MOSL Project

~~Senior Design - Group 2 - Fall 2020~~ Senior Project / Capstone - Spring 2021

Group Members:

> ***Lead**: ChrisOfNormandy: Chris Hinton*

> ***Dev**: AbusedLogic: Brennon Burke*

> ***Dev**: IcyClaw: James Williams*

> ***Dev**: ImportedGlue891: Isabel Casarez*

> ***Dev**: Slayer799: Zach Davis*

Overview

Development of a Microservices Orchestration Framework. Given a cloud-based microservices Enterprise Service Bus (ESB) framework that will include a microservices catalog with automated service registration, develop a domain-specific language establishing an enterprise integration framework and microservices standard (MOSL - Microservices Orchestration Service Language). MOSL would facilitate service registration, advertisement of the microservices catalog, a web-based user interface orchestration service and the scripting of microservices workflow.

Background

The ESB should implement a communications system between mutually interacting software applications - microservices. ESBs are designed to provide a uniform need of moving work-offering applications connections to the bus; to describe messages based on simple structure or business polices.

The Project

Create a framework that allows the orchestration of microservices; to merge the concept of cloud microservices with Service-Oriented Architecture(s) (SOA) implemented via an ESB; to create a more flexible microservice infrastructure. The end result should leverage an ESB for microservice registration and orchestration. Services will advertise their input, output and service availability. The MOSL will allow users to query the service catalogue for their desired service. The catalogue should respond with the useable microservice over the ESB. The MOSL protocol helps manage the growing number of microservices and their interfaces by creating a single domain communications standard and an established ESB for communication.

Semester Layout

First Semester

- ~~Design a domain-specific language (MOSL) that allows users to message between and link microservices on the ESB.~~
- ~~Define features, inputs, outputs.~~
- ~~Design UI and query engine.~~
- ~~Design microservice component management wrapper, allow user to assign microservices to a specific input or output. The microservice should act as a software agent.~~

Second Semester

- ~~Build the MOSL architecture.~~
- ~~Develop the UI. Users should drag-and-drop services around to orchestrate a workflow.~~
- ~~Develop the query engine. Should allow user discovery of microservices.~~
- ~~Build microservice component management wrapper.~~

Installation and Setup

This project requires Node 15.x and all dependencies listed in the `package.json` file. On Linux based systems, such as Ubuntu, when installing node modules using `npm install`, there may be an error stating:

> *gyp ERR! stack Error: not found: make*

The following is required to resolve this issue:

> *sudo apt-get install build-essential*

...and if not already installed, Python 3: https://docs.python-guide.org/starting/install3/linux/

To start the server, use:

> *node server.js*

This will launch two servers running on `3000` and `5000` (by default, which can be changed in the configs).

The server running on `3000` is the web server, responsible for inbound and outbound requests from browsers.

The server running on `5000` is the services server, responsible for making outbound calls to foreign addresses and hosting internal services that should not have direct ties to the web server.

There should be a directory `/temp` at the root of this project, which is not cloned from the project due to it being an empty directory.

This serves as a temporary file storage location for some services. It is not the most ellegant approach to what its purpose serves, but it works for now and can be replaced at a later time.

Development

A general outline of how to clone the project:

1. Clone this repository into any directory of your choice (example, your Desktop).

   > *git clone https://github.com/ChrisOfNormandy/senior-design.git*

2. Verify Node 15.x is downloaded. If not, get it here -> https://nodejs.org/en/

   > *node --version*

3. Open the repository in an IDE (VS Code, for example) and open a terminal inside the project directory.

   > *cd .../senior-design*

4. Execute the following to download the dependent packages:

   > *npm install*

5. When the installation has completed, ensure there is a directory `/node_modules` inside the root directory of this project.
6. To run the program, use:

   > *nodemon*

or

> *node server.js*

Nodemon is a development tool that automatically restarts the system when file changes are made. It should not be used for the live server.

Work should be pushed to a branch named after the ticket it is related to. For instance, MOSL-30 should be on branch `mosl-30` in the repository.

Branches should be deleted after they are merged with `dev`. Any bug fixes or patches for previous work should be assigned to new tickets and have branches assigned to those tickets, not previous.

Everything else should be its own branch, if unique, or on the `dev` branch, considered a "WIP master" branch. Always ensure your work is up to date with the `dev` branch, which in turn should remain up to date with `master`.

Configuration

After cloning the project repository, create a directory `/config` in the root dir (the same place `server.js` and `/node_modules` are located).

Inside, create a new file `config.json`.

Paste the following code within `config.json`:

```
{
  "domain": "http://localhost",
  "port": 3000,
  "aws": {
    "path": "Absolute path to your aws-credentials.json file"
  },
  "services": {
    "domain": "http://localhost",
    "port": 5000,
    "directories": {
        "list": []
    }
  }
}
```

Domain: Any root url. LocalHost = the host device. Axios requires the protocol, http, to use local host. This can be an IP, but must include the protocol. Port: The port the server will listen to.

Combined, `domain:port/{path}` will be the URL format for pages of this system.

AWS: The credentials, configs for the AWS SDK per user. The format of the `aws-credentials.json` file is:

```
{
    "accessKeyId": "ABCDEFGEXAMPLE",
    "secretAccessKey": "ABCDEFGHIJKLMNOPQRSTUVEXAMPLE",
    "region": "us-east-1"
}
```

Access Key ID: The AWS access key to be used for all services requiring access to the AWS SDK.

Secret Access Key: The AWS secret key associated with the access key provided.

Region: The region designated for services utilized within AWS.

This file follows the rules defined at https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html

The `aws-credentials.json` file can be put anywhere on your computer, since it is referenced as an absolute path in the `config.json` file. If desired, saving it within `/config` is suggested during development, since it is ignored by git.

DO NOT PUSH THIS FILE TO GITHUB, and DO NOT SHARE YOUR ACCESS KEY OR SECRET.

Services: The server responsible for handling internal services and requests for services stored in configs on this system.

Amazon Web Services

This project contains services incorporating the AWS SDK - primarily S3.

If these services are undesired, they are located in `/ESB/services/`, referenced in `ESB/esb.js` and called from `/ESB/server.js`.

They may be removed without any impact to the rest of the system.

Additionally, any configs within `ESB/configs` may be removed or replaced without any impact to other services or the web server.