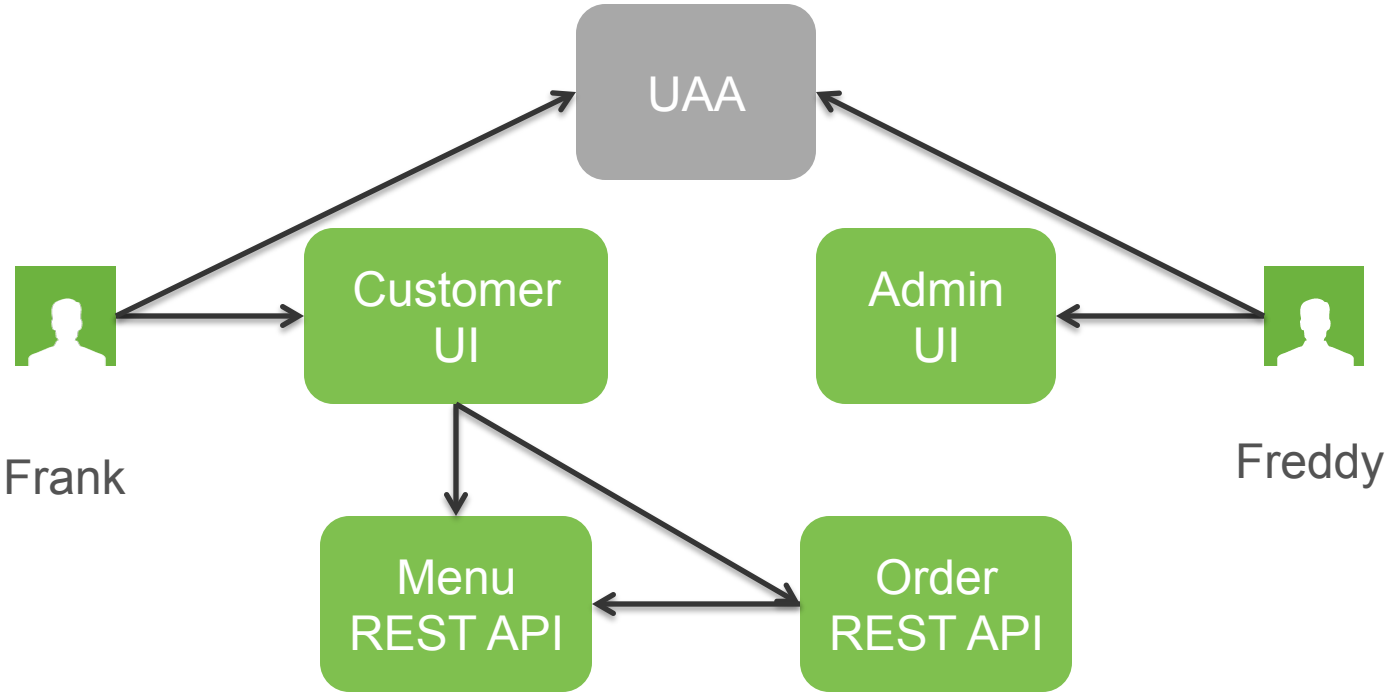
Demo 2: Freddy's BBQ Joint



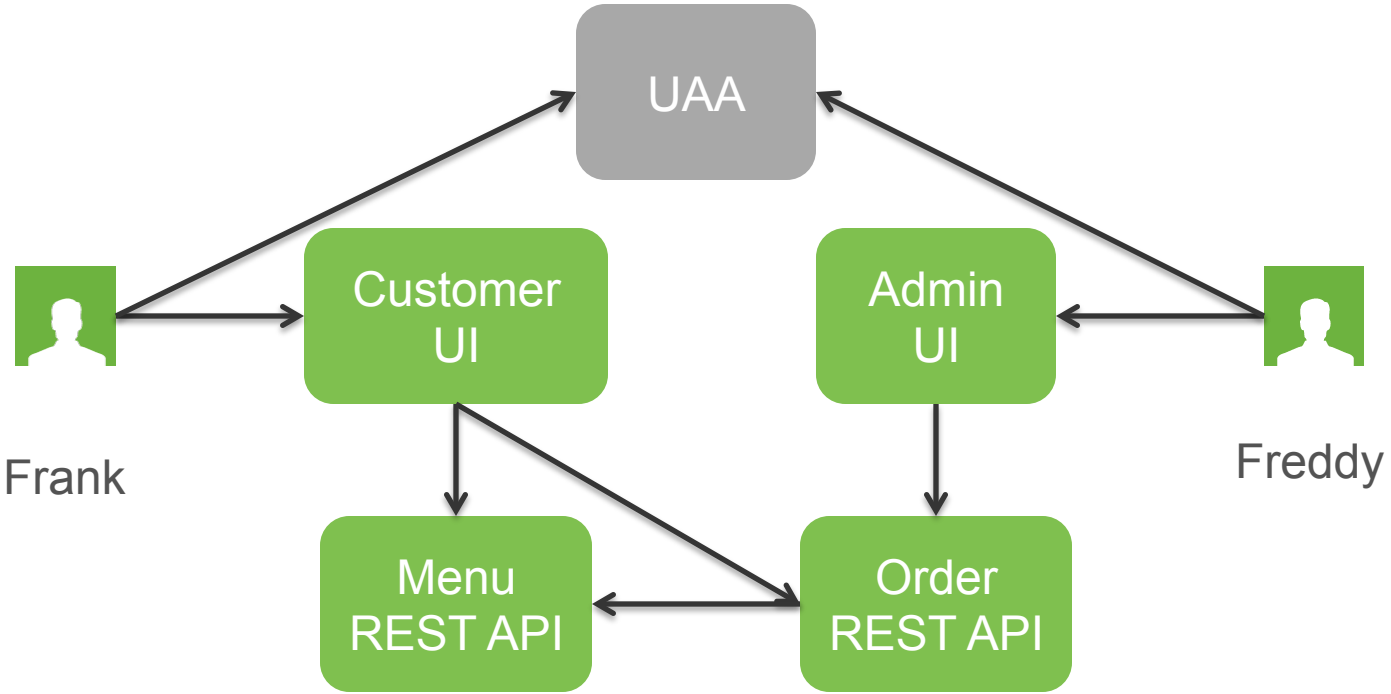
Demo 2: Freddy's BBQ Joint



Demo 2: Freddy's BBQ Joint



Demo 2: Freddy's BBQ Joint



Fin

Thank You Cloud Foundry Identity Team

Madhura Bhave

Chris Dutra

Filip Hanik

Rob Gallagher

Sree Tummidi

SPRINGONE2GX
WASHINGTON, DC

Learn More. Stay Connected.

 @springcentral

 Spring.io/video

Follow me on Twitter: @fivetenwill

Heckle my commits on github.com/william-tran

Questions?