E6156 : Topics in Software Engineering Cloud and Microservice Applications

Donald F. Ferguson, dff@cs.columbia.edu

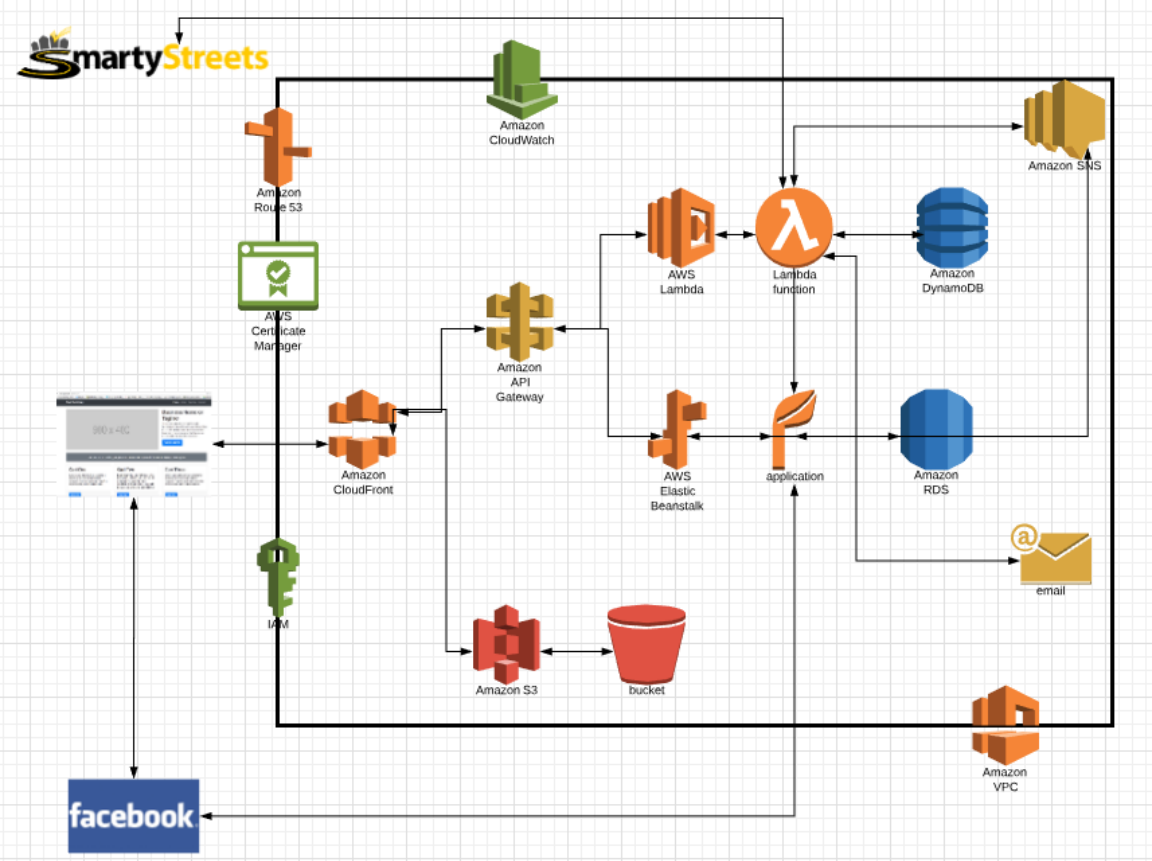
Contents

Contents

- Status
  - Reminder on overall project structure.
  - Semester roadmap.
- Email Verification Continued.
- OAuth2
  - Overview and concepts.
  - Facebook Login example.
- Composition and Orchestration
  - Composition concept.
  - Activiti example.
  - AWS Workflow, AWS Step Functions

Status

Overall Project Structure



E6156: Solution Structure, OAuth2, Facebook Login, Orchestration

From Lecture 1 – Objectives

Build a simple multi-tenant cloud application in a small team

- A set of microservices.
- Using elements of cloud computing:
  - PaaS
  - FaaS
  - Logging
  - Serverless
  - Cloud Database(s)
  - Security: Authentication, Authorization and Federation.
  - Message Queues, Events and Event-Driven-Architecture
  - Simple workflow and service orchestration.
  - Call cloud business APIs.
  - Email, Mobile.
  - Other cool stuff.

Status and Roadmap

- We covered an overview of many concepts, but are behind on applying to a very basic, functioning application.

- Lecture Roadmap:
  - 09-Nov: Email verification, Facebook, Composition/Orchestration Concepts.
  - 16-Nov: Composition/Orchestration, Messaging – Details, Impl.
  - 23-Nov: Thanksgiving.
  - 30-Nov: Graph Database, other cloud databases.
  - 07-Nov: Misc. topics.

- Final project reviews: 10-Dec to 17-Dec.

OAuth2

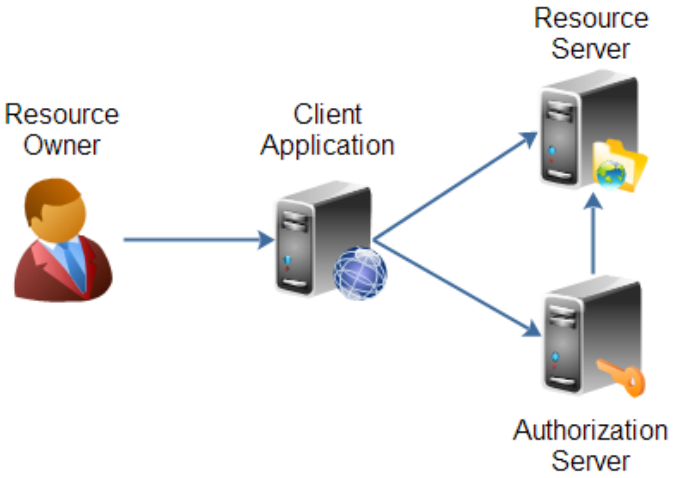
Overview (http://tutorials.jenkov.com/oauth2/index.html)

- 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 Serv
- Authorization Server
  - Grants/rejects authorization
  - Based on Resource Owner decisions.
  - Usually (logically) the same as Resource Ser

OAuth 2.0 defines the following roles of users and applications:

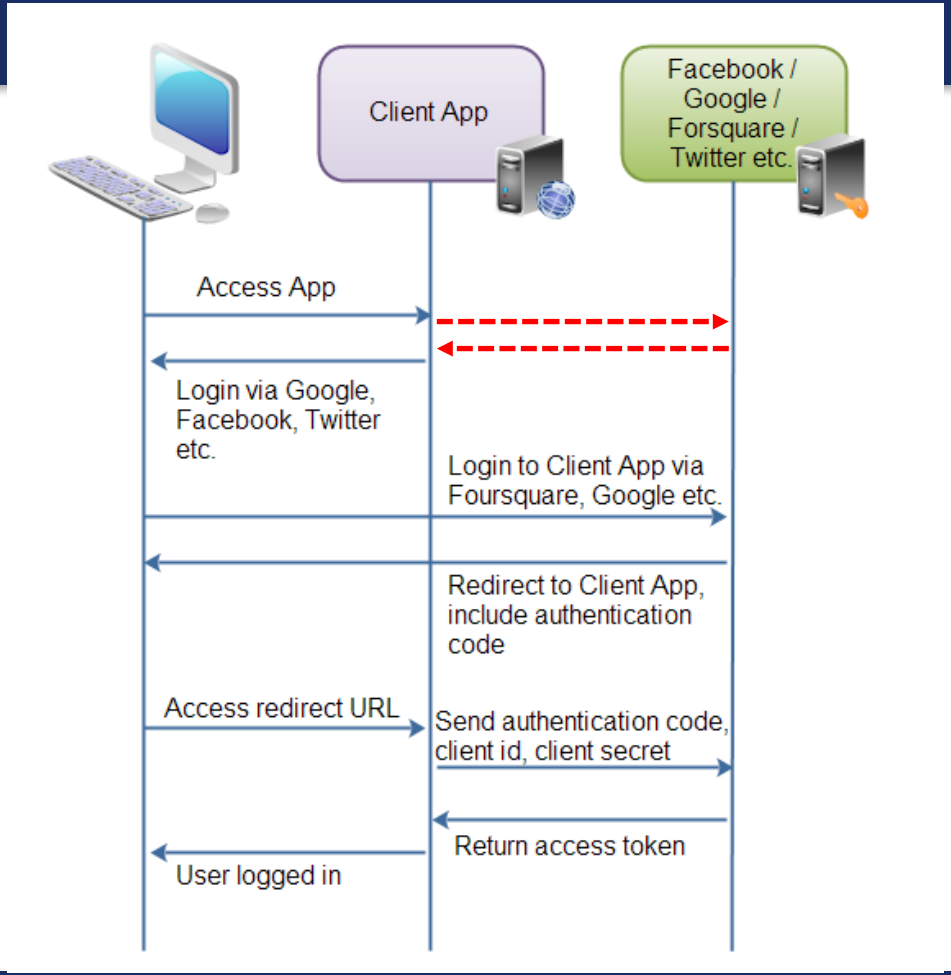
- Resource Owner
- Resource Server
- Client Application
- Authorization Server

These roles are illustrated in this diagram:

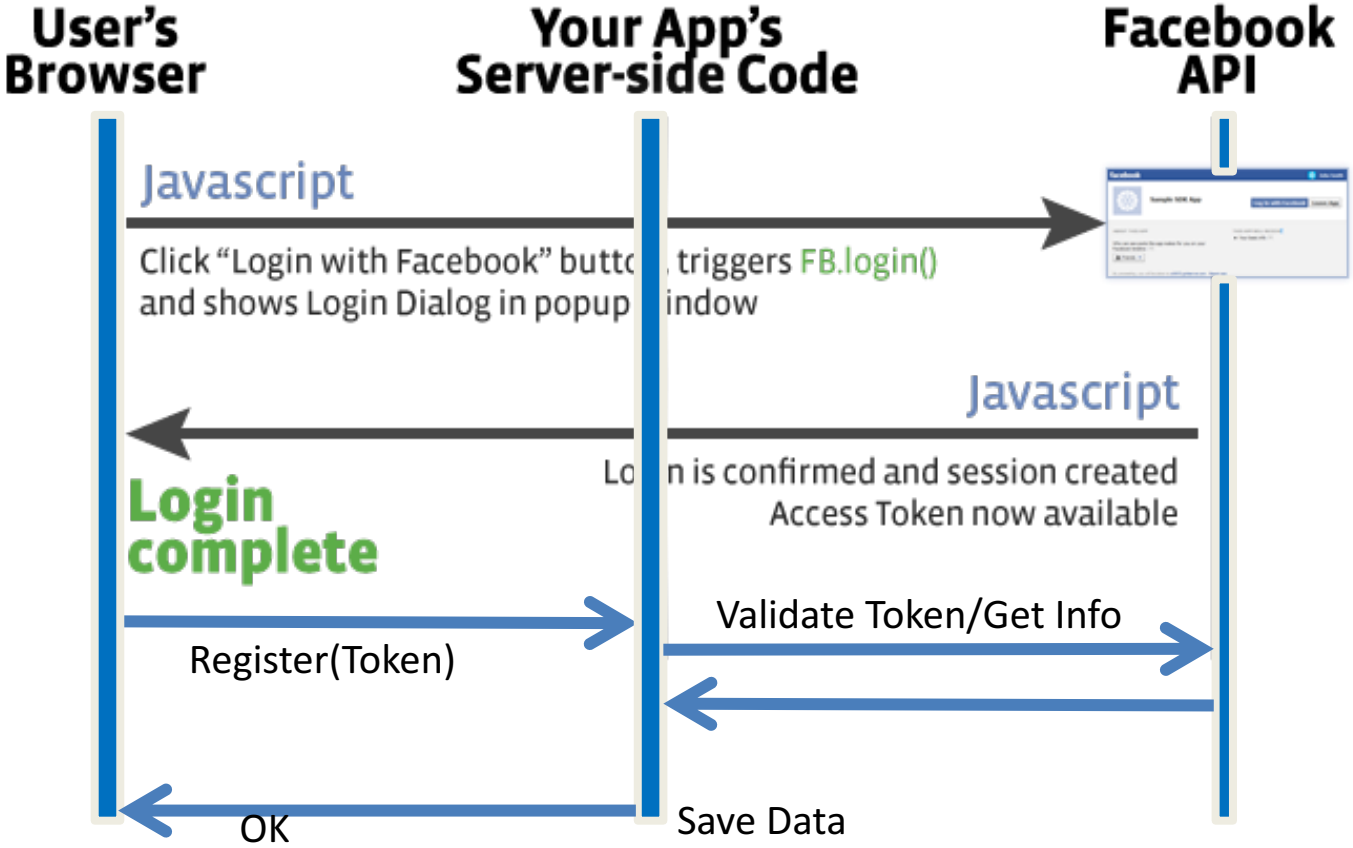


Roles/Flows

- 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.

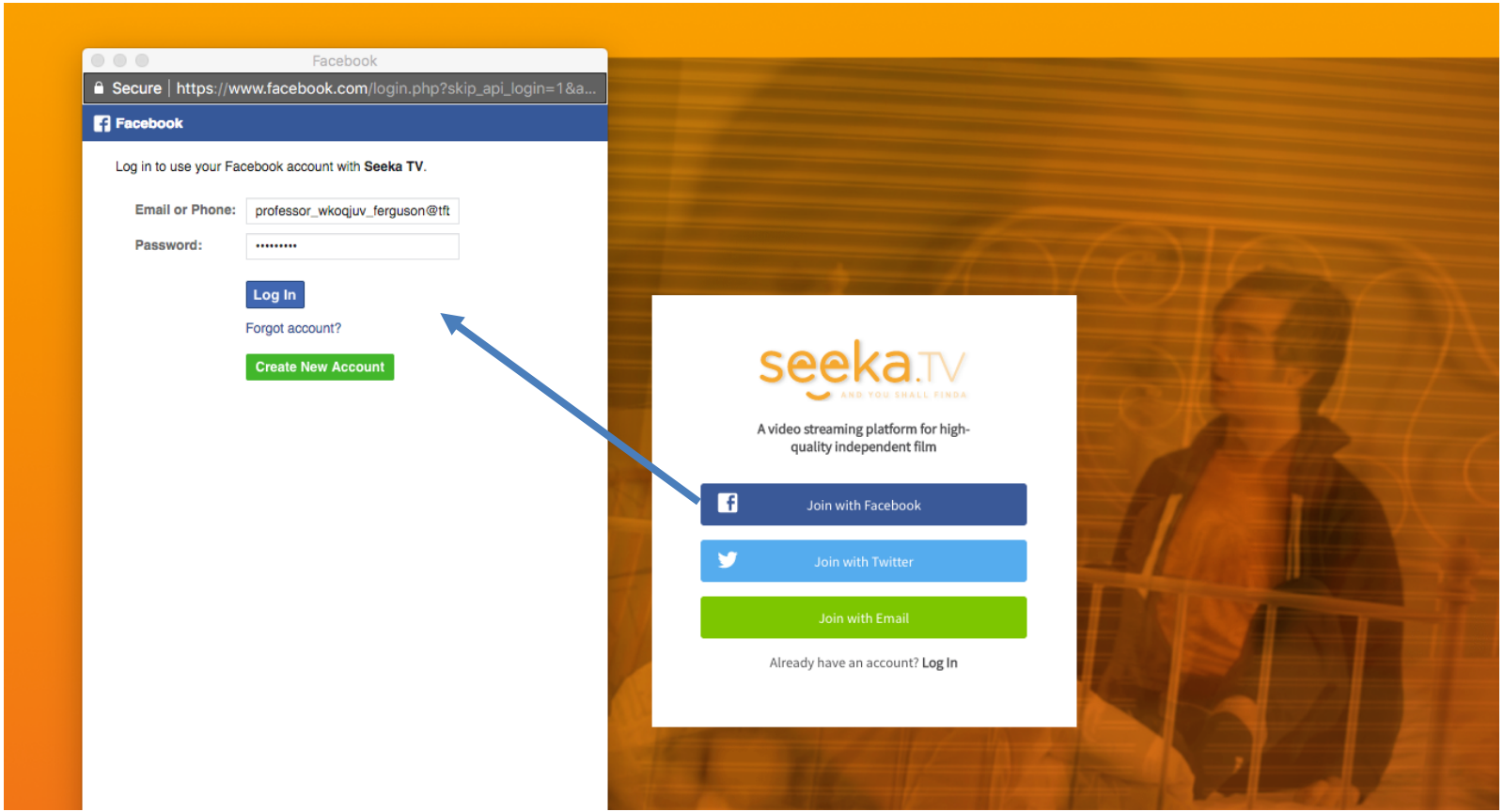


Facebook

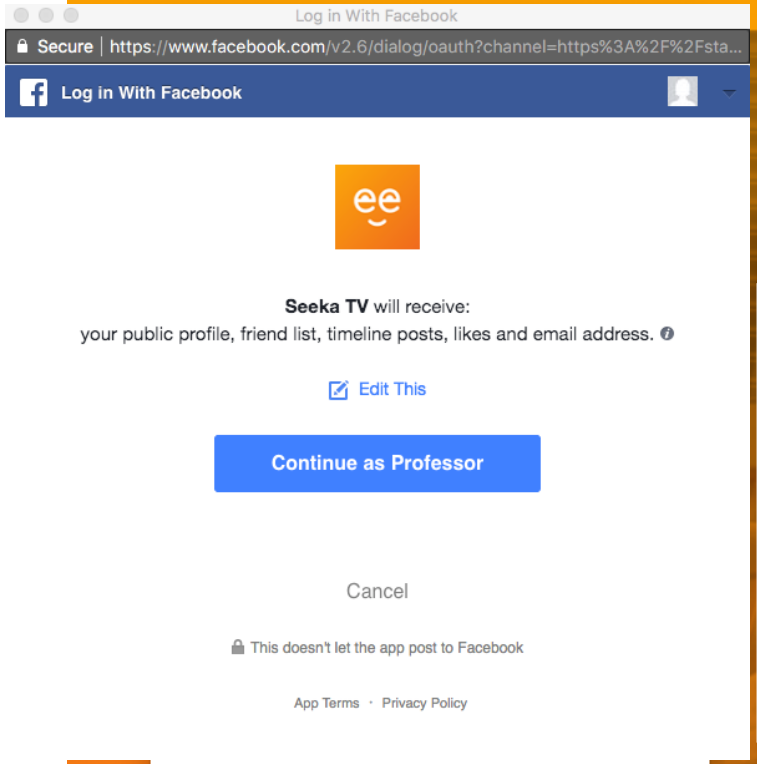


Registration

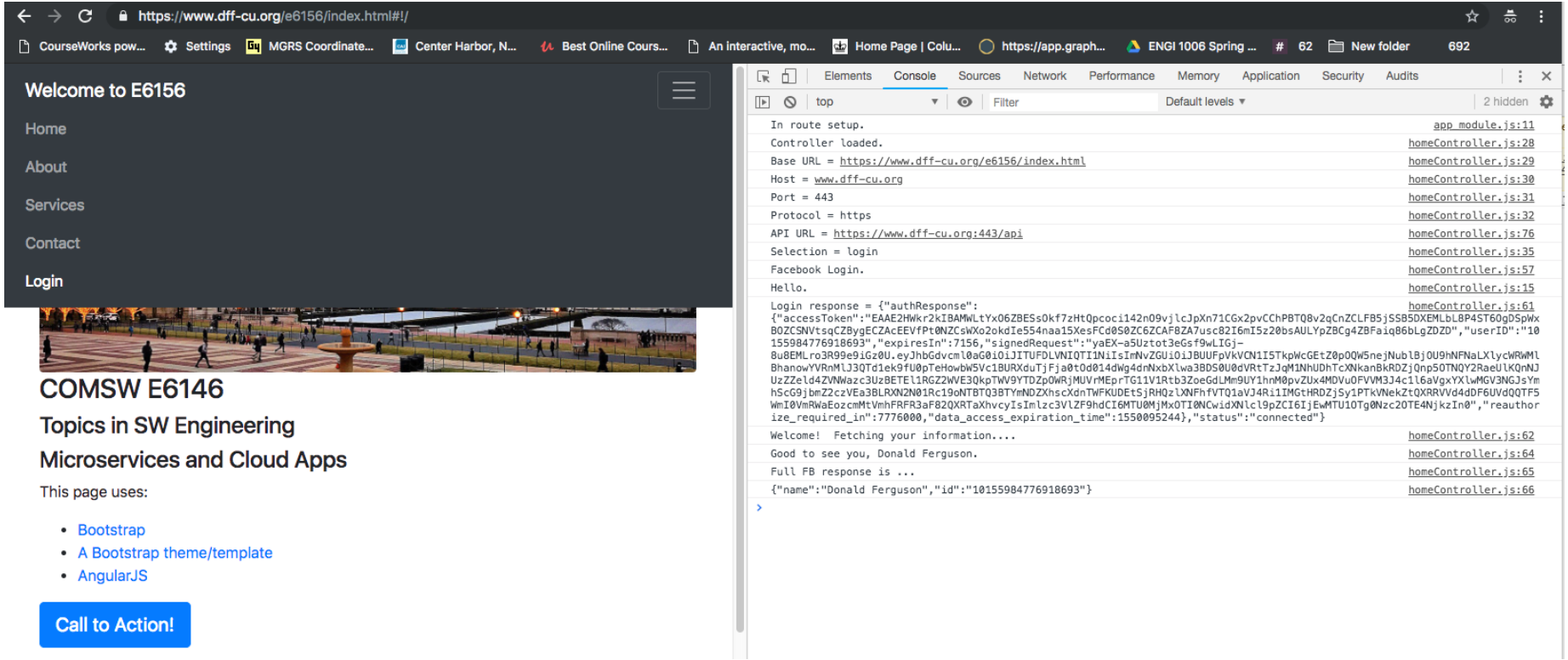
Registration



Registration



E6156 Client Demo



Facebook Login Controller Code

```
$scope.doFacebook = function() {
    console.log("Facebook Login.")
    FB.login(function(response) {
        if (response.authResponse) {
            $("#loginModal").modal("hide");
            console.log("Login response = " + JSON.stringify(response))
            console.log('Welcome!  Fetching your information.... ');
            FB.api('/me', function(response) {
                console.log('Good to see you, ' + response.name + '.');
                console.log("Full FB response is ...")
                console.log(JSON.stringify(response))
            })
        } else {
            console.log('User cancelled login or did not fully authorize.');
        }
    });
};
```

I would call my Beanstalk application to obtain a token from my application, just like for user ID/PW login.

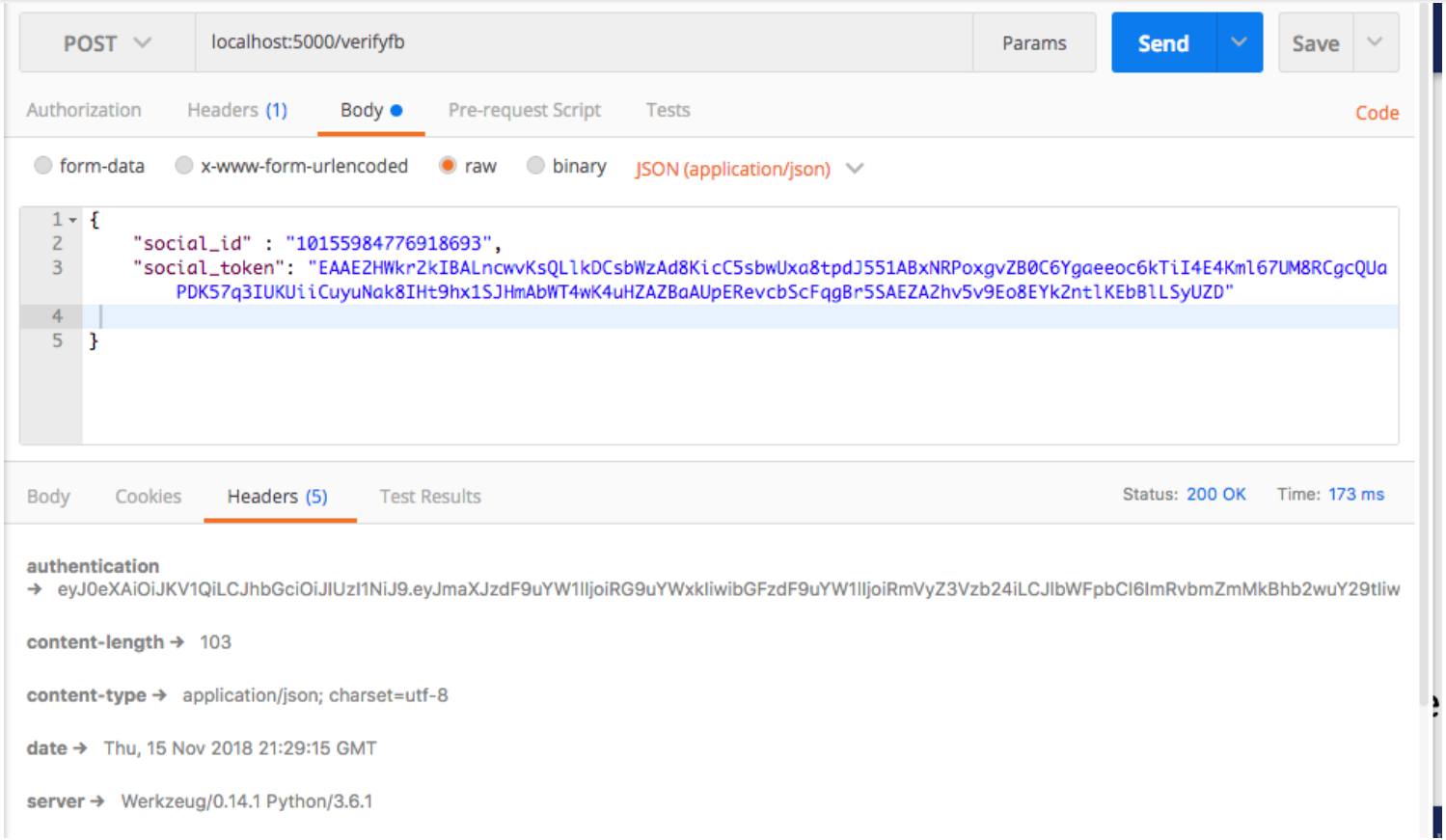
Sample Backend Code

```python
import facebook


def validate(social_media_id, token):
    rsp = None
    try:
        graph = facebook.GraphAPI(access_token=token)
        rsp = graph.get_object(id=social_media_id, fields="first_name,last_name,email")
        print("FB said ...", rsp)
    except Exception as e:
        print("FB error = ", e)
        rsp = None

    return rsp
```

Continued

```
app = Flask(__name__)


def encode_token(info):
    try:
        result = jwt.encode(info, secret)
    except Exception as e:
        print("Error decoding info = ", info)
        print("Exception e = ", e)

    return result


@app.route('/verifyfb', methods=['POST'])
def verify():
    if request.data:
        try:
            dd = json.loads(request.data)
            print("Data = ", json.dumps(dd))
            result = facebooksvc.validate(dd['social_id'], dd["social_token"])
        except Exception as e:
            print("Exception, e = ", e)
            result = None

        if result is not None:
            token = encode_token(result)
            result = json.dumps(result)
            return result, 200, {'Content-Type': 'application/json; charset=utf-8',
                                 'authentication': token}
        else:
            return "Internal error.", 500, {'Content-Type': 'application/text; charset=utf-8'}
    else:
        return "No body.", 500, {'Content-Type': 'application/text; charset=utf-8'}
```

Sample Token Exchange Response



Facebook/Social Media Login

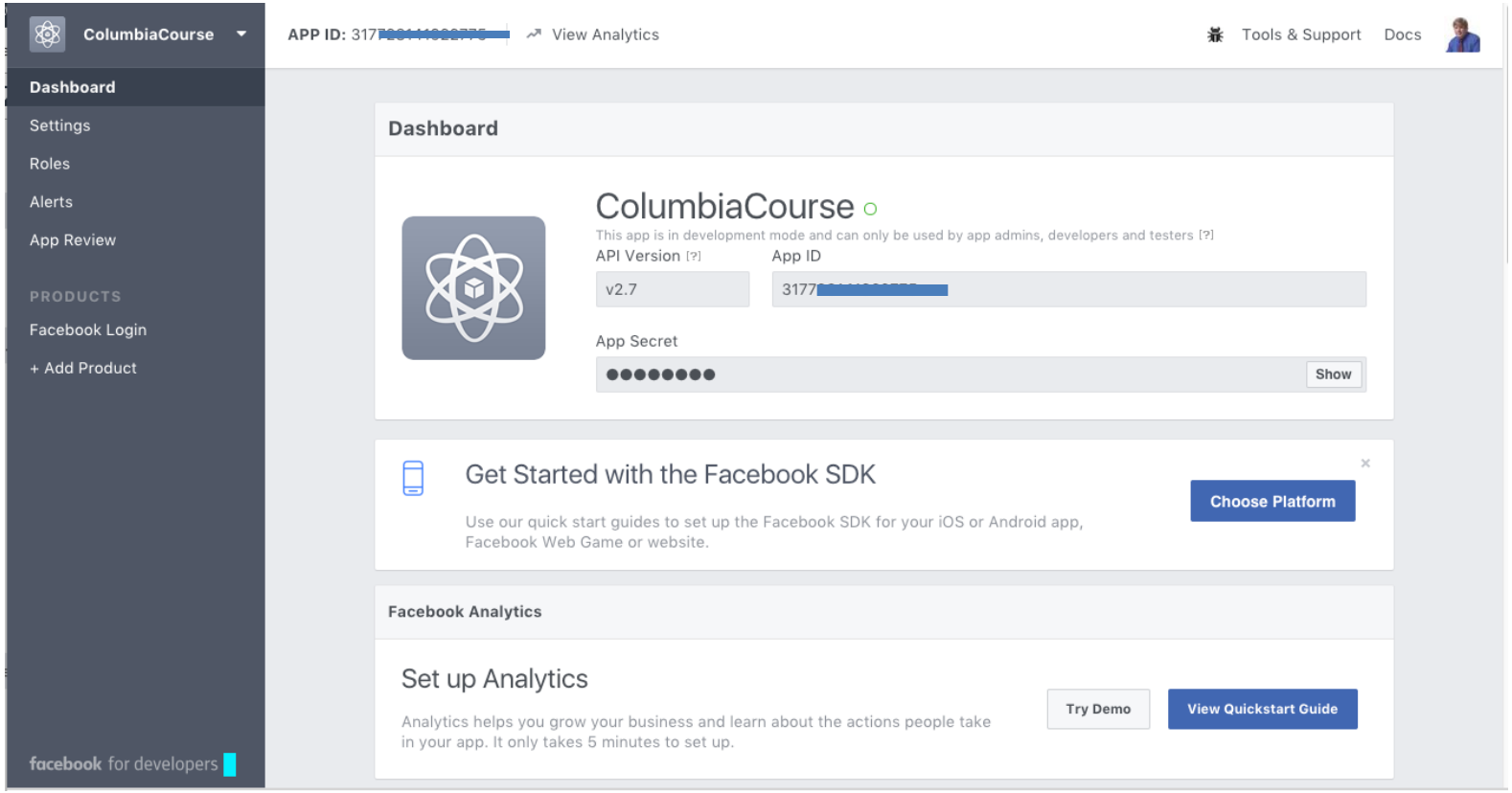
- My application does not maintain security credentials.
- Facebook is the authentication provider.
- For email and passsword, my database maintains
  - User ID, email
  - Password hash
- For social media, my database:
  - Just maintains the social provider, social ID for the user ID.
  - Relies on the social media provider to authenticate the user.
- In either case,
  - I can use the ID for profile and other keys into my data model.
  - Still generate an access token for my environment to grant rights, store session information, etc.

Pros and Cons of Social Media Logon

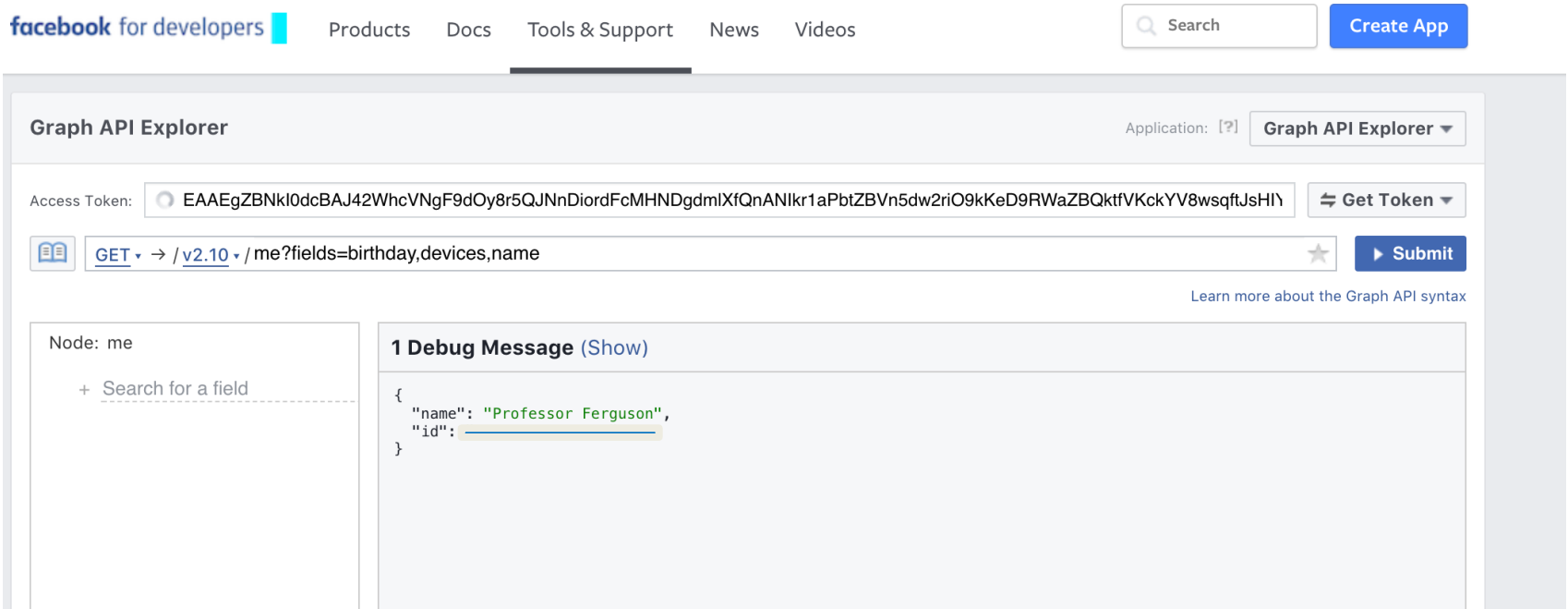
- Pros
  - One password for many services: Some people hate making up passwords
  - Familiarity: People trust Facebook and may not want to give you info.
  - Added convenience drives business: “Forced registration is a major cause of checkout abandonment, with a quarter (26%) of respondents in a recent Econsultancy survey stating that being forced to register would cause them to abandon a purchase.”
  - Sharing: Simplifies customers sharing their use of your site → drives awareness.
  - Access to profile data: Do not require users to enter a new profile.
  - Your site does not have to have code for password reset, locked account, etc.
- Cons
  - Some users don’t want everything to be connected.
  - Some people don’t use Facebook.
  - Muddying your brand image.

(https://econsultancy.com/blog/61911-the-pros-and-cons-of-a-facebook-login-on-ecommerce-sites)

Define Application

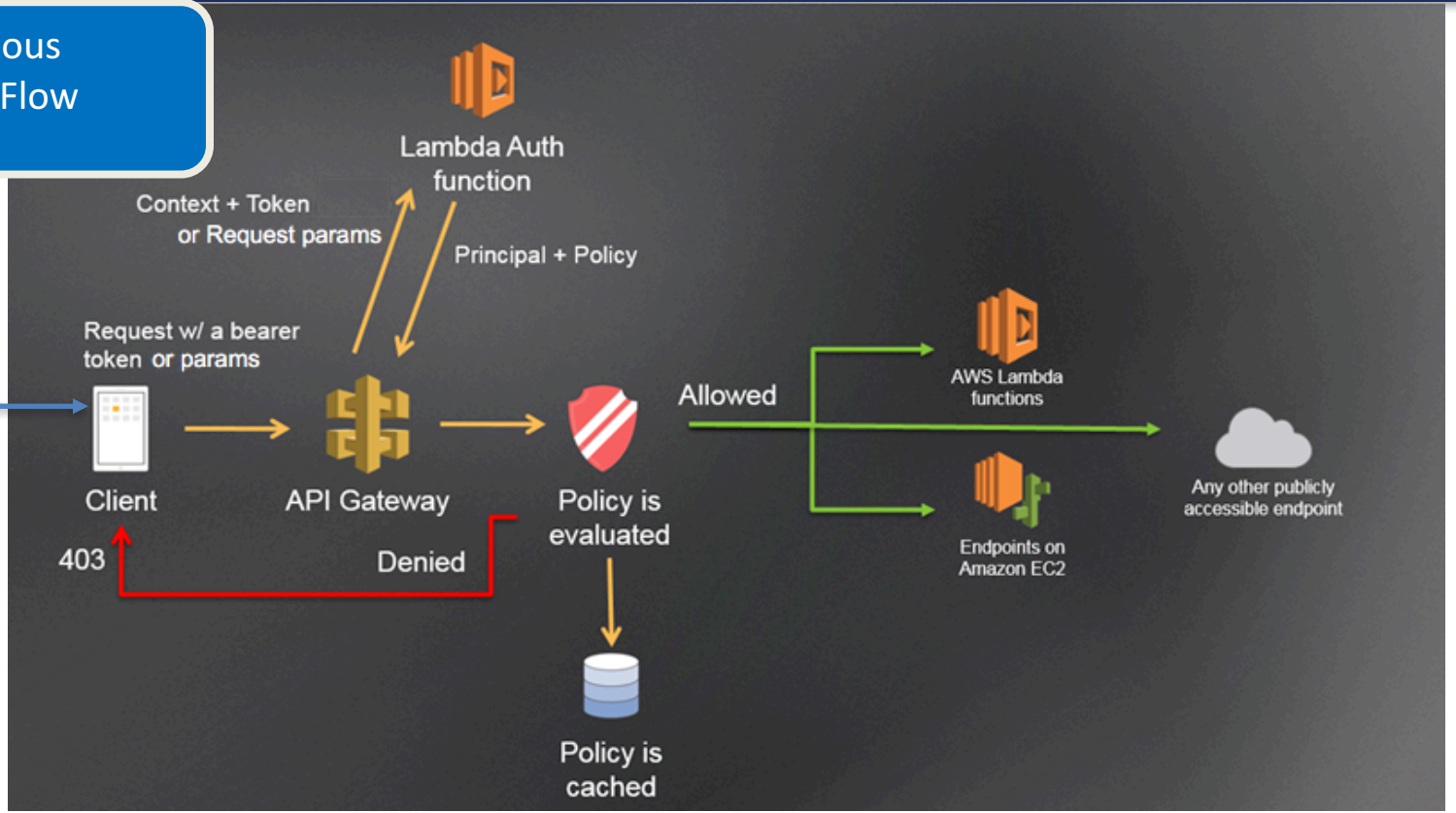


Graph API Explorer Demo



API Gateway Custom Authorizer

Previous
Login Flow



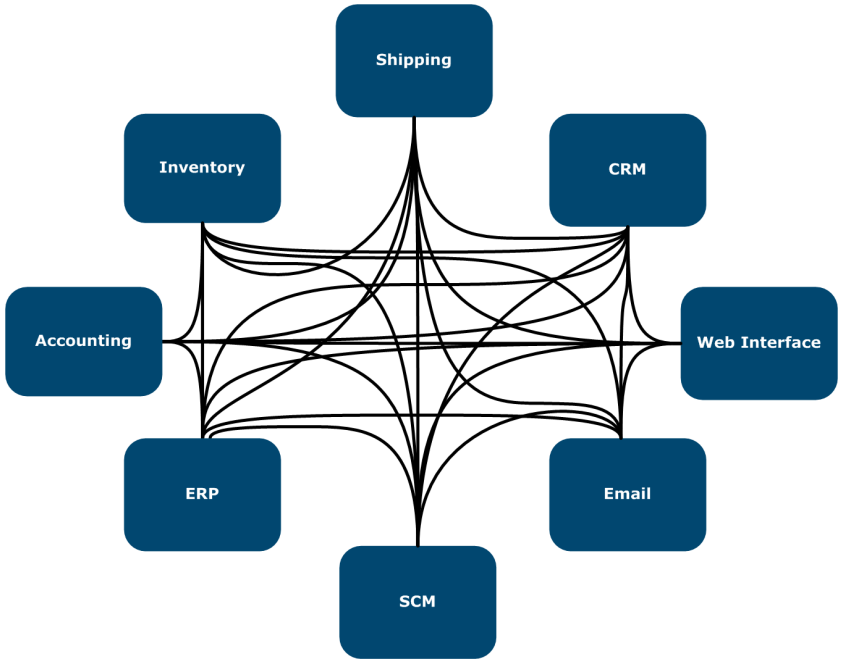
Authorization

- 1st check occurs at perimeter in API GW Authorization
  - Based on (bearer token) JWT token
  - Is this role authorized to invoke this URL?
  - Implementation
    - Lambda function that …
    - Checks an authorization table (role, URL)
    - Or the rights are encoded in the token.
- All down stream microservices
  - Receive the JWT token and perform their own authorization decisions.
  - Run with a set of AWS roles and permissions that determine if AWS allows access, which may include user role.
- We may have time to implement this scenario this semester.

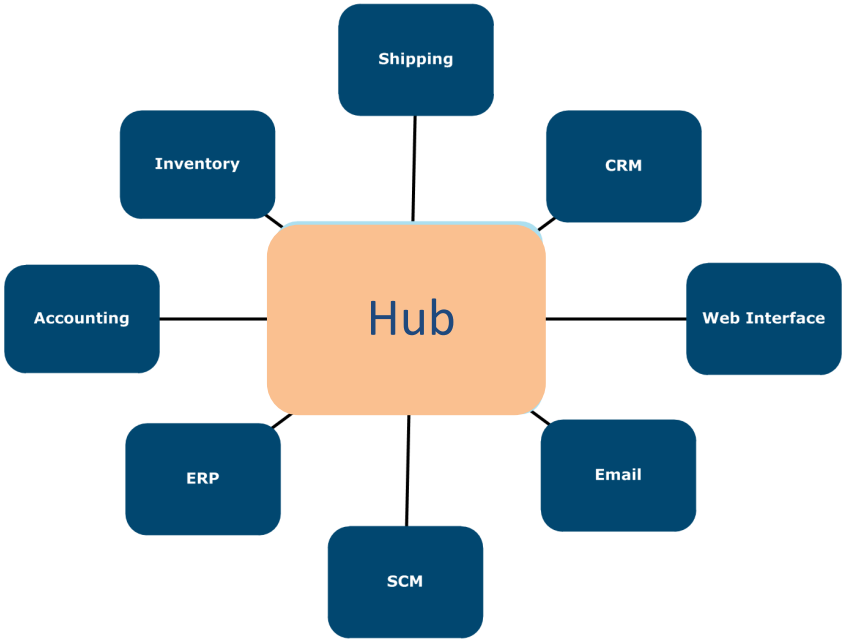
Orchestration -- Introduction

Serverless Functions and Micro-Service Integration *Patterns*

Bad, Bad, Bad, AAAARGH!



Pub/Sub or Message Hub

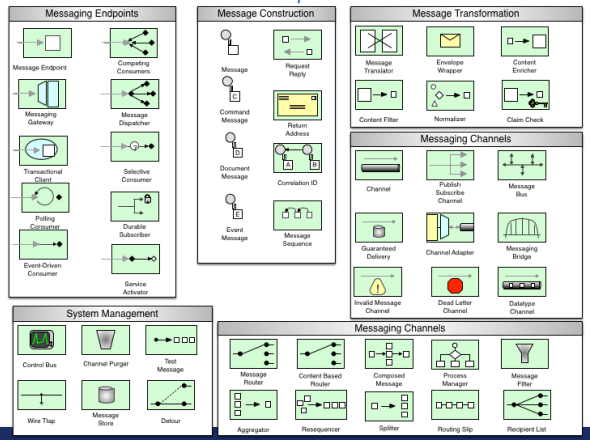


Integration/Composition Patterns

iPaaS and High-Level Abstractions

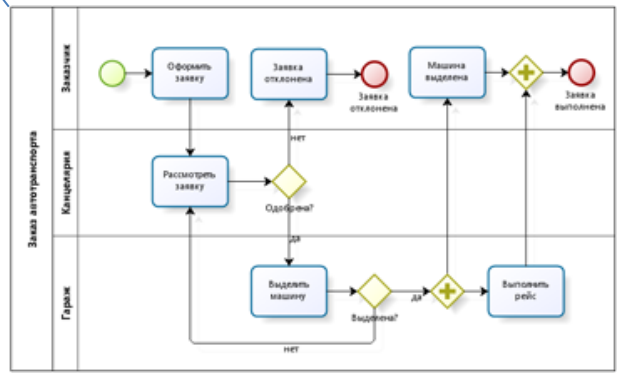
"Code"

Hub



"Patterns"

Orchestration/BPMN

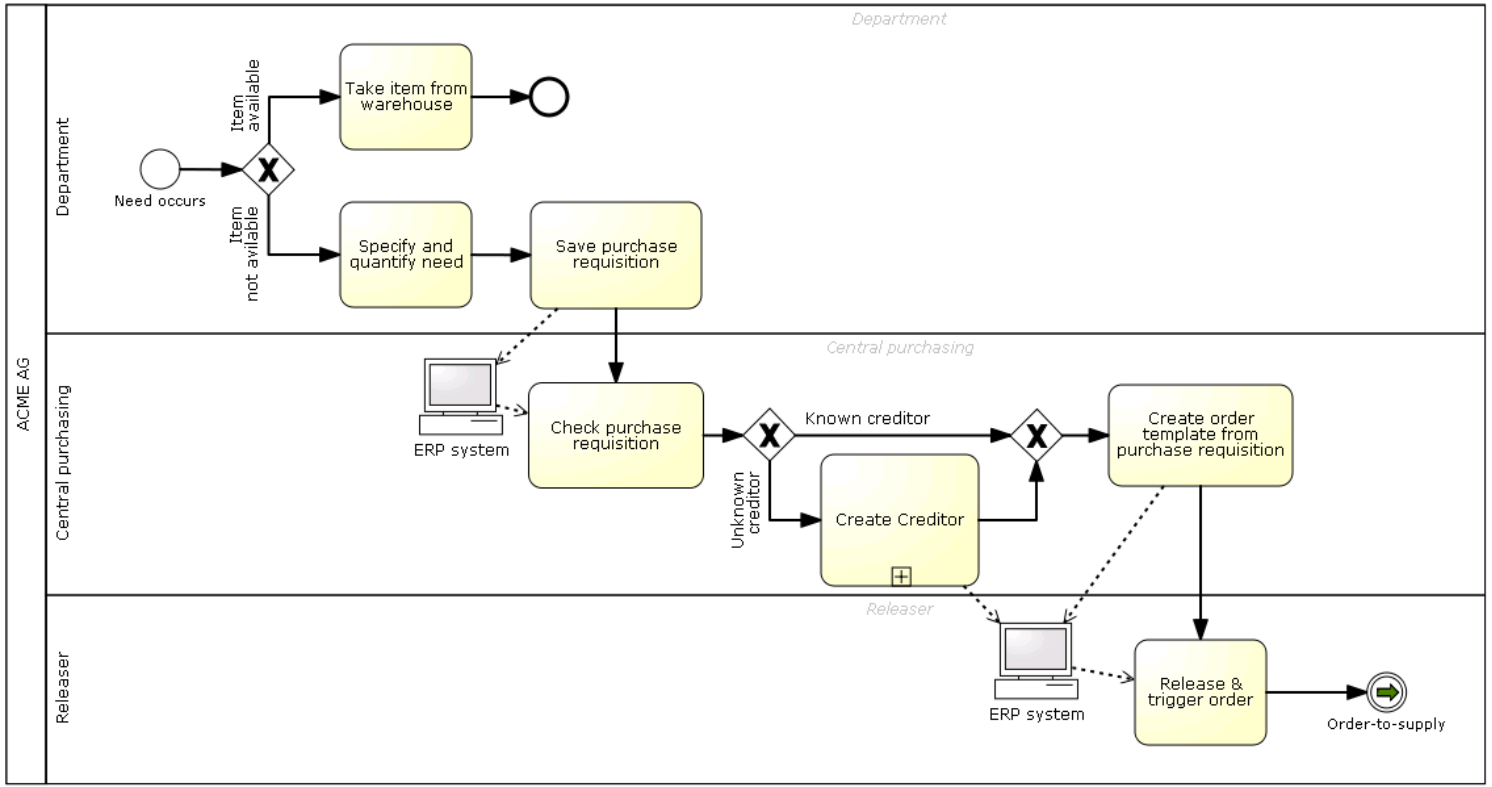


Workflow/Orchestration

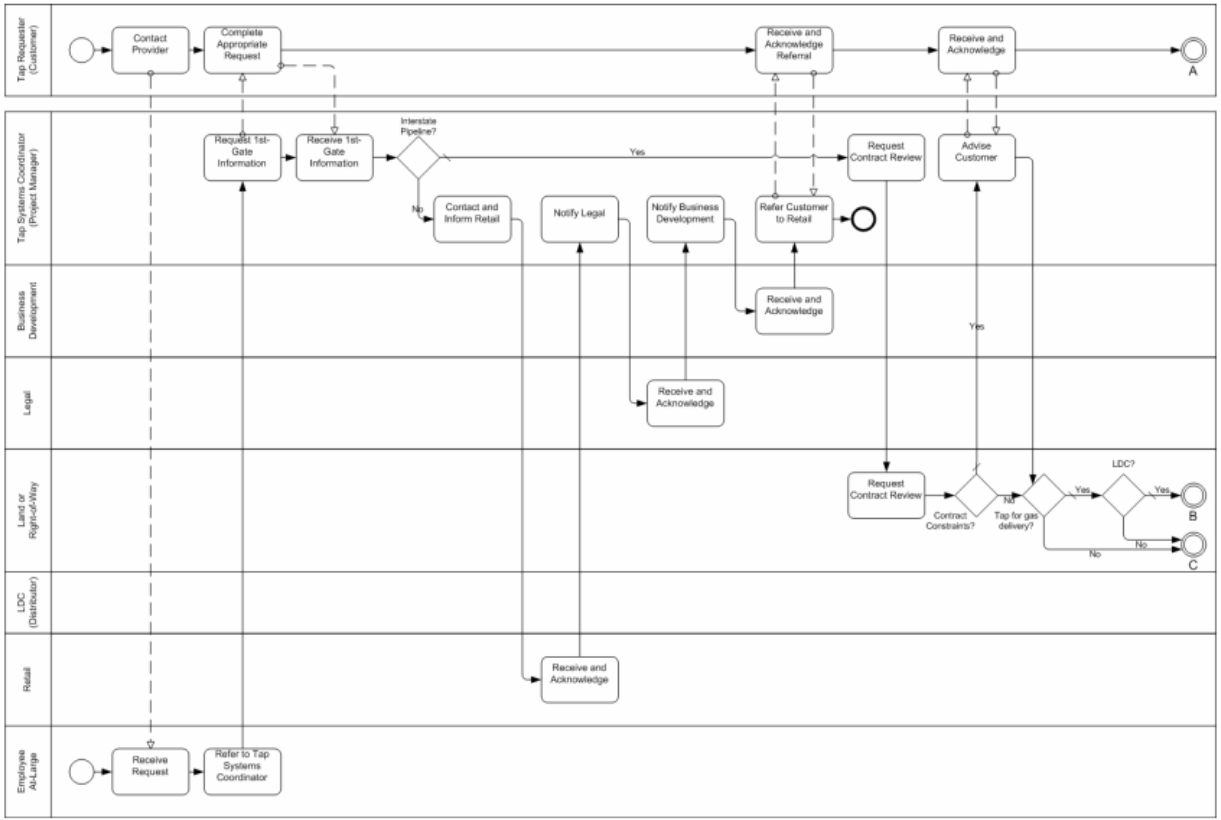
"A **workflow** consists of an orchestrated and repeatable pattern of business activity enabled by the systematic organization of resources into processes that transform materials, provide services, or process information.[1] It can be depicted as a sequence of operations, declared as work of a person or group,[2] an organization of staff, or one or more simple or complex mechanisms." (https://en.wikipedia.org/wiki/Workflow)

"Cloud computing introduces more-granular and specific meanings of the terms "workflows" and "processes" as used in different domains. Processes can be identified at both the level of inter-company business and the level of wide area network ("the cloud") operations. An "orchestrator" is understood to be the entity which manages complex cross-domain (system, enterprise, firewall) processes and handles exceptions. Since an orchestrator is valuable in fulfillment, assurance, and billing processes,[4] service-aware incarnations of an orchestrator should be capable of adjustments based on feedback from monitoring tools. At the most basic level, an orchestrator is a human." (https://en.wikipedia.org/wiki/Orchestration_(computing))

Simple BPMN Diagram



And You can Define Processes

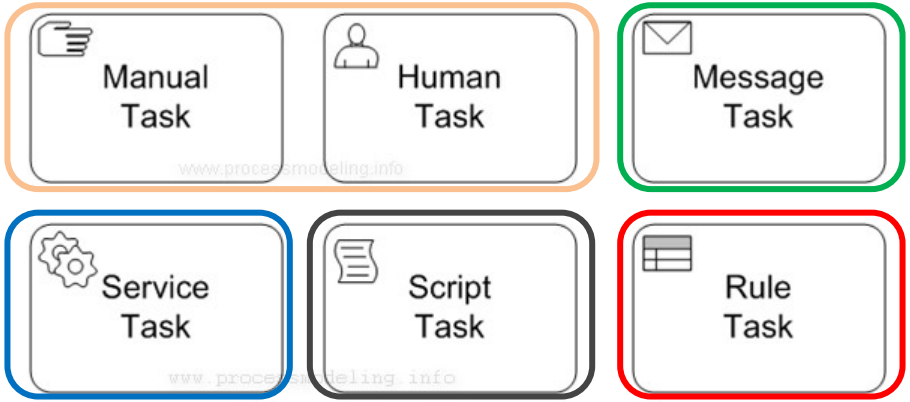


BPMN – How do you Implement?

Notify a person/role.

Have them do something.

Have the use an app that calls the engine to tell you they are done.



Send an email.

POST a document

etc.

Decision table

Call Drools

etc.

Insert into a DB

Call a web service

Run a Java app.

etc.

Run some JavaScript or Java or …

Right at this point in the workflow/process

And manipulate the data/documents/control.

Modeling/Pictures (an aside)

Modeling Benefits

Manage Complexity

Modeling is essential in complexity management. Modeling benefits include:

- Viewing systems from multiple perspectives
- Discovering causes and effects using model traceability
- Improving system understanding through visual analysis
- Discovering errors earlier and reducing system defects
- Exploring alternatives earlier in the system lifecycle
- Improving impact analysis, identifying potential consequences of a change, or estimating modifications to implement a change
- Simulating system solutions without code generation

Modeling/Pictures (an aside)

Preserve Knowledge and Corporate Memory

Modeling helps enterprises preserve knowledge and corporate memory by:

- Storing the corporate memory in a versioned repository
- Enabling quick and easy understanding of your systems within an organization by all members of your teams
- Assisting new team members in getting up to speed quickly

Reuse

Modeling helps you reuse parts of existing information and knowledge in your new projects, saving time and money.

Automate

Modeling facilitates automation including these examples:

- Automate generation of a real working system or part of a system from models
- Automate repeatable tasks by writing scripts
- Use thousands of shortcuts and features for getting the expected results in a single click

AWS Step Functions

Productivity:

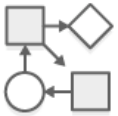
Build Applications Quickly

AWS Step Functions includes a visual console and blueprints for commonly-used workflows that make it easy to coordinate the components of distributed applications into parallel and/or sequential steps. You can build applications in a matter of minutes, and then visualize and track the execution of each step to help ensure the application is operating as intended.

Resilience:

Scale and Recover Reliably

AWS Step Functions automatically triggers each step so your application executes in order and as expected. It can handle millions of steps simultaneously to help ensure your application is available as demand increases. Step Functions tracks the state of each step and handles errors with built-in retry and fallback, whether the step takes seconds or months to complete.



Agility:

Evolve Applications Easily

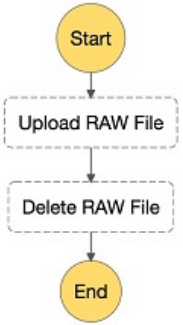
AWS Step Functions makes it easy to change workflows and edit the sequence of steps without revising the entire application. You can re-use components and steps without even changing their code to experiment and innovate faster. Your workflow can support thousands of individual components and steps, so you can freely build increasingly complex applications.

Concepts

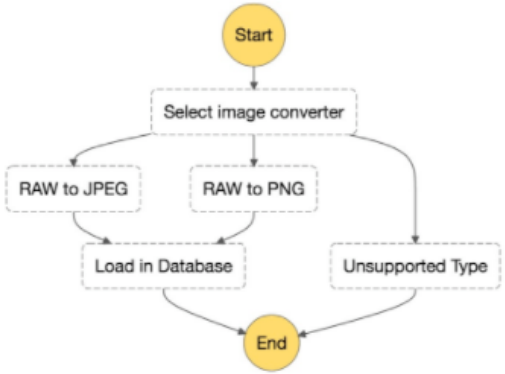
1. Define your Application Visually as a Series of Steps

Define your application workflow as a series of steps. The visual console automatically graphs each step in the order of execution, making it easy to design complex workflows for multi-step applications. The following diagrams provide examples of the flow of steps – including sequential, branching and parallel steps, for a photo sharing application.

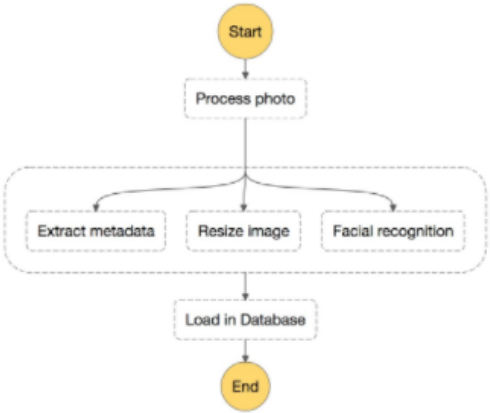
Sequential Steps



Branching Steps (Choice of Path)



Parallel Steps



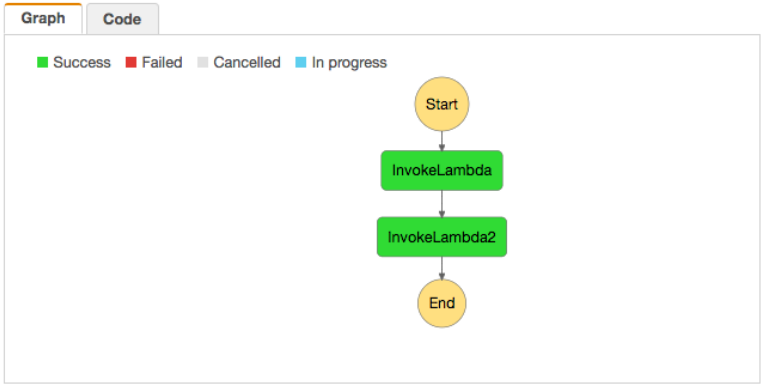
Execute and Monitor

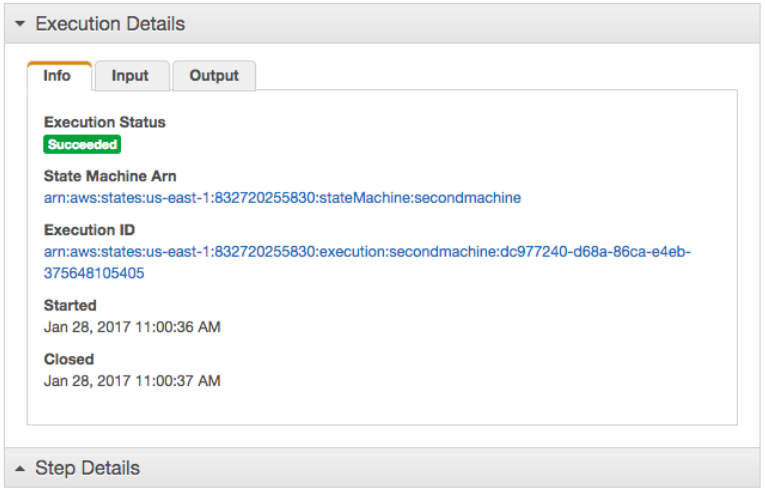
2. Verify Everything is Operating as Intended

Start an execution to visualize and verify the steps of your application are operating as intended. The console highlights the real-time status of each step and provides a detailed history of every execution.

AWS Step Functions

Simple Example





ID	Type	Timestamp
1	ExecutionStarted	Jan 28, 2017 11:00:36 AM
2	TaskStateEntered	Jan 28, 2017 11:00:36 AM
3	LambdaFunctionScheduled	Jan 28, 2017 11:00:36 AM
4	LambdaFunctionStarted	Jan 28, 2017 11:00:36 AM
5	LambdaFunctionSucceeded	Jan 28, 2017 11:00:36 AM
6	TaskStateExited	Jan 28, 2017 11:00:36 AM
7	TaskStateEntered	Jan 28, 2017 11:00:36 AM

Simple Example

Graph | Code

```
{
  "Comment": "A Hello World example of the Amazon States Language using an AWS Lambda Function",
  "StartAt": "InvokeLambda",
  "States": {
    "InvokeLambda": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:832720255830:function:helloworld",
      "Next": "InvokeLambda2"
    },
    "InvokeLambda2": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:832720255830:function:columbiaecho",
      "End": true
    }
  }
}
```

- AWS Step Functions Concepts
  - Amazon States Language
  - States
  - Tasks
  - Transitions
  - State Machine Data
  - Executions
  - Error Handling
  - Creating IAM Roles for Use with AWS Step Functions

```
{
  "Comment": "An Amazon States Language example using a Choice state.",
  "StartAt": "FirstState",
  "States": {
    "FirstState": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:FUNCTION_NAME",
      "Next": "ChoiceState"
    },
    "ChoiceState": {
      "Type" : "Choice",
      "Choices": [
        {
          "Variable": "$.foo",
          "NumericEquals": 1,
          "Next": "FirstMatchState"
        },
        {
          "Variable": "$.foo",
          "NumericEquals": 2,
          "Next": "SecondMatchState"
        }
      ],
      "Default": "DefaultState"
    },

    "FirstMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnFirstMatch",
      "Next": "NextState"
    },

    "SecondMatchState": {
      "Type" : "Task",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:OnSecondMatch",
      "Next": "NextState"
    },
```

Tasks and Activities

Tasks

All work in your state machine is done by *tasks*. A task can be:

- An Activity, which can consist of any code in any language. Activities can be hosted on EC2, ECS, mobile devices—basically anywhere. Activities must poll AWS Step Functions using the GetActivityTask and SendTask* API calls. (Ultimately, an activity can even be a human task—a task that waits for a human to perform some action and then continues.)
- A Lambda function, which is a completely cloud-based task that runs on the Lambda service. Lambda functions can be written in JavaScript (which you can write using the AWS Management Console or upload to Lambda), or in Java or Python (uploaded to Lambda).

Tasks are represented in Amazon States Language by setting a state's type to `Task` and providing it with the ARN of the created activity or Lambda function. For details about how to specify different task types, see Task in the Amazon States Language Overview.