COMSE6998: Modern Serverless Cloud Applications

Lecture 6: Some Patterns, OAuth2, Social Media, Static Website, Graph DB

Dr. Donald F. Ferguson
Donald.F.Ferguson@gmail.com

Contents

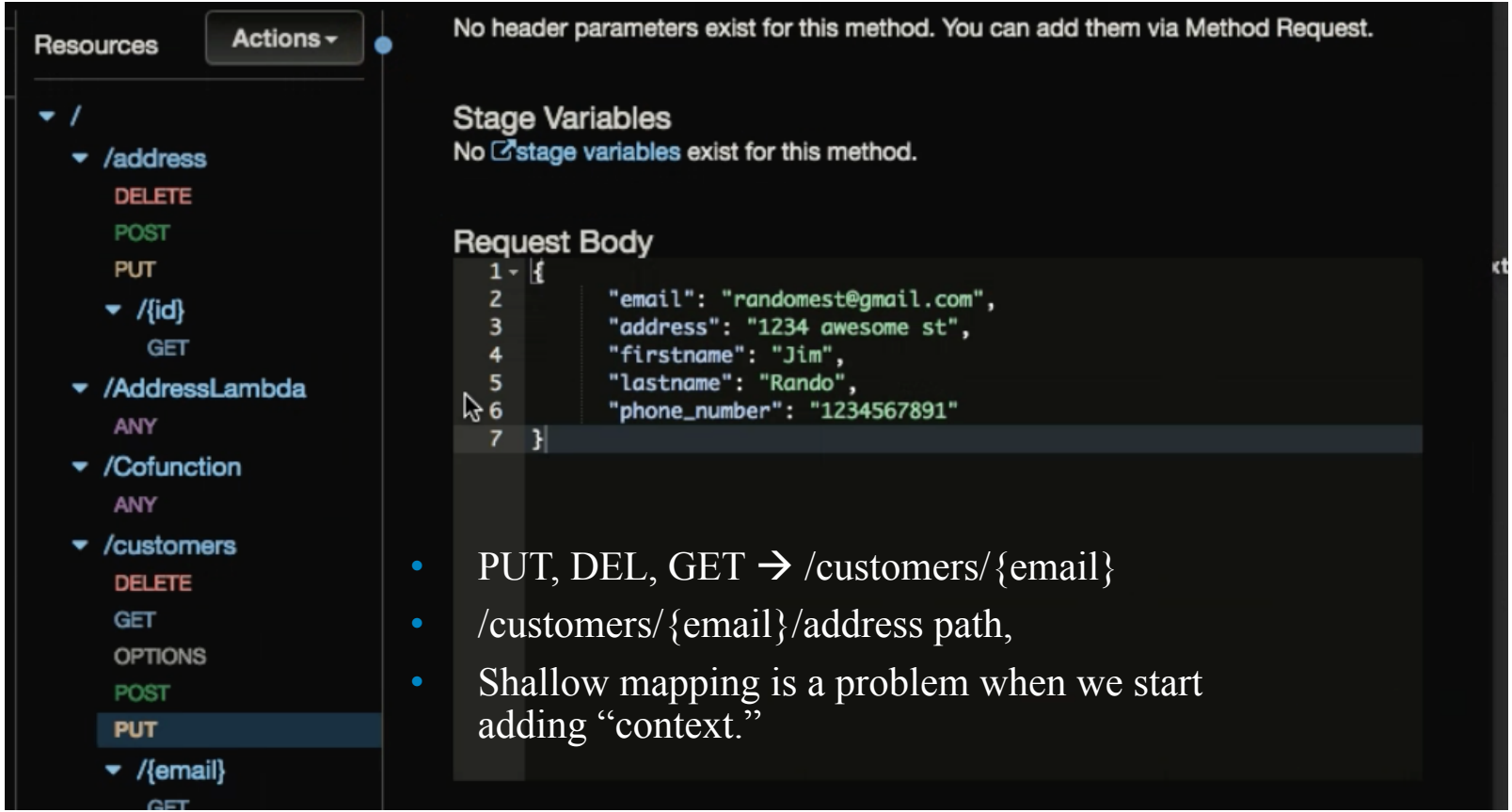
- Introduction
  - Homework status, and some questions.
  - Could of Comments: Avoid direct model → code mapping.
  - Q&A
- New technical topics:
  - OAuth2
  - Social Media Integration (Review)
  - Building a Web Site – Serverless
  - Graph Databases (Intro).
  - ~~Content/Asset Management~~
- 3rd Project

Introduction

Contents

- Introduction
  - Homework status, and some questions.
  - Could of Comments: Avoid direct model → code mapping.
  - Q&A
- New technical topics:
  - OAuth2
  - Social Media Integration (Review)
  - Building a Web Site – Serverless
  - Graph Databases (Intro).
  - ~~Content/Asset Management~~
- 3rd Project

Some Comments



- PUT, DEL, GET → /customers/{email}
- /customers/{email}/address path,
- Shallow mapping is a problem when we start adding "context."

Incoming/Outgoing "Event"

```
public class RESTMessage implements Cloneable {


    private static final org.slf4j.Logger logger = LoggerFa

    private Context context;
    private Body body;
    private Header header;
    private Environment environment;

    public RESTMessage() {

    }
```

```
public class Header implements Cloneable {

    private static final org.slf4j.Logger logger = LoggerFactory.getLogger(Header.class);

    // The API gateway request mapping with set the operation ID.
    // Otherwise, different invocation channels, if used, may set the ID. If not
    // set,
    // we should generate.
    private String opId;

    // The message could be a REQUEST or a RESPONSE
    private Integer messageType;

    // The verb is part of the operation, not the context.
    private String verb;

    // We use this to figure out which REST service to call. We should change
    // this to the
    // "path" to the resource, e.g. /api/customers, /api/customer/sociallogin.
    private String resourcePath;

    // MessageStatus
```

```
public class Context extends JSONUtil implements Cloneable{

    private static final org.slf4j.Logger logger = LoggerFactory.getLogger(Context.class);

    protected   HashMap<String, String> headers;
    protected   HashMap<String, String> queryParams;
    protected   HashMap<String, String> pathParams;

    public  Context() {
```

Technical Topics

OAuth2

Demos

- Facebook
- Twitter

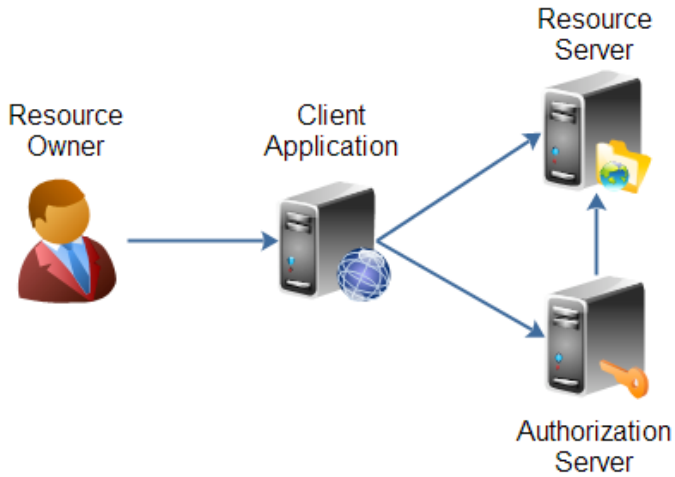
Overview (http://tutorials.jenkov.com/oauth2/index.html)

- Resource Owner
  - Controls *access* to the "data."
  - Facebook **user**, LinkedIn **user**, …
- Resource Server
  - The website that holds/manages info.
  - Facebook, LinkedIn, ...
  - And provides access API.
- Client Application
  - "The product you implemented."
  - Wants to read/update
    - "On your behalf"
    - The data the data that the Resource Server maintains, e.g. posts, status, tweets, ...
- Authorization Server
  - Grants/rejects authorization
  - Based on Resource Owner decisions.
  - Usually (logically) the same as Resource Server.

OAuth 2.0 defines the following roles of users and applications:

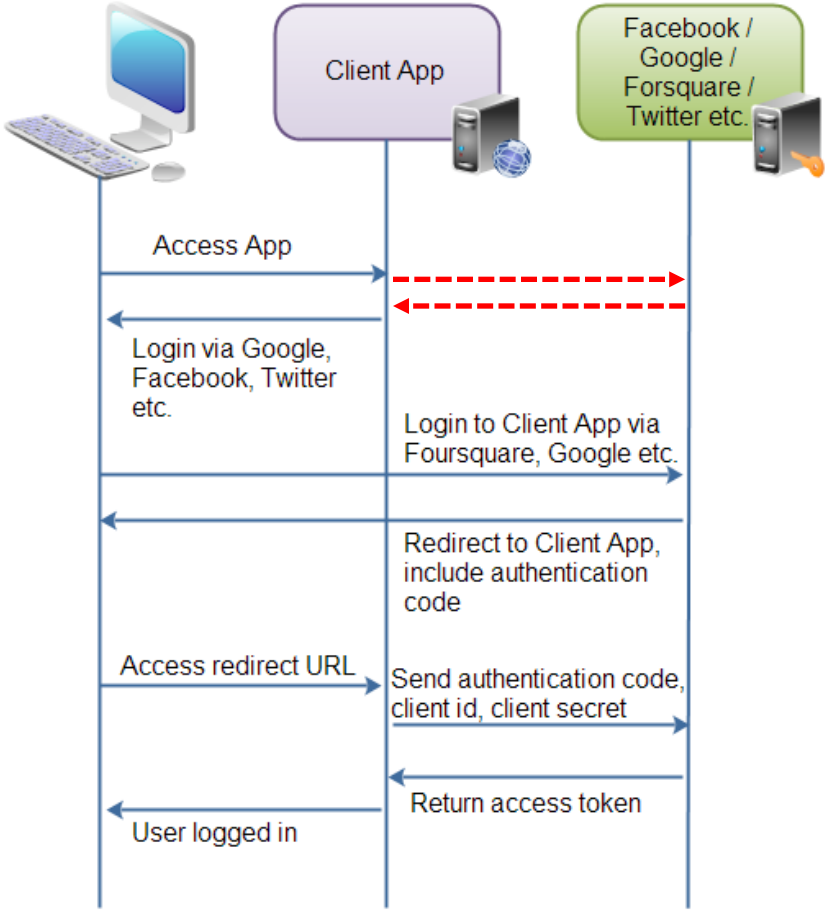
- Resource Owner
- Resource Server
- Client Application
- Authorization Server

These roles are illustrated in this diagram:

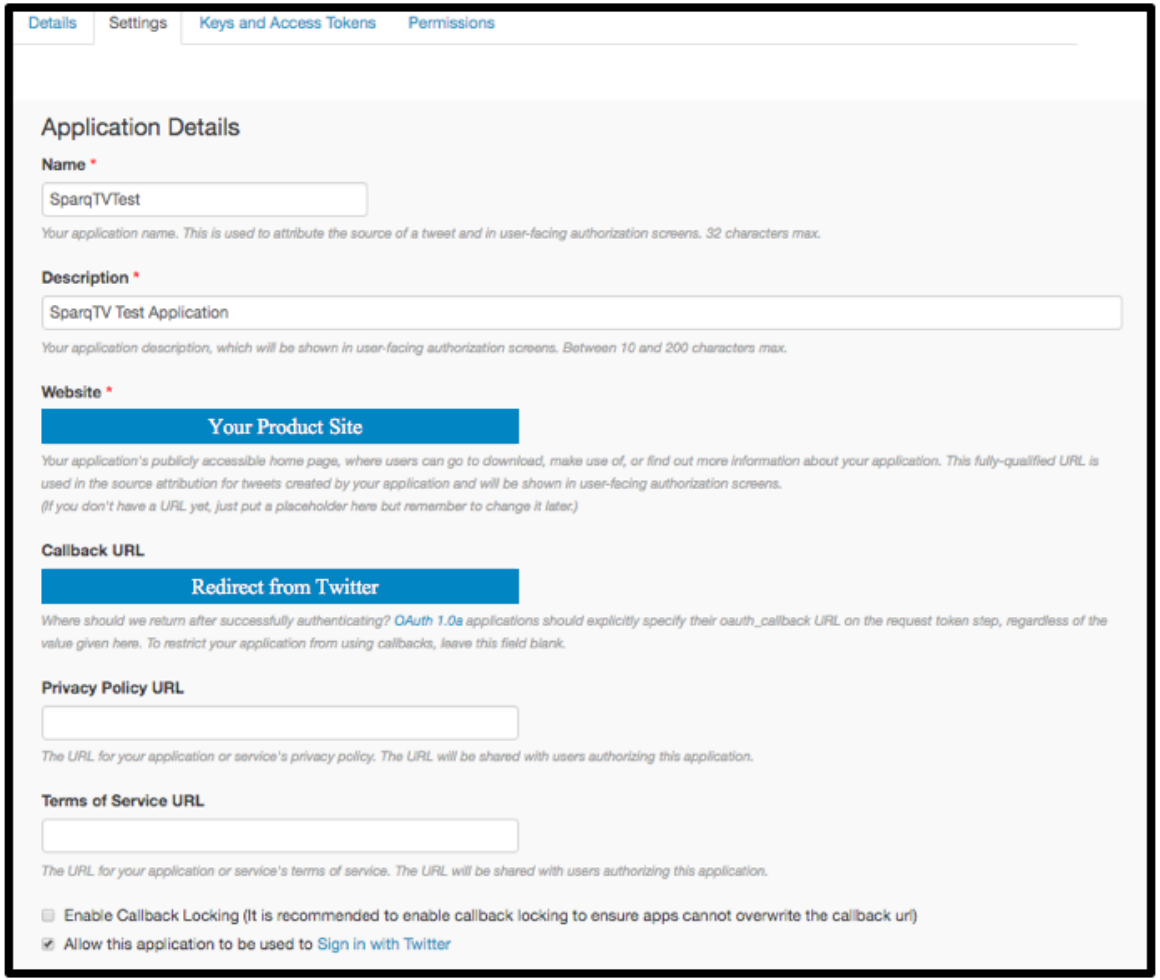


Roles/Flows

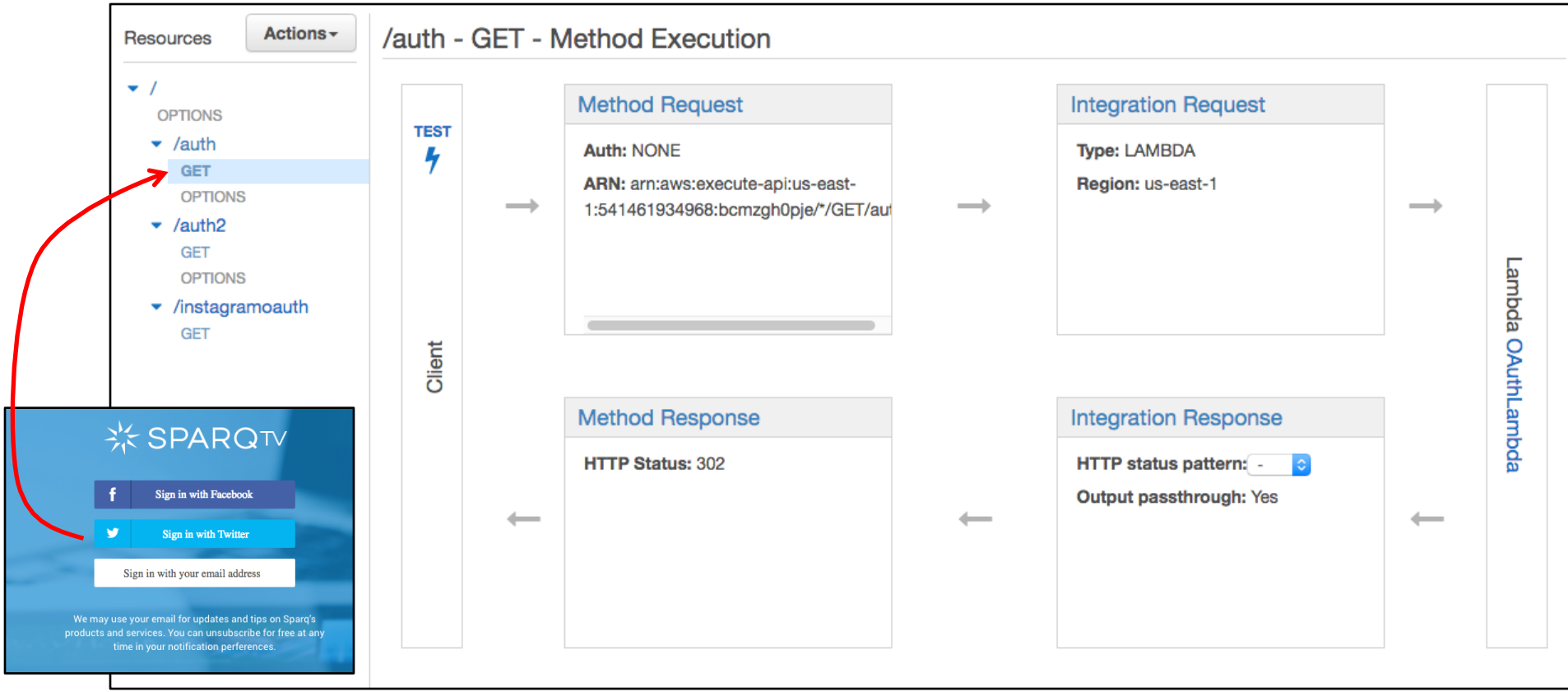
- User Clicks “Logon with XXX”
  - Redirect user to XXX
  - With Client App application ID.
  - And permissions app requests.
  - Code MAY have to call Resource Server to get a token to include in redirect URL.
- Browser redirected to XXX
  - Logon on to XXX prompt.
  - Followed by “do you grant permission to …”
  - Clicking button drives a redirect back to Client App. URL contains a temporary token.
- User/Browser
  - Redirected to Client App URL with token.
  - Client App calls XXX API
  - Obtains access token.
  - Returns to User.
- Client App can use access token on API calls.



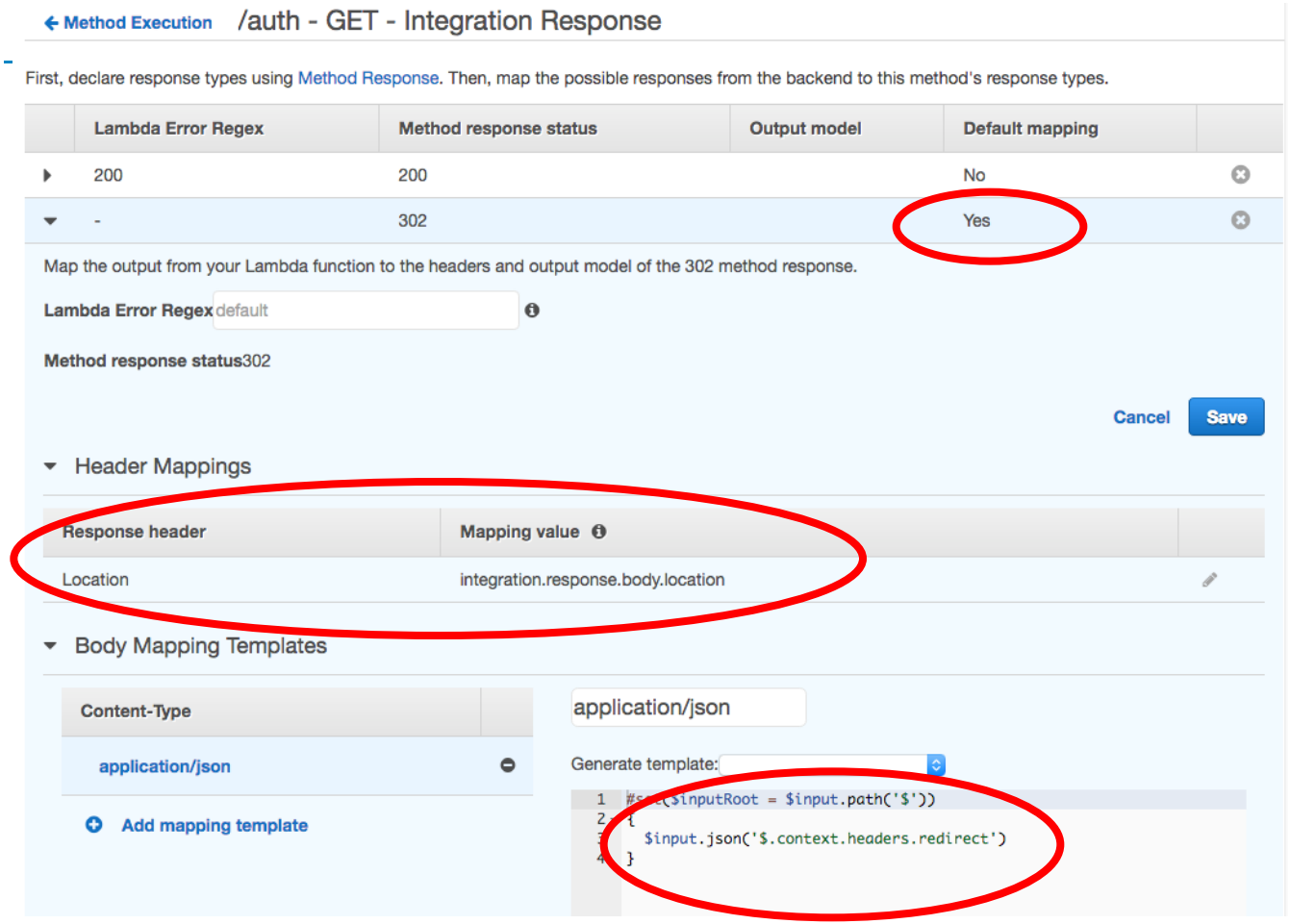
Configuring Your Application at XXX



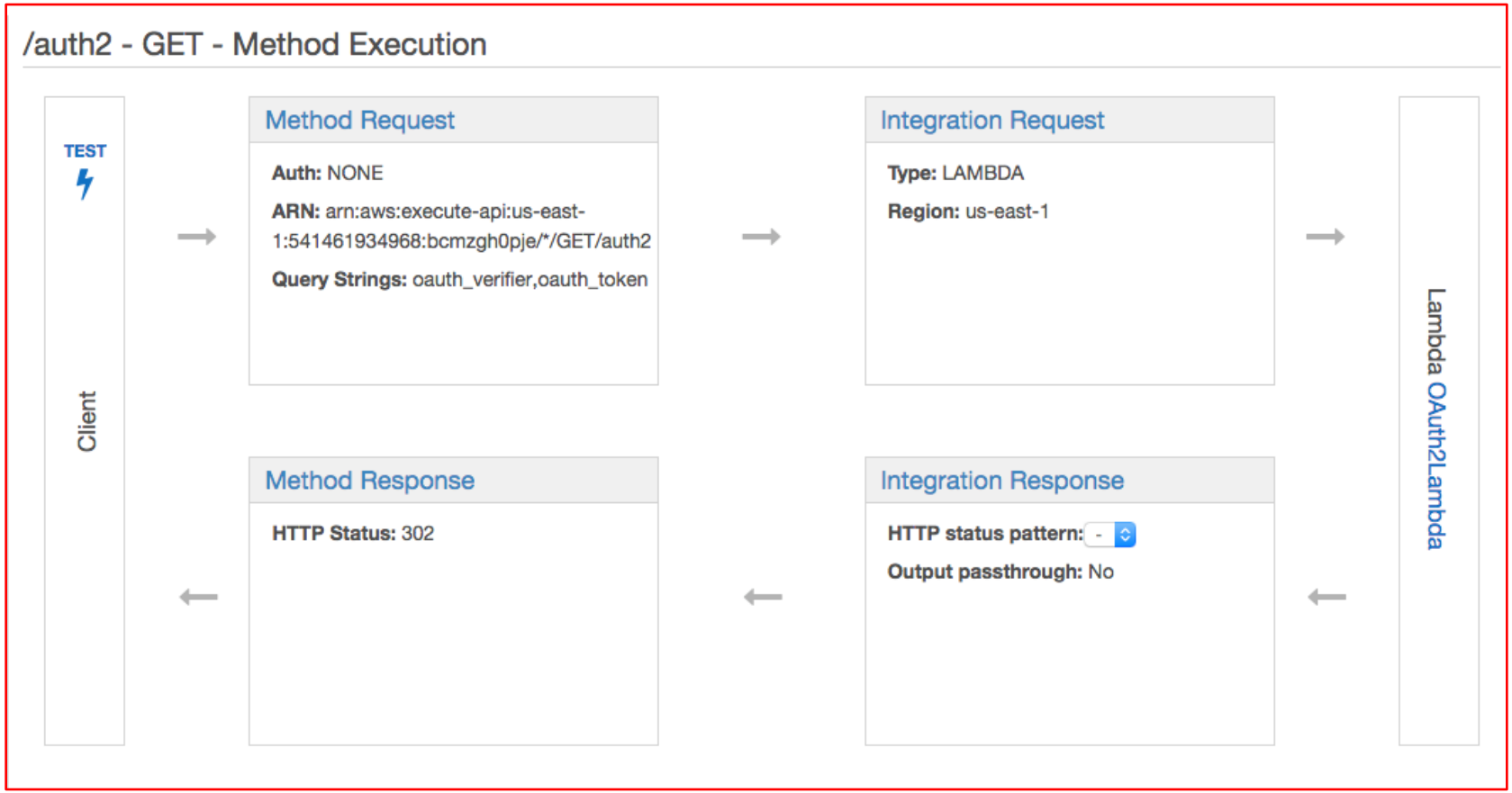
Twitter Example – Step 1



Twitter Example - Step 1 (Cont)



Step 2 – Same Basic Pattern



```
public class TwitterConnector {

    private static final String S3_BUCKET =
    // Consumer Key (API Key)
    private static final String TWITTER_CONSUMER_KEY =
    // Consumer Secret (API Secret)
    private static final String TWITTER_CONSUMER_SECRET =
    // Info: Callback URL = S3_BUCKET + "/dev/auth2"

    /*
     * Interactions with Twitter require a RequestToken. There are two forms: 1.
     * Just information identifying the application. 2. Also including user
     * authentication information. This class represents the simple version.
     */
    public static class SimpleRequestToken {
        String token;
        String secret;

        public SimpleRequestToken(String t, String s) {
            this.token = t;
            this.secret = s;
        }
    }

    /*
     * This is the persistent OAuth2 credential that we get back from completing
     * the successful, second step in the Twitter login process. We save this
     * for subsequent interactions.
     *
     * TODO I am not sure if we should embed the class inside the connector.
     *
     */
    public class TwitterAuth {
        public String oauth_token;
        public String oauth_verifier;
    }
```

Base for our Web Site

Key (ID for App)

Password

```
public class TwitterConnector {

    private static final String S3_BUCKET =
    // Consumer Key (API Key)
    private static final String TWITTER_CONSUMER_KEY =
    // Consumer Secret (API Secret)
    private static final String TWITTER_CONSUMER_SECRET =
    // Info: Callback URL = S3_BUCKET + "/dev/auth2"

    /*
     * Interactions with Twitter require a RequestToken. There are two forms: 1.
     * Just information identifying the application. 2. Also including user
     * authentication information. This class represents the simple
     */
    public static class SimpleRequestToken {
        String token;
        String secret;

        public SimpleRequestToken(String t, String s) {
            this.token = t;
            this.secret = s;
        }
    }

    /*
     * This is the persistent OAuth2 credential that we get back
     * the successful, second step in the Twitter login process.
     * for subsequent interactions.
     *
     * TODO I am not sure if we should embed the class inside the connector.
     *
     */
    public class TwitterAuth {
        public String oauth_token;
        public String oauth_verifier;
    }
```

Base for our Web Site

Key (ID for App)

Password

Hard coding these as strings is in the code is a bad idea. But, I was lazy. Lazy is how security breaches occur.

```
private static boolean initializeConnector(String access_id, String access_token) {

    // Turns out that every Lambda function request does not get a fresh
    // JVM. So, consecutive requests were reusing the wrong request token.
    // so for now we just reinitialize every time.
    // TODO Fix this later.
    //
    // if (TwitterConnector.connectorInitialized == false) {

    logger.debug("Initializing the Twitter Connector");

    try {
        TwitterConnector.setup();
        TwitterConnector.connectorInitialized = true;

        logger.debug("Classes initialized.");

        // TODO We need to move this configuration information to somewhere
        // else.
        //
        twitterProperties = new Properties();
        twitterProperties.setProperty("oauth.consumerKey", TWITTER_CONSUMER_KEY);
        twitterProperties.setProperty("oauth.consumerSecret", TWITTER_CONSUMER_SECRET);
        if (access_id != null) {
            twitterProperties.setProperty("oauth.accessToken", access_id);
        }
        if (access_token != null) {
            twitterProperties.setProperty("oauth.accessTokenSecret", access_token);
        }

        logger.debug("Initialized properties.");

        PropertyConfiguration pc = new PropertyConfiguration(twitterProperties);
        twitter = new TwitterFactory(pc).getInstance();
        logger.debug("Initialized Twitter4J.");

    } catch (Exception e) {
        logger.warn("Exception while initializing Twitter Connector, e = " + e);
        connectorInitialized = false;
        twitterProperties = null;
        twitter = null;
    }

    logger.debug("Successfully initialized Twitter Connector.");
    // }

    return connectorInitialized;
}
```

```
public static HTTPRedirect twitterRedirect1() {

    HTTPRedirect result = null;
    UUID statusUUID = null;

    try {
        logger.debug("Step 1 in Twitter OAuth2 login.");

        TwitterConnector.initializeConnector();

        logger.debug("Attempting to get Twitter request token.");
        RequestToken requestToken = twitter.getOAuthRequestToken();

        // TODO Writing this info to a log file is probably a security
        // exposure.
        logger.debug("After getOAuthRequestToken = " + "token=" + requestToken.getToken() + ", " + "secret="
                + requestToken.getToken());

        // Save the request token, which we will need for the second phase.
        SimpleRequestToken st = new TwitterConnector.SimpleRequestToken(requestToken.getToken(),
                requestToken.getTokenSecret());
        statusUUID = saveState(st);

        if (statusUUID != null) {
            String authorizationUrl = requestToken.getAuthenticationURL();
            result = new HTTPRedirect(authorizationUrl, HTTPRedirect.success);
        } else {
            logger.warn("Could not save SimpleRequestToken in SessionStorage");
        }
    } catch (Exception e) {
        logger.warn("twitterRedirect1 failed with exception", e);
    }

    if (result == null) {
        logger.warn("Could not get redirect URL. Returning error URL.");
        result = new HTTPRedirect(failureUrl, HTTPRedirect.failure);
    }

    return result;

}
```

```
public static HTTPRedirect twitterRedirect2(TwitterAuth accessToken) {
    long id;
    twitter4j.User u = null;
    AccessToken authorizationToken = null;
    String s = null;

    try {

        logger.debug("Attempting step 2 in Twitter Login. accessToken = " + JSONUtil.toPrettyJSON(accessToken));

        TwitterConnector.initializeConnector();

        logger.debug("Attempting to load state from step 1.");
        SimpleRequestToken st = TwitterConnector.loadState(accessToken.oauth_token);

        if (st != null) {
            logger.debug("Got simple request token = " + JSONUtil.toPrettyJSON(st));
            RequestToken rt = new RequestToken(st.token, st.secret);

            logger.debug("Attempting to get Authorization Token");
            authorizationToken = twitter.getOAuthAccessToken(rt, accessToken.oauth_verifier);
            logger.debug("Got Authorization Token = " + JSONUtil.toPrettyJSON(authorizationToken));
            s = JSONUtil.toSimpleJSON(authorizationToken);

            logger.debug("Trying to get info from Twitter.");
            id = twitter.getId();
            logger.debug("ID = " + id);
            u = twitter.showUser(id);
            logger.debug("User = " + prettyGson.toJson(u));
        }
    } catch (Exception e) {
        logger.debug("Exception = " + e);
        e.printStackTrace();
        u = null;
    }

    HTTPRedirect r;
    if (u != null) {
        r = new HTTPRedirect("http://" + S3_BUCKET + "/#/twitterintegration?name=" + u.getName() + "&id="
                + u.getId() + "&token=" + authorizationToken.getToken() + "&tokenSecret="
                + authorizationToken.getTokenSecret() + "&result=true", HTTPRedirect.success);
    } else {
        r = new HTTPRedirect(TwitterConnector.failureUrl, HTTPRedirect.success);

    }
    logger.debug("returning " + JSONUtil.toPrettyJSON(r));
    return r;

}
```

OAuth Process Allows Us to

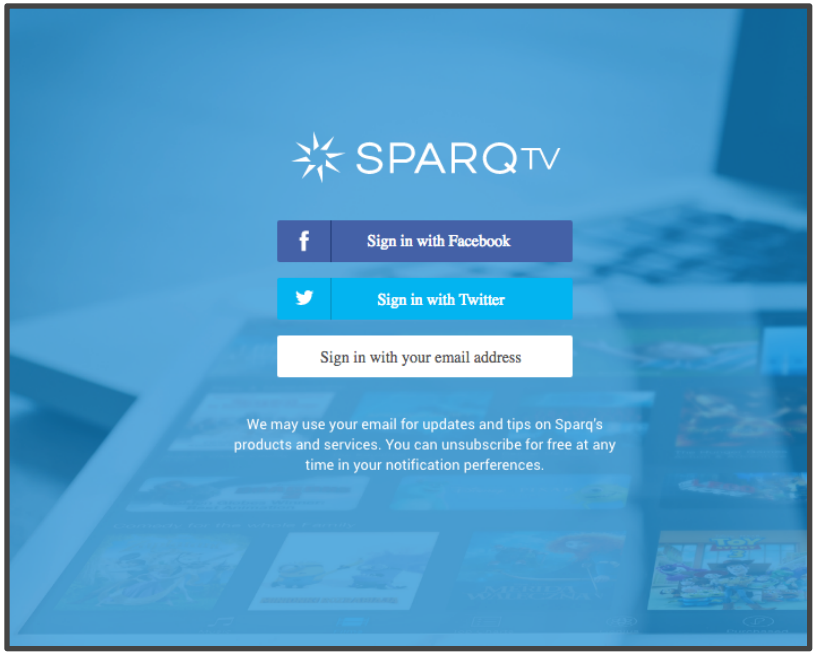
- Get user profile info
  - Name
  - Email
  - Location
  - … …
- Get user Social network
  - User follows ...
  - Follows user...
- Tweet on user's behalf when something happens in my application.

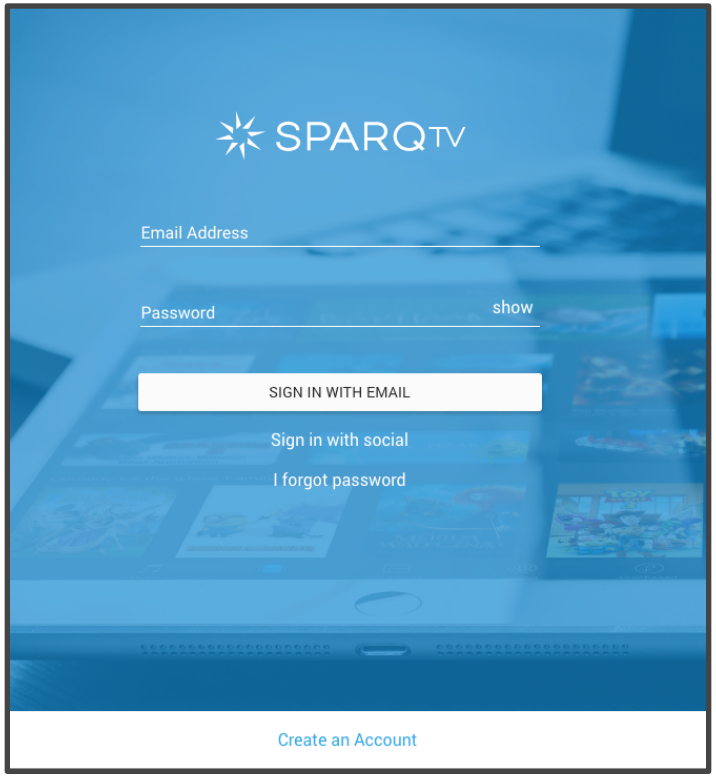
```
public static int getRetweets(String access_id, String access_token, String post_id) {
    TwitterConnector.initializeConnector(access_id, access_token);
    Status result = null;
    int retweets = 0;
    try {
        long id = Long.parseLong(post_id);
        result = twitter.showStatus(id);
        retweets = result.getRetweetCount();
    } catch (Exception e) {
        e.printStackTrace();
    } catch (OutOfMemoryError e) {
        e.printStackTrace();
    }
    return retweets;
}

public static List<Object> getFollowers(String access_id, String access_token) {
    TwitterConnector.initializeConnector(access_id, access_token);
    ArrayList<Object> followers = new ArrayList<>();
    PagableResponseList<User> all = null;
    long userId = Long.parseLong(access_id.substring(0, access_id.indexOf('-')));
    try {
        all = twitter.getFollowersList(userId, -1);
        followers.addAll(all);
        return followers;
    } catch (Exception e) {
        e.printStackTrace();
    } catch (OutOfMemoryError e) {
        e.printStackTrace();
    }
    return followers;
}
```

Social Media

Register and Logon

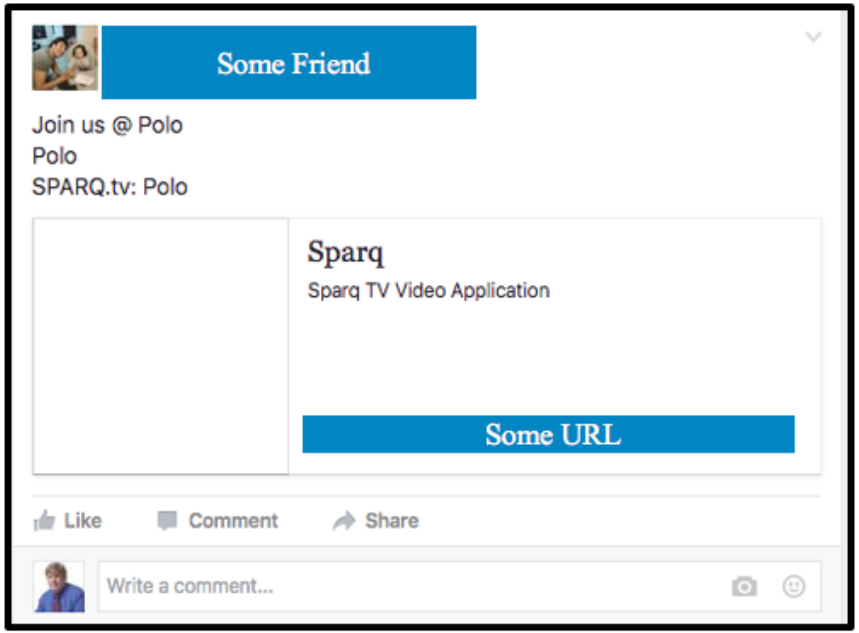
Simplifies and expedites logon and registering for your product.

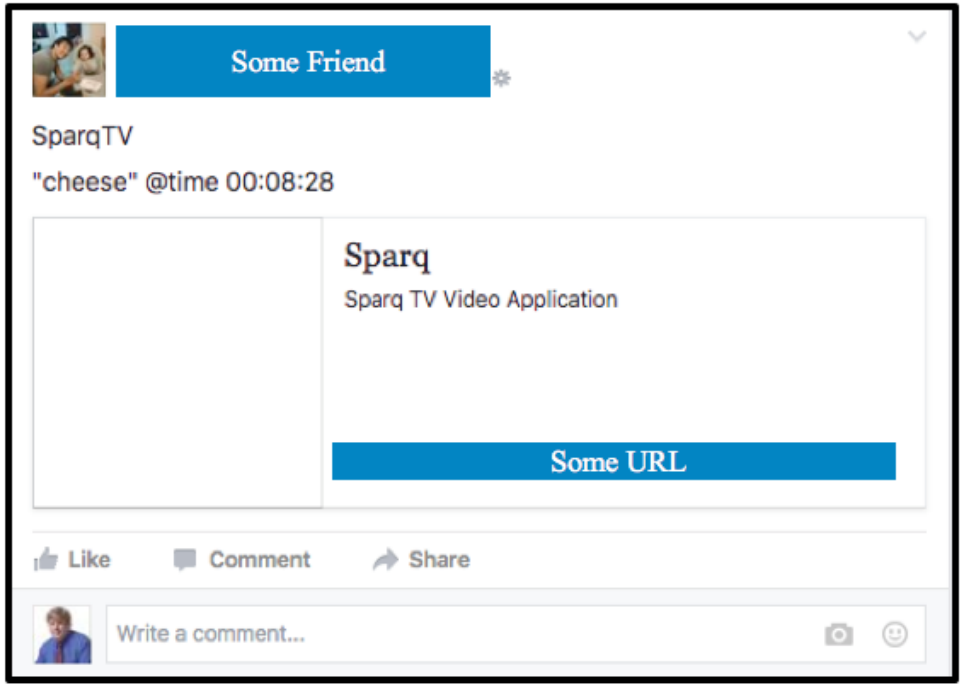




Federate Social Experience

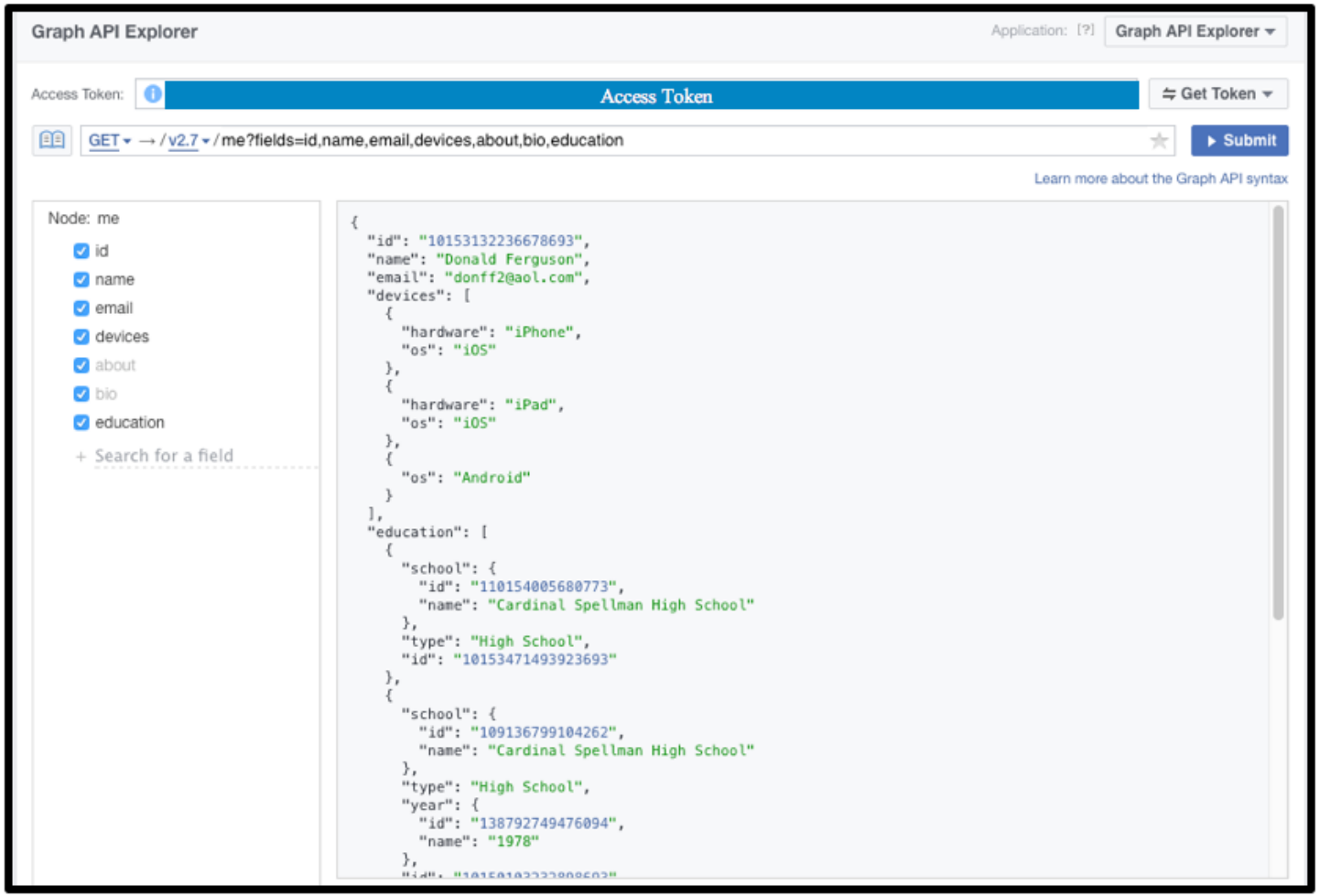
Expands reach of your product to attract new users.
Simplifies your customers "sharing" what they are doing (no "cut and paste")





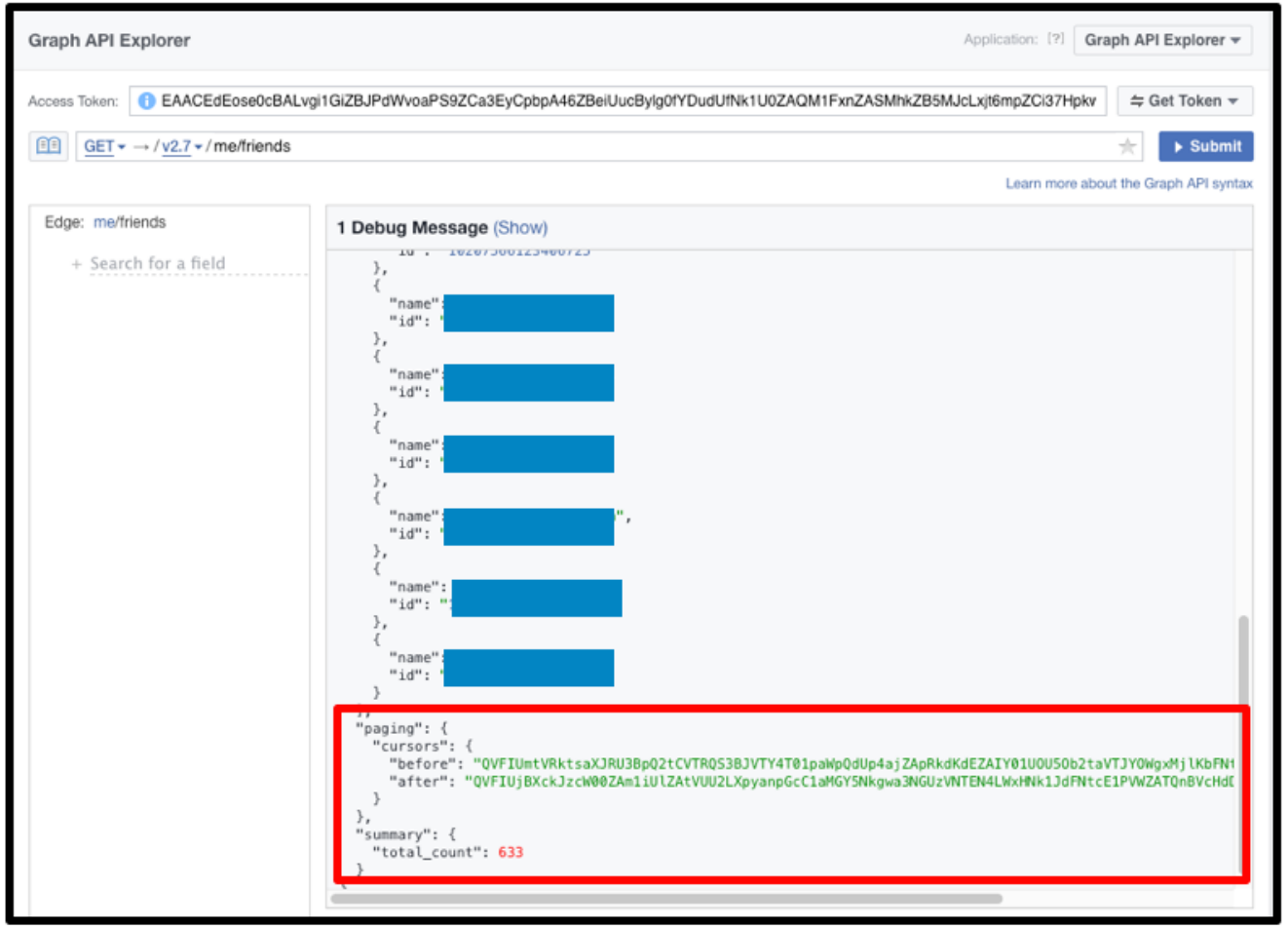
Graph API

Provides user insights to enable customized, optimized interactions and support.

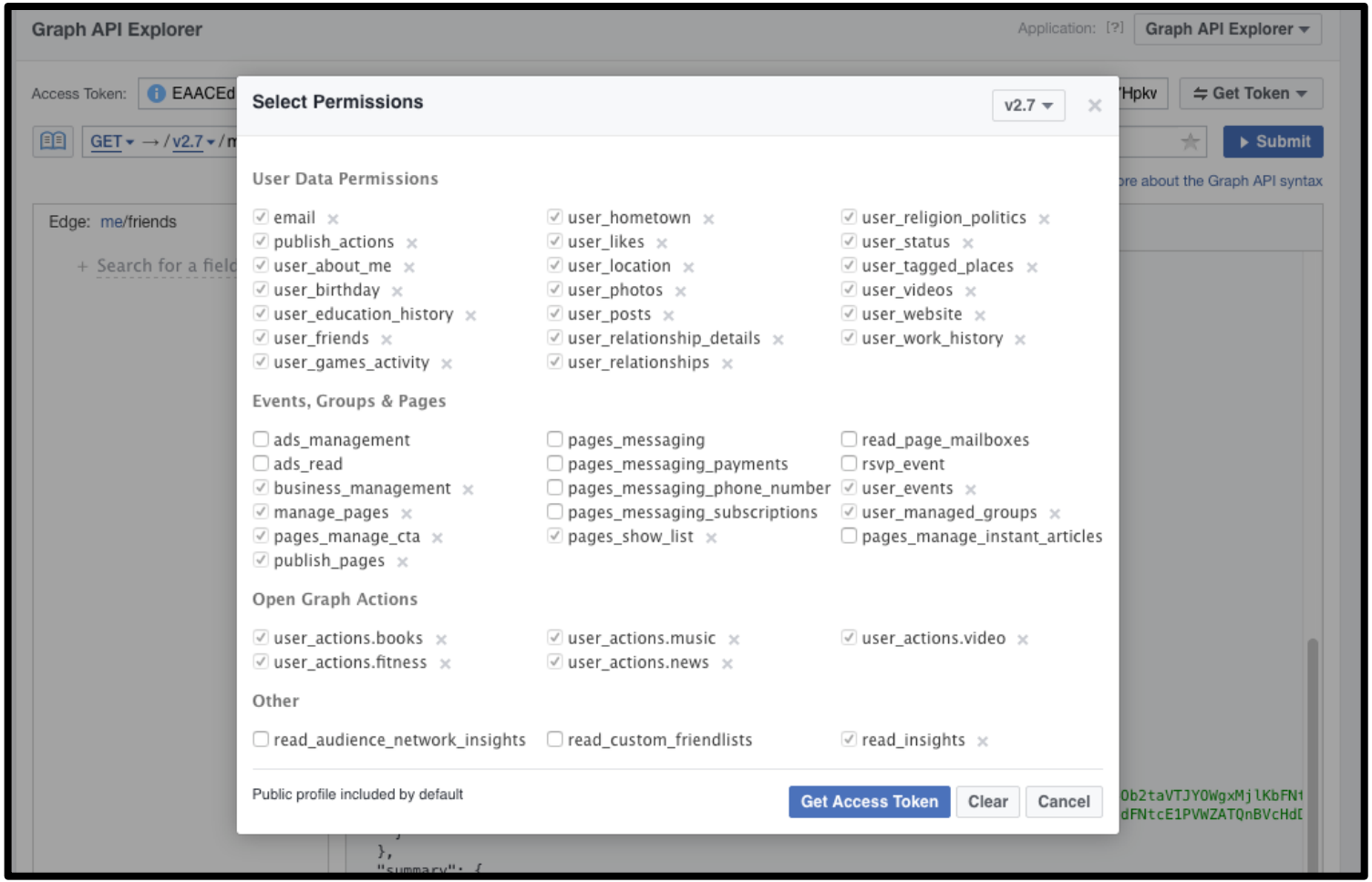


Graph API

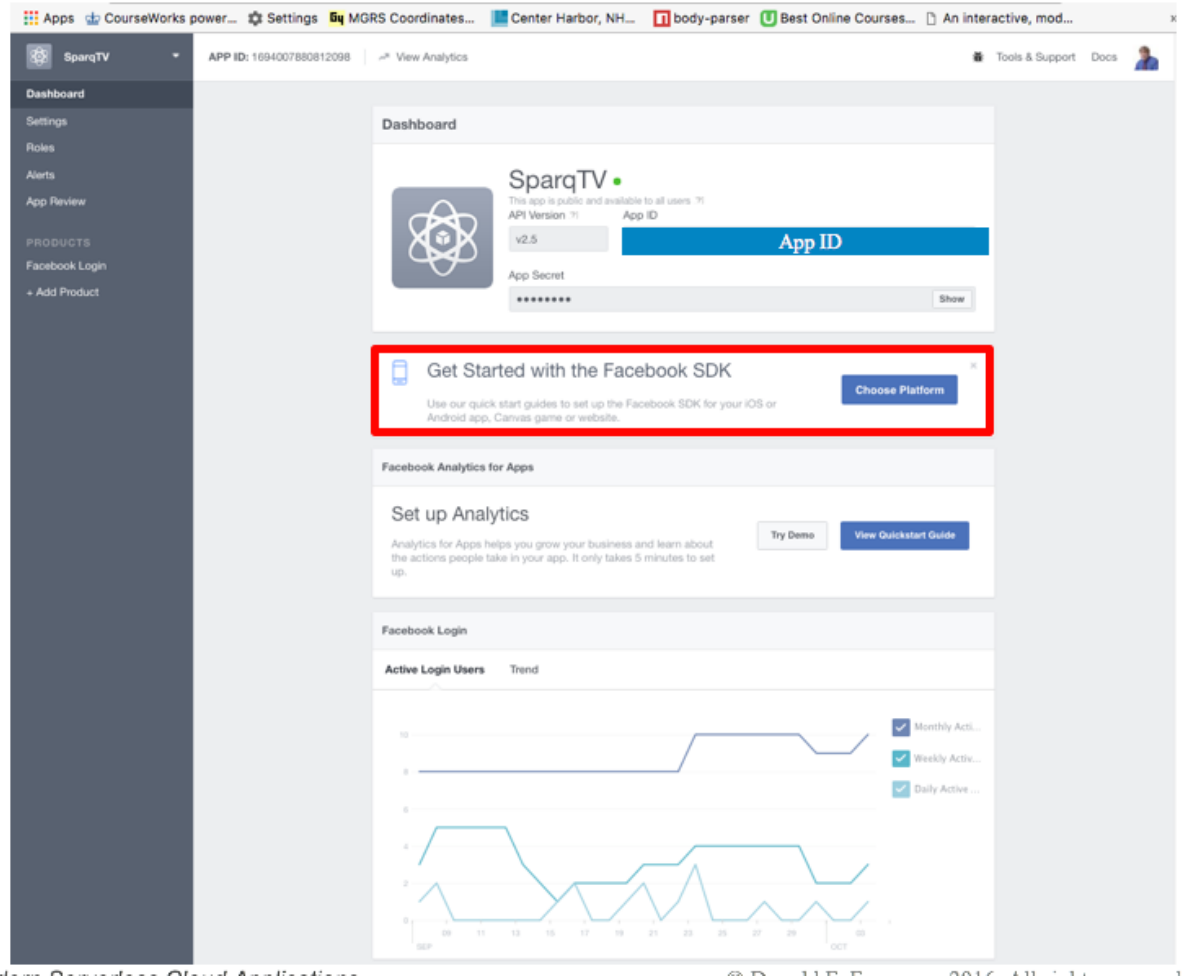
Simplifies inviting and finding friends to use/ already using the product.



Graph API Permissions



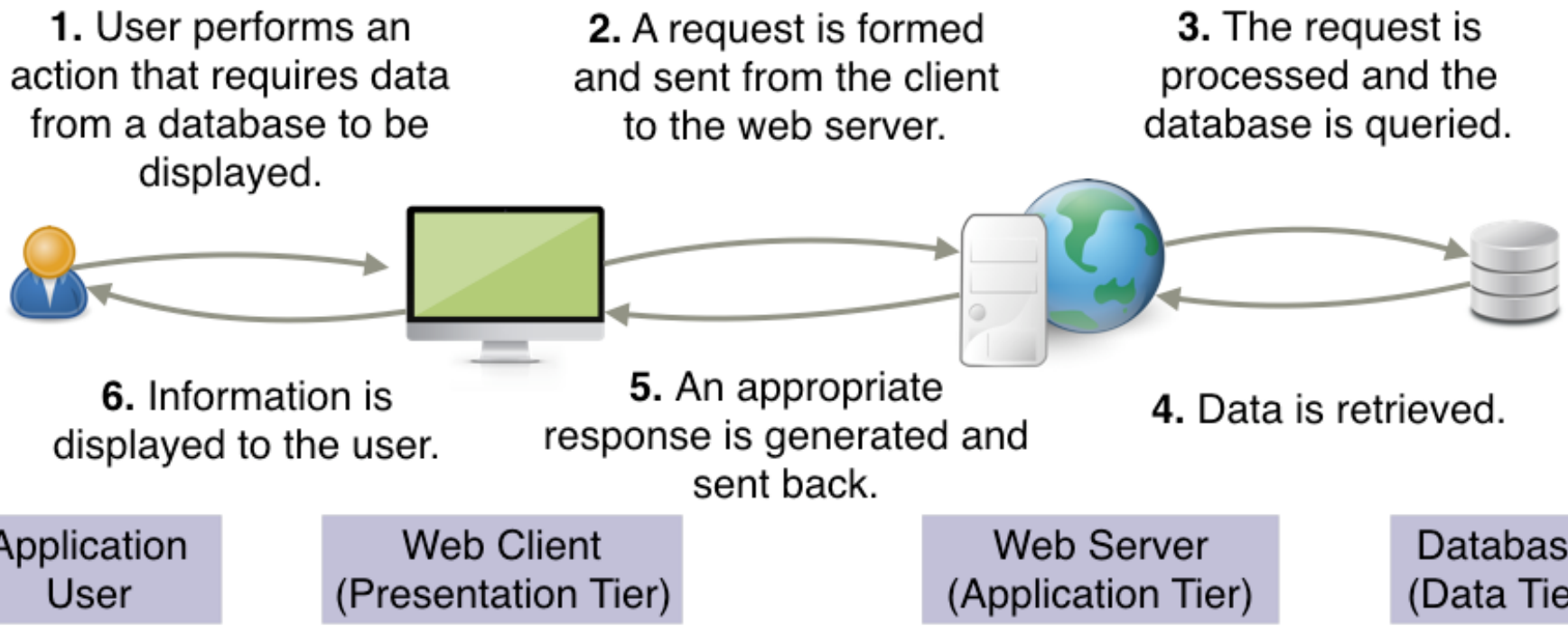
Define an App



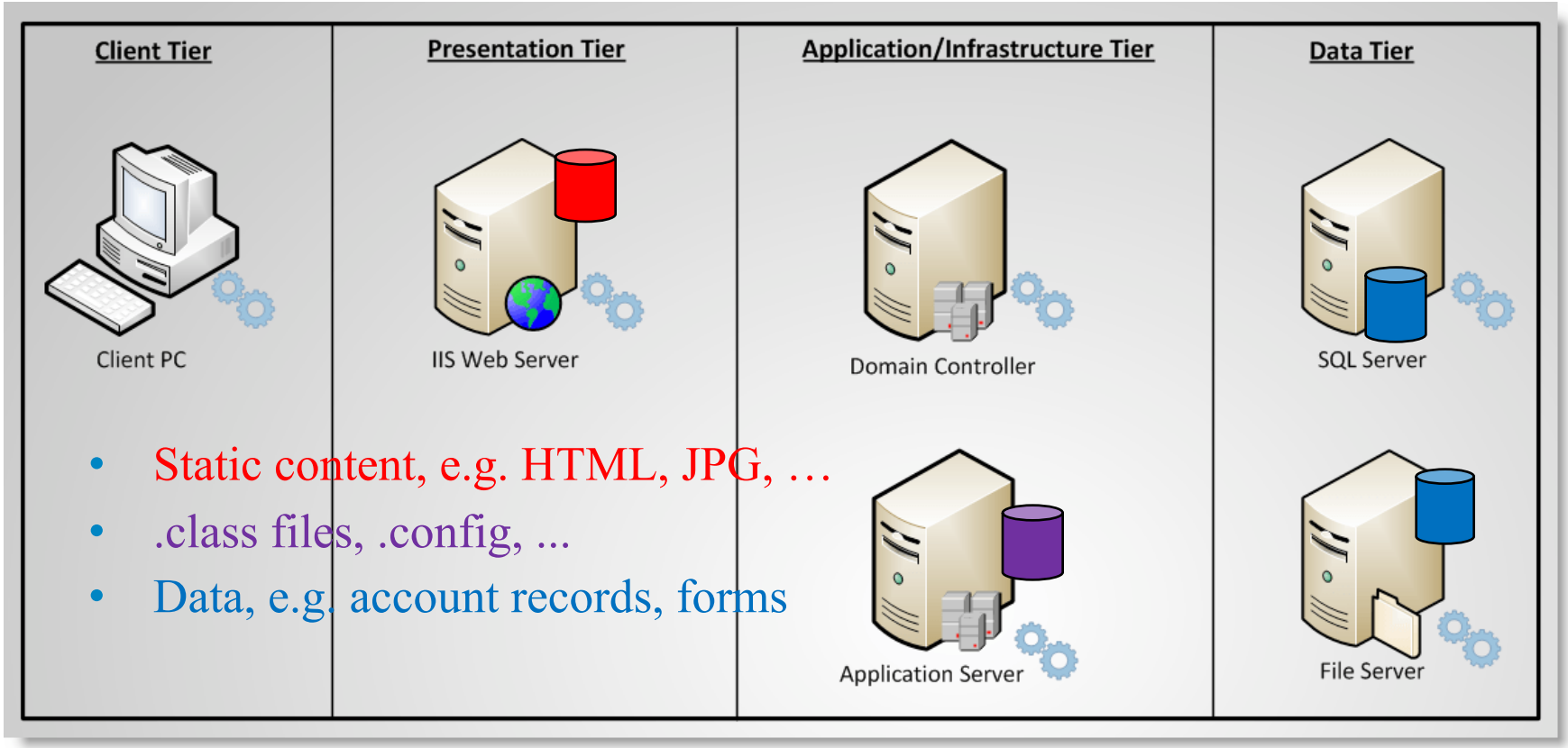
Graph API Explorer Demo

Static Website

Web Application Basic Concepts

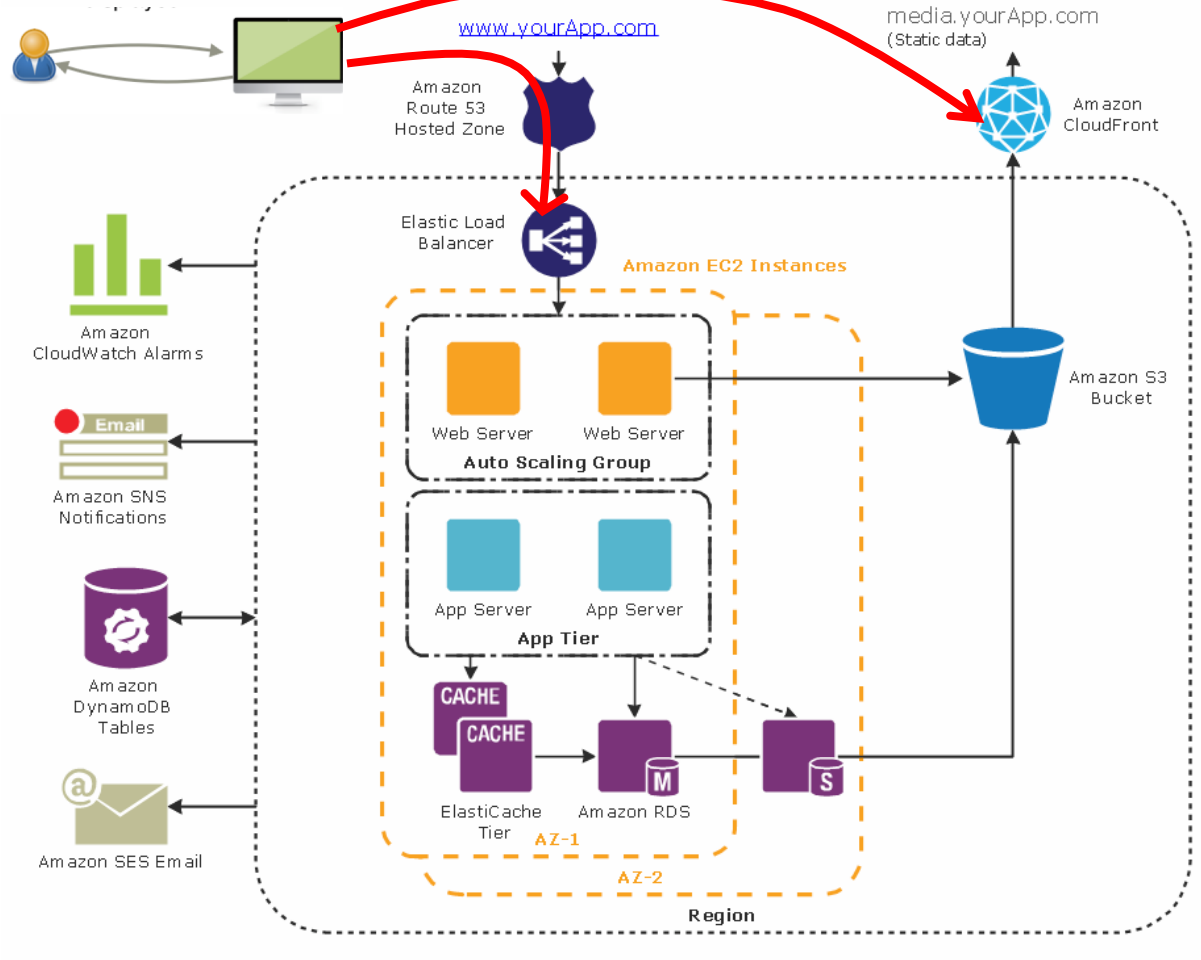


Physical Topology

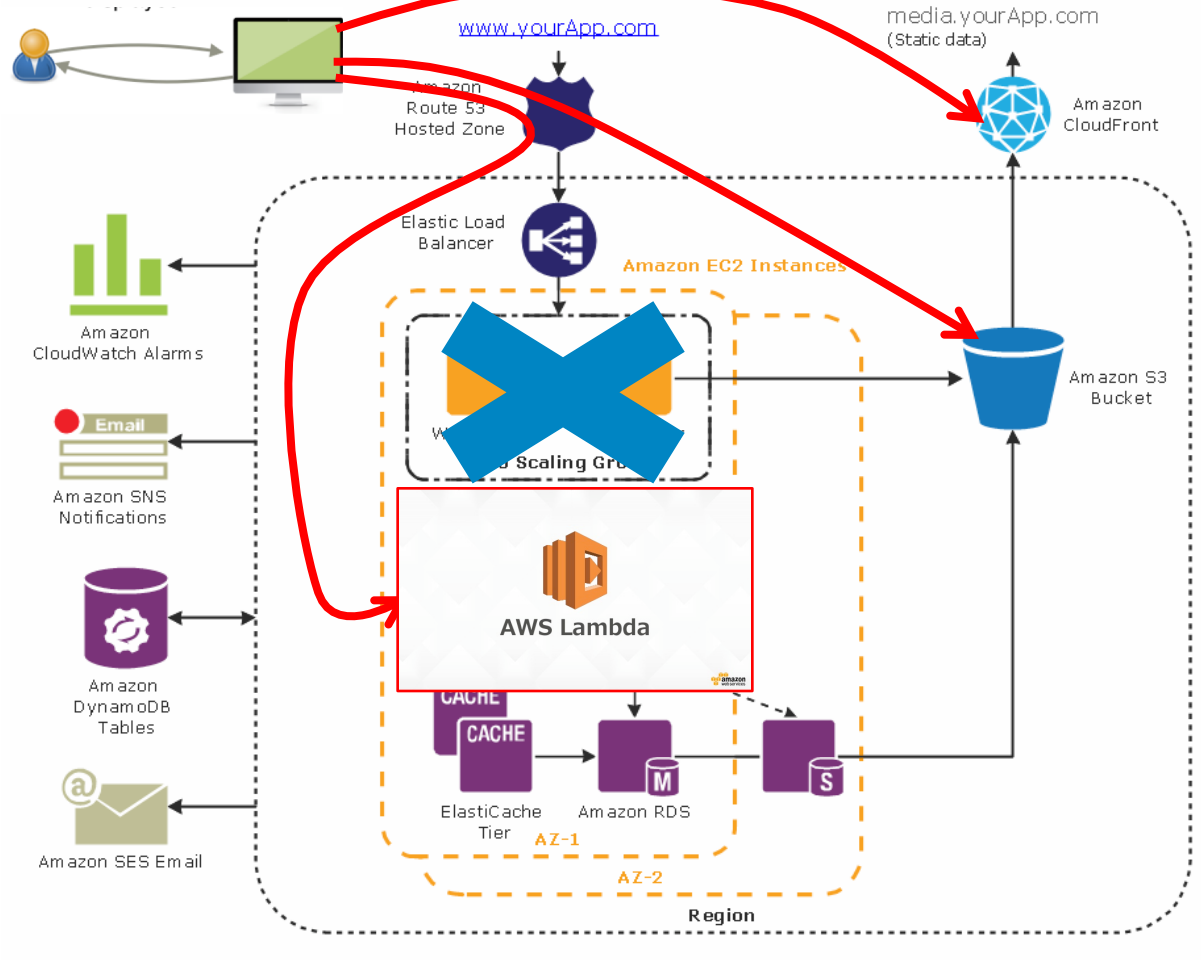


- Static content, e.g. HTML, JPG, …
- .class files, .config, ...
- Data, e.g. account records, forms

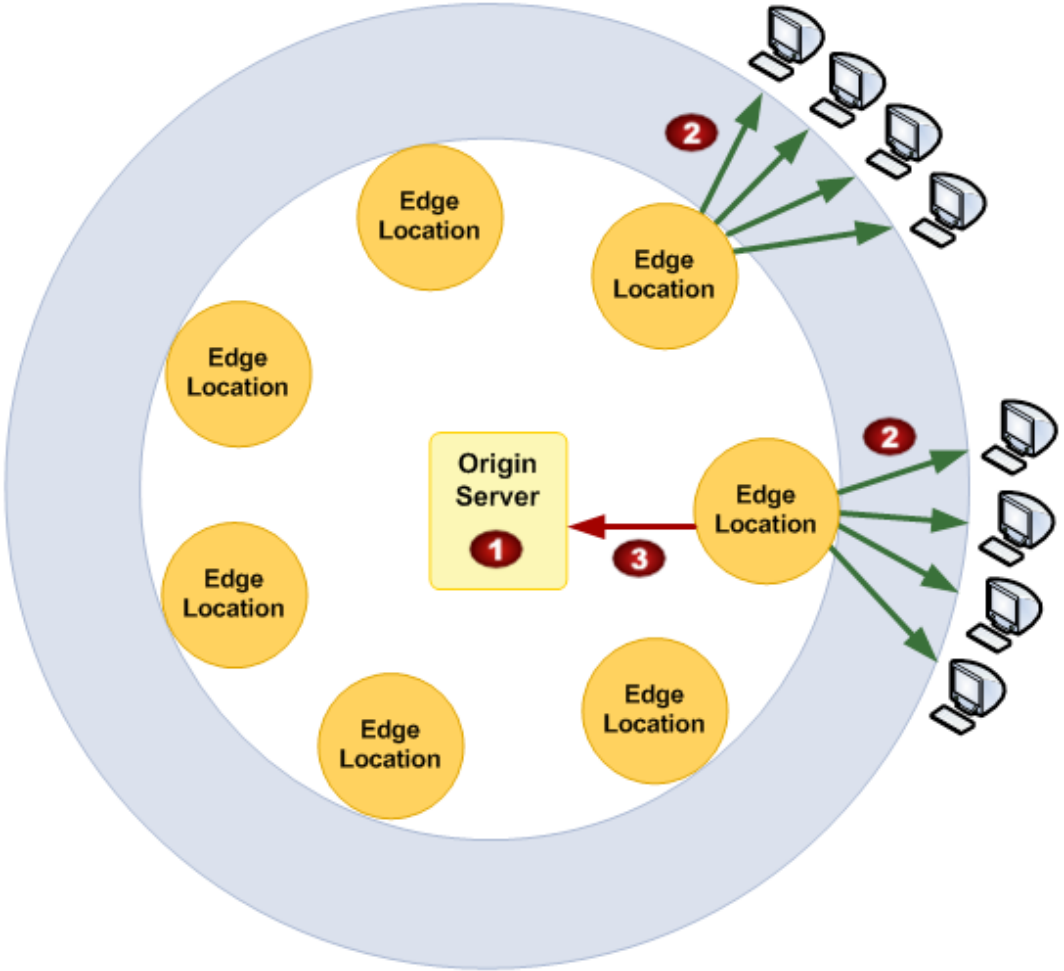
“Traditional” AWS Topology



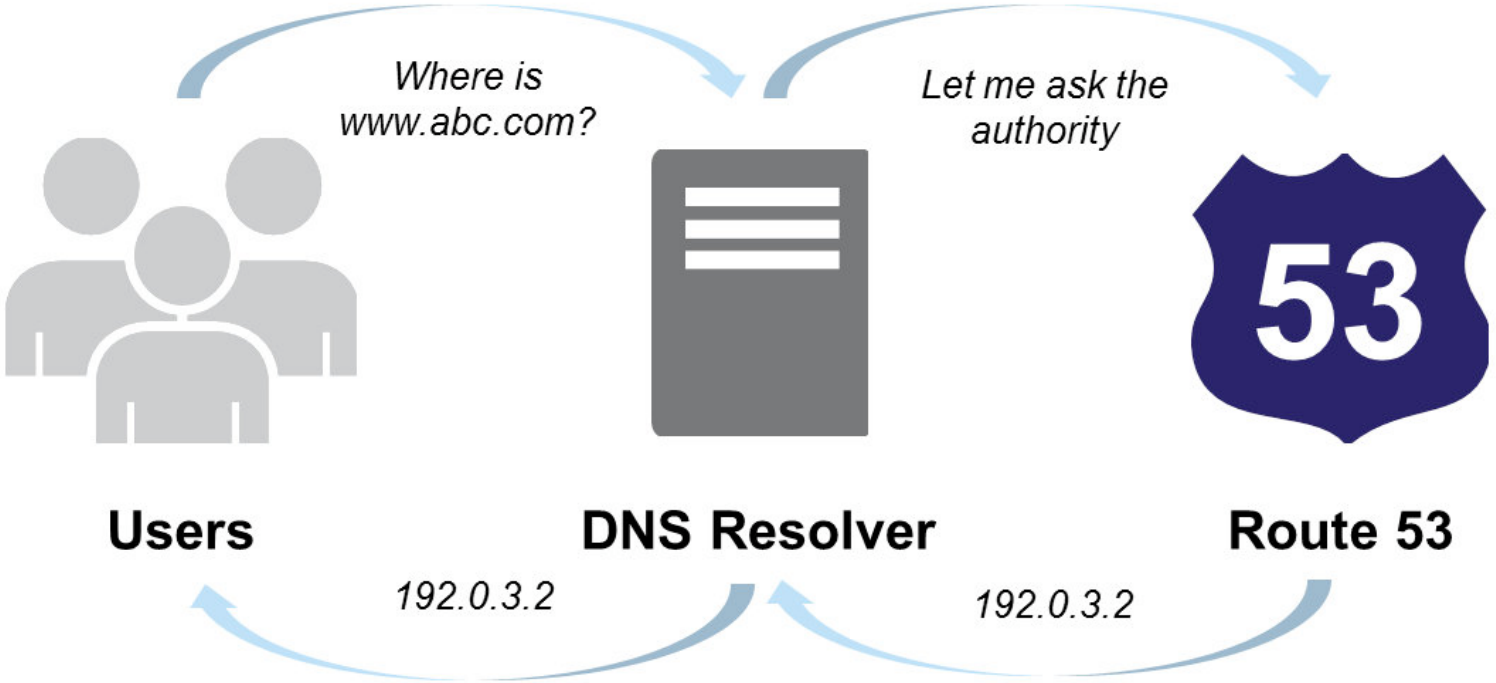
Our Model



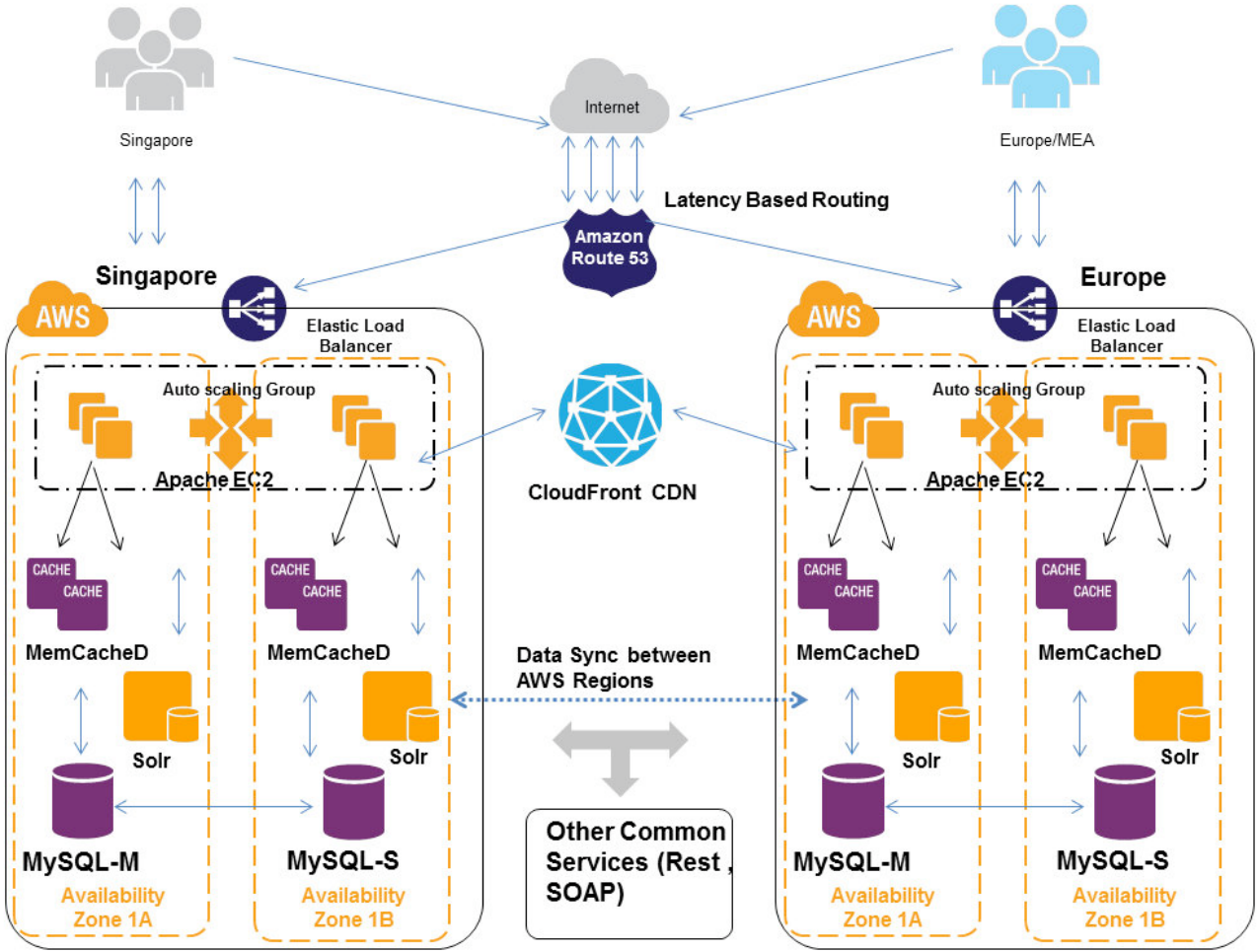
AWS CloudFront



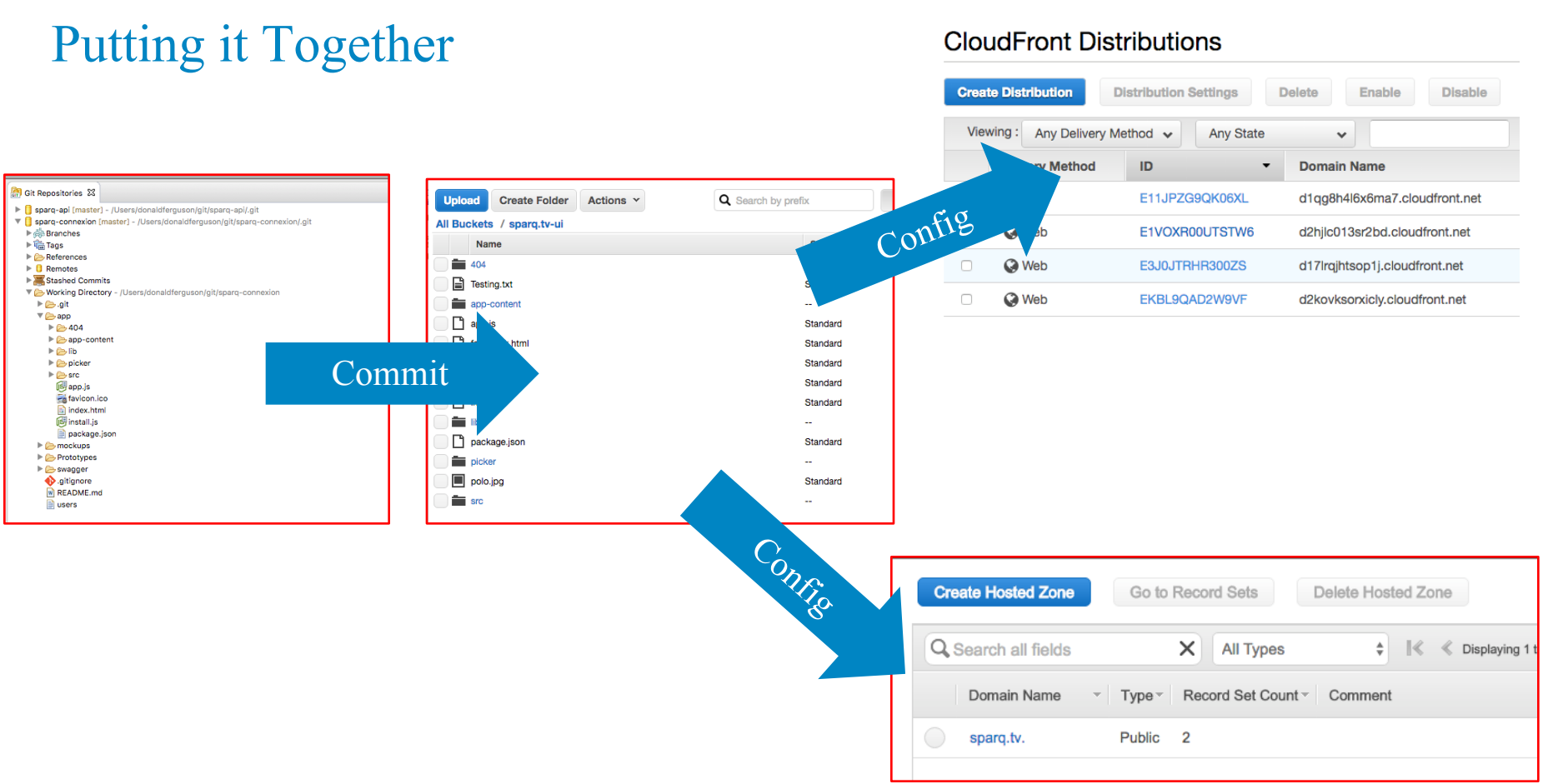
AWS Route 53



Example



Putting it Together



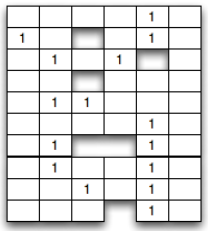
Graph Database (Intro)

Four NOSQL Categories

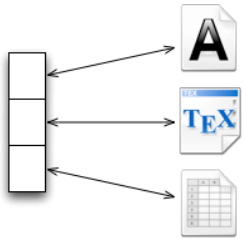
Key-Value

Graph DB

BigTable



Document



www.ideal.ece.utexas.edu/courses/ee380l.../**ppt/Neo4j.ppt**

Pros and Cons

- Strengths
  - Powerful data model. Ideal, and simpler, for scenarios
    - Social networks
    - Relationships between people and things, which are also related.
    - etc.
  - Fast
    - For connected data, can be many orders of magnitude faster than RDBMS
- Weaknesses:
  - Sharding
    - Though they *can* scale reasonably well
    - And for some domains you can shard too!

Overview http://graphdatabases.com/

The Labeled Property Graph Model --

- A labeled property graph is made up of nodes, relationships, properties, and labels.

- Nodes contain properties. Think of nodes as documents that store properties in the form of arbitrary key-value pairs.
  - In Neo4j, the keys are strings and the values are the Java string and primitive data types, plus arrays of these types.
  - Nodes can be tagged with one or more labels.
  - Labels group nodes together, and indicate the roles they play within the dataset.
- Relationships connect nodes and structure the graph.
  - A relationship always has a direction, a single name, and a start node and an end node—there are no dangling relationships
  - Like nodes, relationships can also have properties.

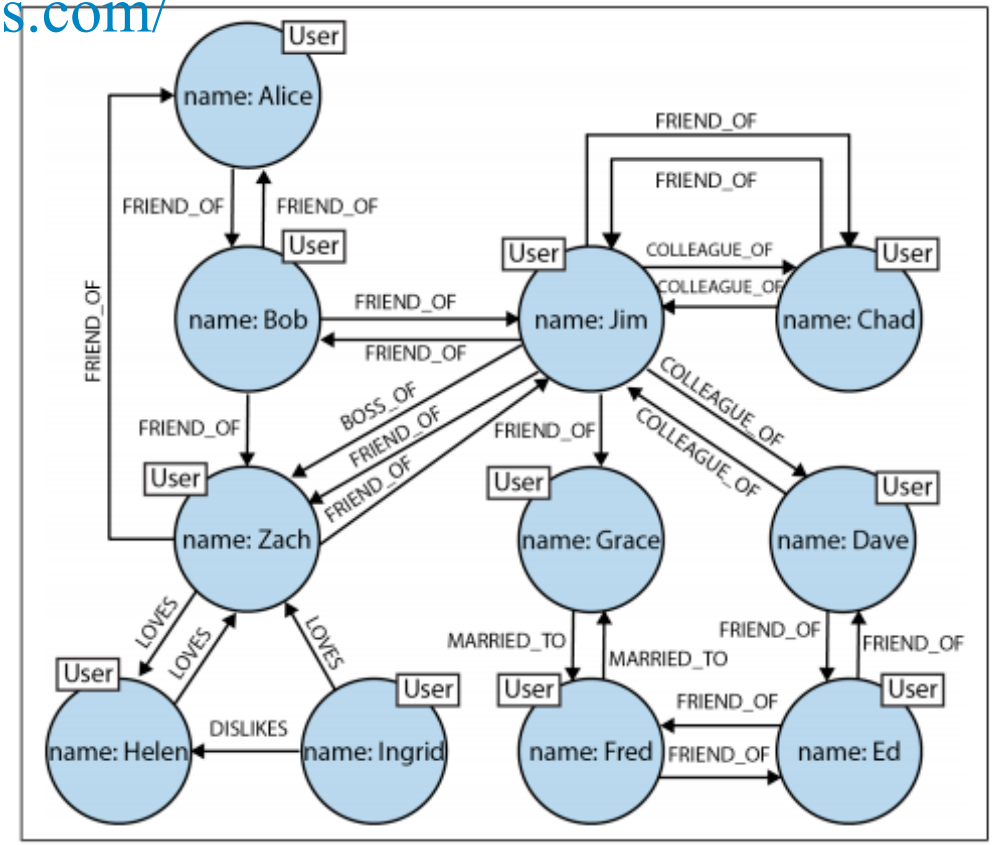


Figure 2-5. Easily modeling friends, colleagues, workers, and (unrequited) lovers in a graph

Social Network "path exists" Performance

- Experiment:
  - ~1k persons
  - Average 50 friends per person
  - `pathExists(a,b)` limited to depth 4
  - Caches warm to eliminate disk IO

	# persons	query time
Relational database	1000	2000ms
Neo4j	1000	2ms
Neo4j	1000000	2ms

www.ideal.ece.utexas.edu/courses/ee380l.../**ppt/Neo4j.ppt**

Neo4J-– Cypher DDL

```
CREATE (TheMatrix:Movie {title:'The Matrix', released:1999, tagline:'Welcome to the Real World'})
CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
CREATE (Carrie:Person {name:'Carrie-Anne Moss', born:1967})
CREATE (Laurence:Person {name:'Laurence Fishburne', born:1961})
CREATE (Hugo:Person {name:'Hugo Weaving', born:1960})
CREATE (AndyW:Person {name:'Andy Wachowski', born:1967})
CREATE (LanaW:Person {name:'Lana Wachowski', born:1965})
CREATE (JoelS:Person {name:'Joel Silver', born:1952})
CREATE
  (Keanu)-[:ACTED_IN {roles:['Neo']}]->(TheMatrix),
  (Carrie)-[:ACTED_IN {roles:['Trinity']}]->(TheMatrix),
  (Laurence)-[:ACTED_IN {roles:['Morpheus']}]->(TheMatrix),
  (Hugo)-[:ACTED_IN {roles:['Agent Smith']}]->(TheMatrix),
  (AndyW)-[:DIRECTED]->(TheMatrix),
  (LanaW)-[:DIRECTED]->(TheMatrix),
  (JoelS)-[:PRODUCED]->(TheMatrix)
```

Cypher DDL

```
CREATE (AFewGoodMen:Movie {title:"A Few Good Men", released:1992, tagline:"In the heart of the nation's capital
CREATE (TomC:Person {name:'Tom Cruise', born:1962})
CREATE (JackN:Person {name:'Jack Nicholson', born:1937})
CREATE (DemiM:Person {name:'Demi Moore', born:1962})
CREATE (KevinB:Person {name:'Kevin Bacon', born:1958})
CREATE (KieferS:Person {name:'Kiefer Sutherland', born:1966})
CREATE (NoahW:Person {name:'Noah Wyle', born:1971})
CREATE (CubaG:Person {name:'Cuba Gooding Jr.', born:1968})
CREATE (KevinP:Person {name:'Kevin Pollak', born:1957})
CREATE (JTW:Person {name:'J.T. Walsh', born:1943})
CREATE (JamesM:Person {name:'James Marshall', born:1967})
CREATE (ChristopherG:Person {name:'Christopher Guest', born:1948})
CREATE (RobR:Person {name:'Rob Reiner', born:1947})
CREATE (AaronS:Person {name:'Aaron Sorkin', born:1961})
CREATE
```

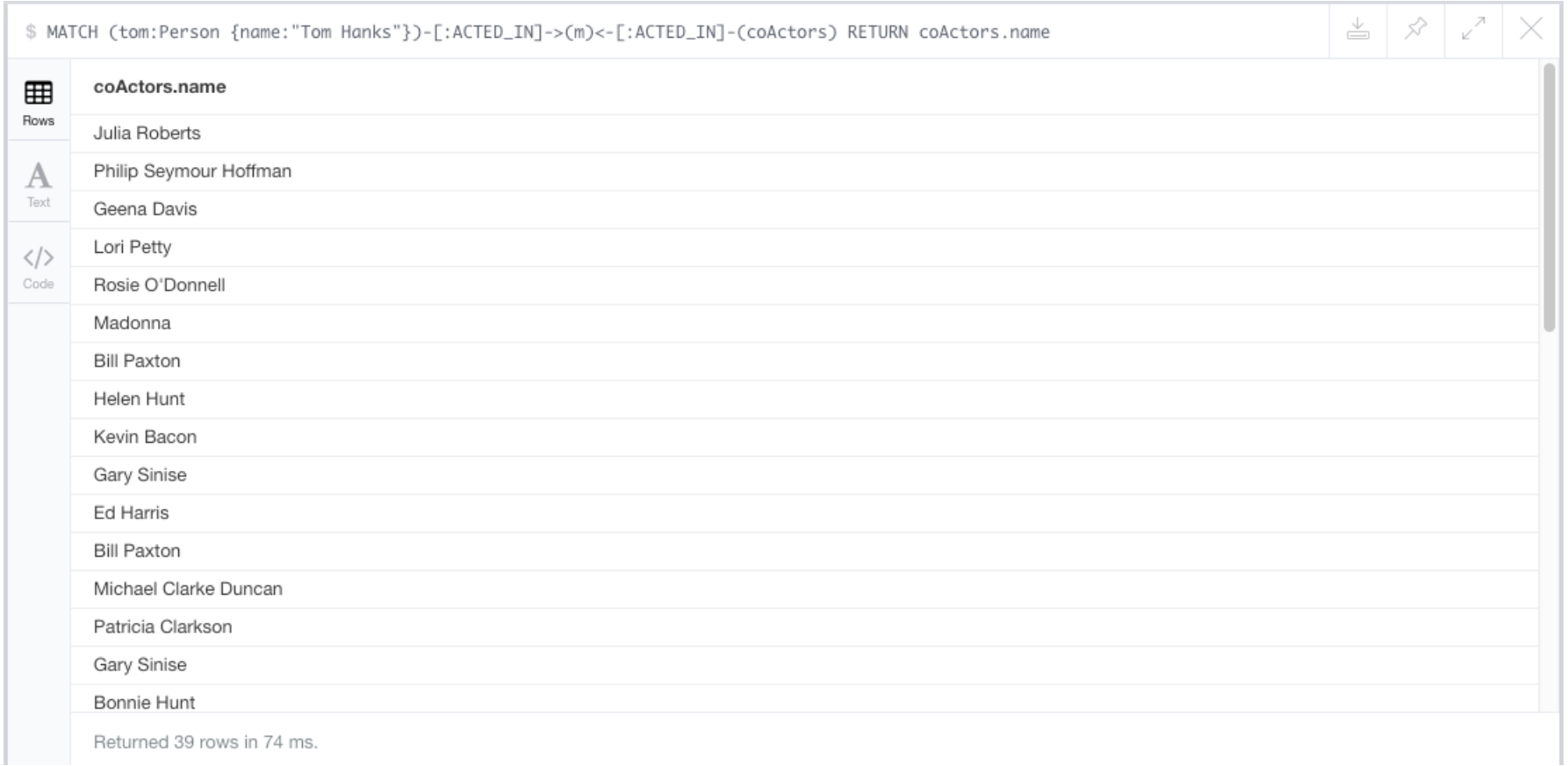
Cypher DDL

```
CREATE
  (TomC)-[:ACTED_IN {roles:['Lt. Daniel Kaffee']}]->(AFewGoodMen),
  (JackN)-[:ACTED_IN {roles:['Col. Nathan R. Jessup']}]->(AFewGoodMen),
  (DemiM)-[:ACTED_IN {roles:['Lt. Cdr. JoAnne Galloway']}]->(AFewGoodMen),
  (KevinB)-[:ACTED_IN {roles:['Capt. Jack Ross']}]->(AFewGoodMen),
  (KieferS)-[:ACTED_IN {roles:['Lt. Jonathan Kendrick']}]->(AFewGoodMen),
  (NoahW)-[:ACTED_IN {roles:['Cpl. Jeffrey Barnes']}]->(AFewGoodMen),
  (CubaG)-[:ACTED_IN {roles:['Cpl. Carl Hammaker']}]->(AFewGoodMen),
  (KevinP)-[:ACTED_IN {roles:['Lt. Sam Weinberg']}]->(AFewGoodMen),
  (JTW)-[:ACTED_IN {roles:['Lt. Col. Matthew Andrew Markinson']}]->(AFewGoodMen),
  (JamesM)-[:ACTED_IN {roles:['Pfc. Louden Downey']}]->(AFewGoodMen),
  (ChristopherG)-[:ACTED_IN {roles:['Dr. Stone']}]->(AFewGoodMen),
  (AaronS)-[:ACTED_IN {roles:['Man in Bar']}]->(AFewGoodMen),
  (RobR)-[:DIRECTED]->(AFewGoodMen),
  (AaronS)-[:WROTE]->(AFewGoodMen)
```

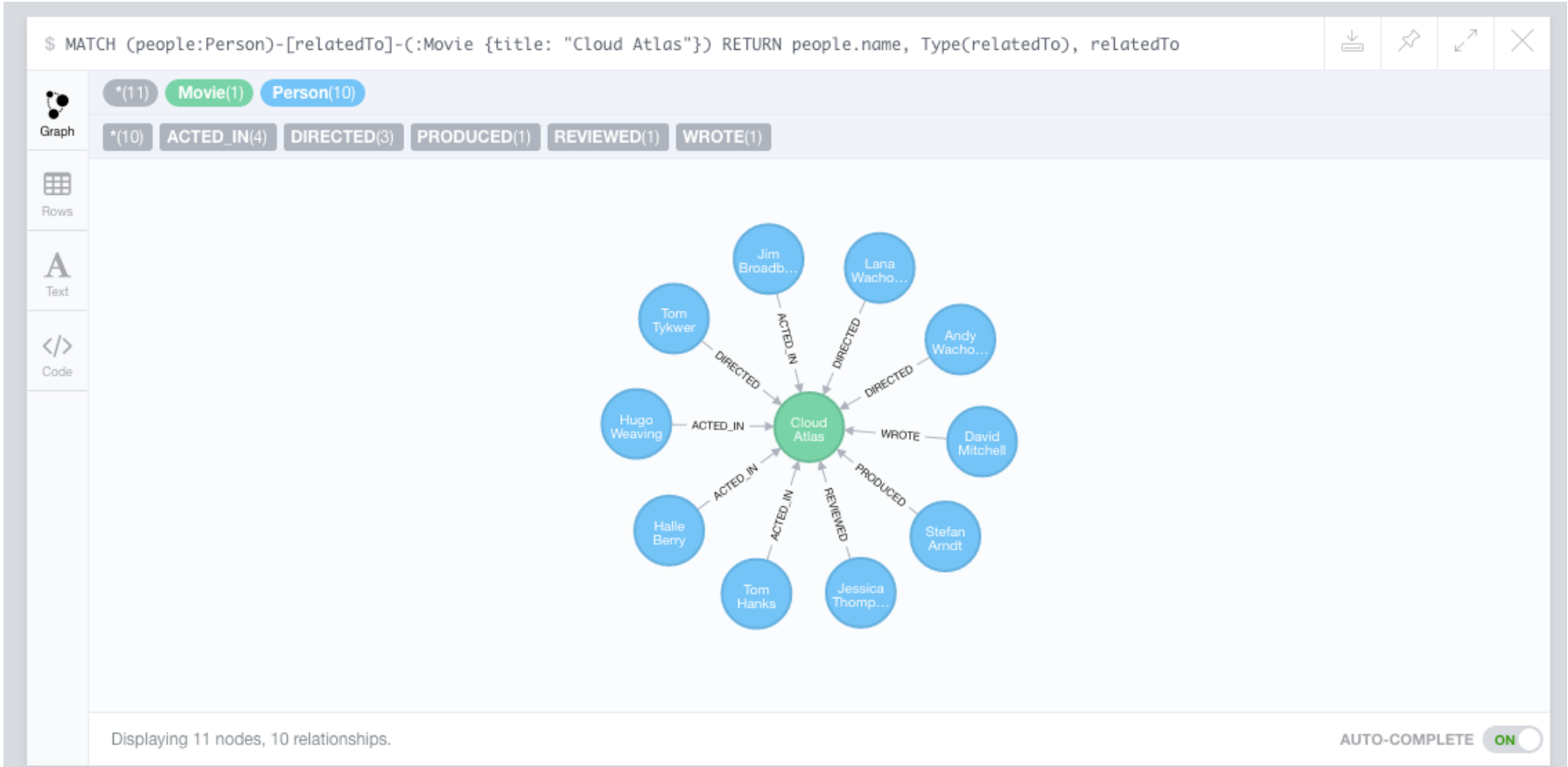
Cypher DML

Cypher DML

Cypher DML



Cypher DML



Some Spooky Stuff

The Movie Graph

Solve

You've heard of the classic "Six Degrees of Kevin Bacon"? That is simply a shortest path query called the "Bacon Path".

1. Variable length patterns
2. Built-in shortestPath() algorithm

Movies and actors up to 4 "hops" away from Kevin Bacon

```
MATCH (bacon:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)
RETURN DISTINCT hollywood
```

Bacon path, the shortest path of any relationships to Meg Ryan

```
MATCH p=shortestPath(
  (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
)
RETURN p
```

The Movie Graph

Recommend

Let's recommend new co-actors for Tom Hanks. A basic recommendation approach is to find connections past an immediate neighborhood which are themselves well connected.

For Tom Hanks, that means:

1. Find actors that Tom Hanks hasn't yet worked with, but his co-actors have.
2. Find someone who can introduce Tom to his potential co-actor.

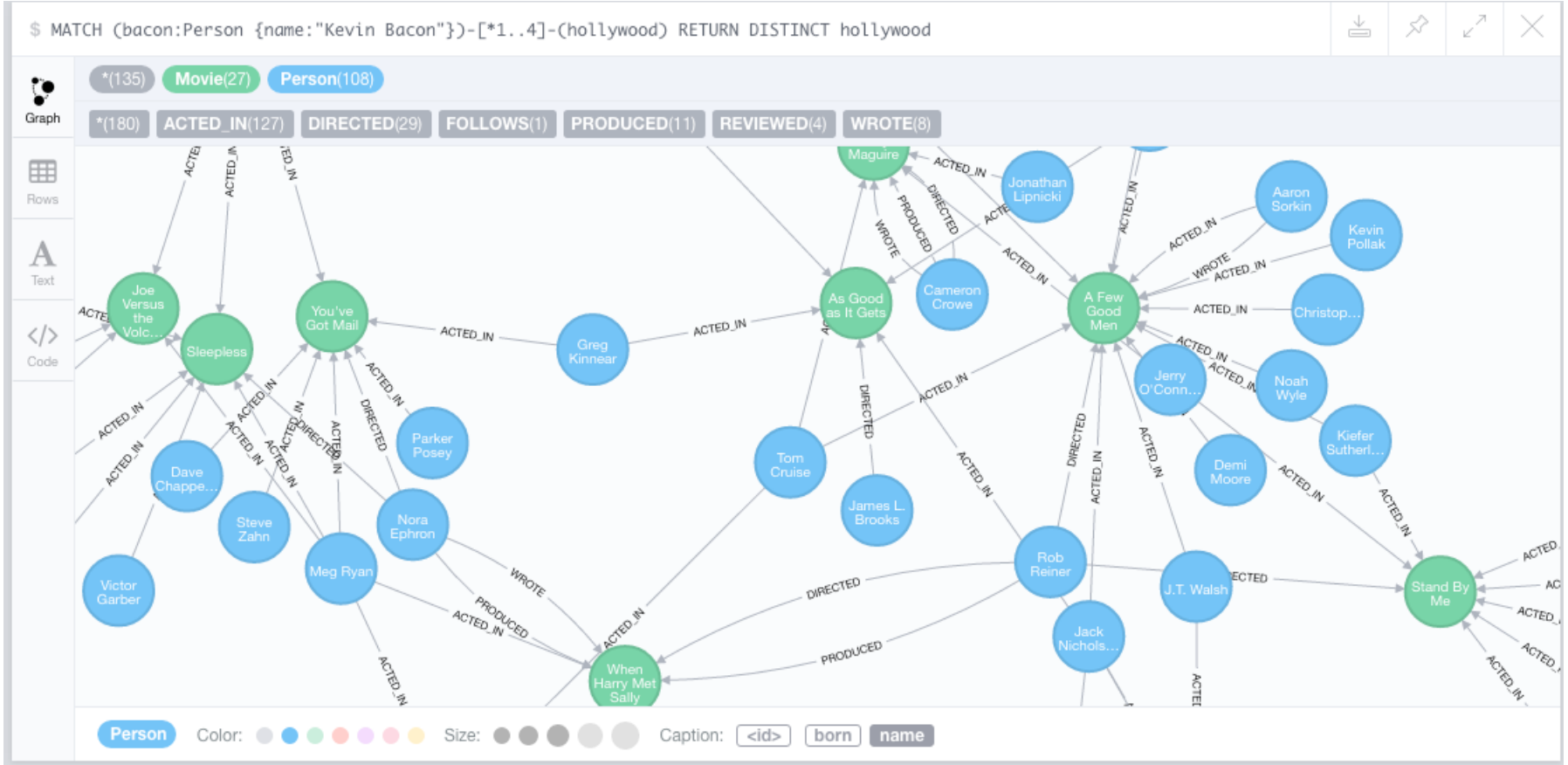
Extend Tom Hanks co-actors, to find co-co-actors who haven't work with Tom Hanks...

```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cocoActors)
WHERE NOT (tom)-[:ACTED_IN]->(m2)
RETURN cocoActors.name AS Recommended, count(*) AS Strength ORDER BY Strength DESC
```

Find someone to introduce Tom Hanks to Tom Cruise

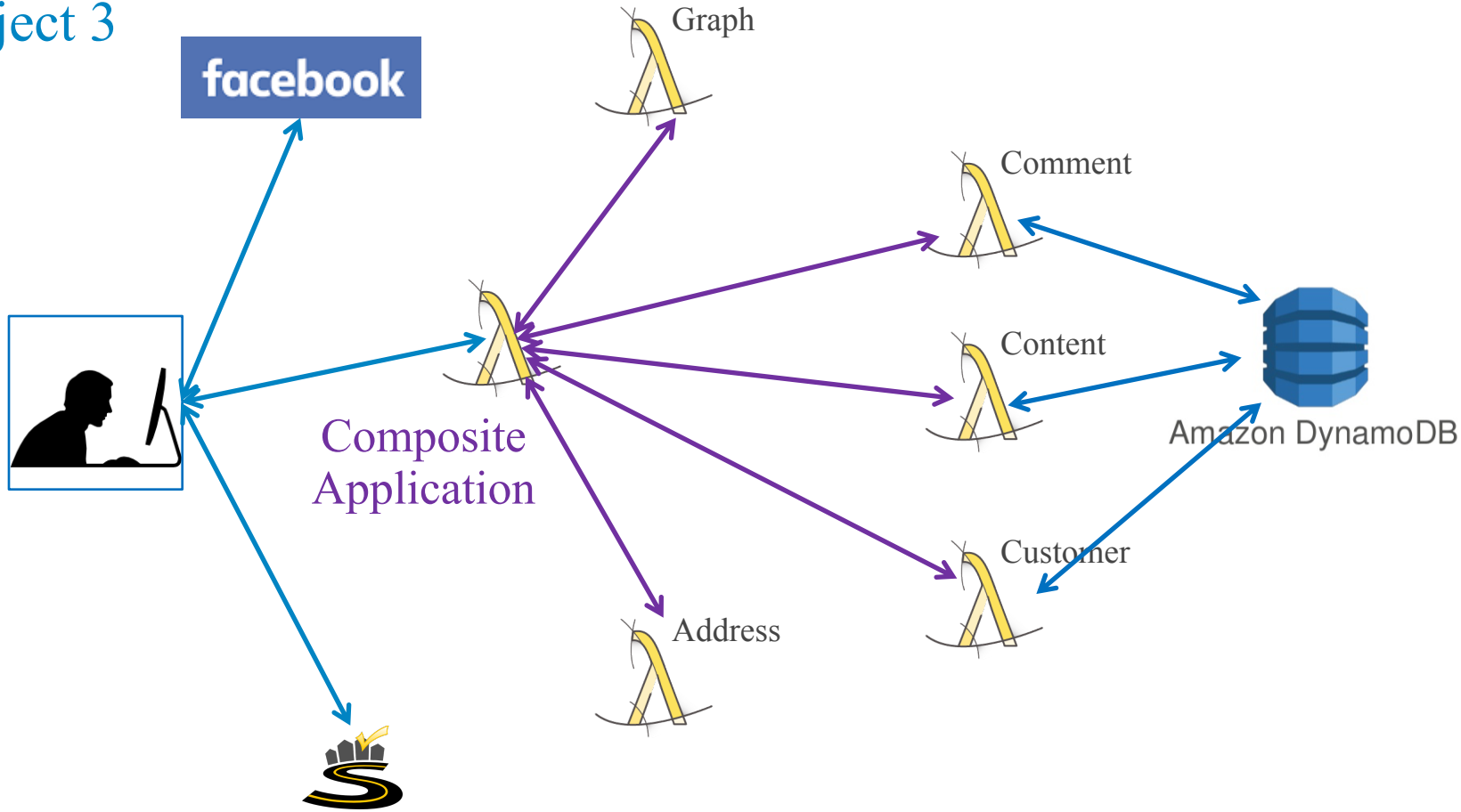
```
MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-(coActors),
      (coActors)-[:ACTED_IN]->(m2)<-[:ACTED_IN]-(cruise:Person {name:"Tom Cruise"})
RETURN tom, m, coActors, m2, cruise
```

Some Spooky Stuff



3rd Project

Project 3



3rd Project (Part 1)

- Define and provide a Lambda/REST interface to a "content catalog" in DynamoDB.
  - Model: Content Instance
    - Property (e.g. HBO, CBS) is a set of one or more Franchises.
    - A Franchise (e.g. Game of Thrones, Bing Bang Theory) is a set of one or more Series.
    - A Series (e.g. Season 1, Christmas Specials) is a set of one or more Episodes.
    - An Episode is an instance of a "TV show."
    - Data for each type is just (ID, name).
  - Comment (ID, user, content instance, comment)
- Graph database
  - Signup for/begin to use REST callable Neo4J database (e.g. graphenedb.com)
  - Node type are Person, Content (one of Property, Franchise, …) and Comment.
  - Relationships between people (Follows, Friends)
  - Relationships between people and comments (commented, responded, liked, ...)
  - Relationships between people and content (Likes, Rates, ...)
- Expand REST API and Lambda functions to provide integrated API for accessing data and relationships
  - Customer/person, Comment, Content Instance, ... in DynamoDB including property links.
  - CRUD on Customer, Content Instance, Comment create nodes and links in DynamoDB.
- Login and Register with Facebook
  - Register creates name, email, etc. From Facebook data.
  - User can change address, name, preferred email, etc.

3rd Project (Part 1)

- Define and pro… Lambda/REST interfaces “…” in Dynamo DB
  - Model
    - …
    - …
    - …
    - …
    - …
  - Con…
- Graph da…
  - Sign…
  - Nod…
  - Rela…
  - Rela…
  - Rela…
- Expand l… …ips
  - Cus…
  - CRU…
- Login and Re…
  - Register creates name, email, etc. From Facebook data.
  - User can change address, name, preferred email, etc.

- Welcome to the “Big Leagues,”or H*ll, depending on your point-of-view.
- This project is going to “force you” to
  - Think in terms of user stories, scenarios, …
  - Do some basic high-level design diagrams.
  - Carefully think about API model
  - etc.
- Three weeks completion, with interim reviews.