Hammer-io IoT Extension

Project Plan

Team: sddec19-24
Client/Advisor: Lotfi Ben-Othmane
Members:

Yussef Saleh	*Team Leader*
Matt Bechtel	*Chief Designer / Engineer*
Brett Wilhelm	*Assistant Programmer / Engineer*
Chakib Ahlouche	*Assistant Programmer / Engineer*

Email: sddec1924@gmail.com
Website: http://sddec19-24.sd.ece.iastate.edu

Table of Contents

i - Glossary

(See references at the bottom for official documentation of given technologies)

- API
    - Application Programming Interface
- Docker
    - A virtualization technology
- Express
    - lightweight web framework for NodeJS that provides easy set up and extensibility
- Git
    - Version control software
- Hammer-io
    - Existing senior design project that we are extending
- Heroku
    - An enterprise deployment service that is used by the existing Hammer-io system
- IoT Device
    - ‘Internet of Things’ Device, this term is fairly broad and usually refers to a low weight device (such as a raspberry pi) that is connected to the internet. In this Project Plan we will narrow this term a bit defining it as a low resource device that runs Linux, and in particular does not have enough resources to a run virtualization technology, such as Docker.
- Linux
    - Operating system
- Microservices
    - A type of software architecture that involves making logical splits between components such that each component is loosely coupled with the others in the system. Typically each component (service) in the system is stateless, relying on a data store to house information pertinent to system, this allows for each component (service) in the system to be scaled as needed. The existing Hammer-io system is built using a microservices architecture, our new components will simply add to it.
- NodeGit
    - Library for NodeJS that provides an API to Git commands
- NodeJS
    - Javascript for the backend
- Repository
    - In this paper when we mention repository, we mean a Git repository.

1 - Problem Statement

The problem that has prompted this project is the need to extend the existing Hammer-io system to support the deployment of software to client's IoT devices. Some caveats come with this problem though, such as the restriction that the IoT deployment can only rely on the fact that the IoT device is running some sort of Linux. With this requirement comes the request that we support a wide range of IoT devices, thus we cannot expect to be able leverage the power of virtualization technologies such as Docker, as they take up too much of the device's resources.

2 - Project Goals and Deliverables

The goal of the project, as mentioned above, is to provide a extension to the current DevOps system, Hammer-io, that will allow clients to deploy their services to the IoT device of their choosing. As of now, the system only allows deployment via Heroku, so we will have to deliver an extension to Hammer-io that allows for the deployment of software to IoT devices. This extension will be in the form of the new services, the 'IoT Deployment Service' and the 'IoT Deployment Manager'. These new services will add on to the existing microservices architecture of the Hammer-io system, and once integrated, should allow for clients to use Hammer-io to push and deploy services to their desired IoT device. The two new services will work in conjunction to provide this new functionality, with the 'IoT Deployment Service' running alongside the rest of the Hammer-io system in a microservices architecture, and the 'IoT Deployment Manager' running on the actual IoT device.

The deliverable will be as mentioned above, the extended Hammer-io system with the new services 'IoT Deployment Service' and 'IoT Deployment Manager' added on, to provide the needed IoT deployment functionality to the client.

3 - Proposed Design

Our extension design for the Hammer-io DevOps system will involve the creation of a new 'IoT Deployment Service', and its companion (which will run on the IoT device itself) the 'IoT Deployment Manager', for the deployment of client code onto Linux-based IoT devices. The existing Hammer-io codebase was built using a microservices architecture, so we will simply add another microservice and integrate it with the existing system.

To integrate the 'IoT Deployment Service' with the existing services, endor, koma, tyr, and the frontend yggdrasil, we will have to extend them so that they may communicate with our new deployment service in the same fashion they communicate with Heroku (the deployment service currently used by Hammer-io for deployment of docker containers).

When implemented, the 'IoT Deployment Service' will field requests from the UI, Yggdrasil, to handle a client's deployment. To start a deployment pipeline, the client will need to 1. Setup their IoT device so that it is running Linux and is connected to the internet, 2. install our instance side service, the 'IoT Deployment Manager', and run it on a port which they have opened to the public (or at least open to the 'IoT Deployment Service's' IP). From here, the client should go through the UI, Yggdrasil, to pass off their IoT device endpoint (IP plus the port), the git repository path of desired software to run, credentials of the repository (if any), and the launch script for the code in the repository.

Once the 'IoT Deployment Service' has the information necessary to talk to the client's IoT device, and the repository of the code to be ran, the client can then kick off their deployment. Deployment will involve a request from Yggdrasil to the 'IoT Deployment Service' to begin deployment. When the 'IoT Deployment Service' receives a request, it will authenticate it, and then, in turn, send it off to the 'IoT Deployment Manager' which is running on the IoT device that the client specified. The 'IoT' Deployment Manager' will then handle the request and perform the given deployment task. Deployment tasks received by the 'IoT Deployment Manager', running on the IoT device, can include, but are not limited to: cloning a new repository onto the IoT device, uploading a start script for a given repository, running the start script for a given repository (in turn running, 'deploying', the clients code onto the IoT device), checking the status of a running program controlled by the 'IoT Deployment Manager', shutting down a program, pulling down new commits in a given repository and starting up the new code, etc.

The end goal of the 'IoT Deployment Service' and its companion the 'IoT Deployment Manager' will be to function as a deployment service, such as Heroku, but instead of deploying to the cloud, it will deploy software onto a client's IoT device.

4 - Design Specifications: IoT Deployment Service

4.1 - Technical Specifications

- NodeJS web server leveraging the Express framework
- Must be stateless, fitting into the existing microservice architecture

4.2 - Functional Requirements

- Provides API for deployment requests
    - Initial Setup
        - Sets up initial communication with IoT instance, using the credentials provided by the client (endpoint)
        - Passes clients version control credentials and the client code's start script to our service, 'IoT Deployment Manager' via HTTPS, which is running on the client's IoT instance
        - Once the connection is successfully set up, the 'IoT Deployment Service' will set the state of the client's instance to 'Ready for Deployment'
    - Deployment
        - If the instance is 'Ready for Deployment' the 'IoT Deployment Service' will send a request to the running 'IoT Deployment Manager' to tell it to pull the code and check for updates, if there are updates, the manager will run the client's code with the given start script that was provided by the client
        - The 'IoT Deployment Service' will field a request from the 'IoT Deployment Manager' to notify the client about the state of their deployment (this notification will be passed back to the UI for client viewing)
- Provides API for the client to acquire feedback about a client's instance
    - Instance State
        - Information about whether the software deployed into the IoT device is running/in a healthy state
    - Deployment History
        - Information about the deployment history of the instance, this should include, deployment times, commit information (version) about said deployments, user that executed said deployment

4.3 - Non-Functional Requirements

- Usability
    - The 'IoT Deployment Service' API should be easy to use and should be easily extensible if new functionality is needed
- Supportability
    - The 'IoT Deployment Service' should be written in NodeJS and should be usable with (supported by) Linux systems
- Reliability
    - The service should be continuously running so that clients can control their instances with complete reliability
    - Uptime should be 99.9% meaning that updates to it must be done in a rolling fashion
- Security
    - The service must communicate with the client's instance over a secure connection (HTTPS) as sensitive information must be exchanged

5 - Design Specifications: IoT Deployment Manager

5.1 - Technical Specifications

- NodeJS web server leveraging the Express framework
- Must be stateless
- Must be lightweight enough to run on an IoT device

5.2 - Functional Requirements

- Provides API for deployment requests
    - Initial Setup
        - Controller exists to receive a request with the client's information, including git credentials and start up script for the client's code
    - Deployment
        - Controller exists to handle requests to deploy (start) software
        - Controller exists to update a given repository on the device (this will be a fetch and a merge, or as many know it, a pull)
    - Security
        - In order for the API requests to be secure, the Deployment Manager running on the instance needs to be provided the 'IoT Deployment Service's IP address so that the manager knows that the request to setup the instance is coming from the correct IP
            - This identity check could also be done using an RSA key pair (this is undecided as of now)

5.3 - Non-Functional Requirements

- Usability
    - The ‘IoT Deployment Manager’ API should be easy to use and should be easily extensible if new functionality is needed
- Supportability
    - The ‘IoT Deployment Manager’ should be written in NodeJS and should be usable on (supported by) Linux IoT devices
- Reliability
    - The service should be continuously running on the IoT device, so that clients can control their instances with complete reliability, this means we must be very careful when performing error handling, we cannot have the program crash.
- Security
    - The manager must verify the identity of the service sending the request for deployment to it
        - This could be done through whitelisting or through signing using an RSA key pair

6 - Previous Work / Literature Review

Because our project is an extension of a previous senior design project, we met with one of the members of the previous team to get a broad overview of the current project. This meeting was extremely informative and gave a great scope of what the project can and can’t do at the moment.

Along with meeting with the previous team to discuss the state of the project, we researched the purpose of following the DevOps ideology, and how it applies to our project. The research into the existing project and DevOps in general, gave us a great starting place for our design.

7 - Market Survey

Since our project is an extension to an existing system, our market survey did not influence our decision to go forward, but it did aid in the development of our design. Surveying the DevOps landscape, we found that usability, security, and reliability are key aspects of any enterprise deployment tool. In our design we used the knowledge gained from this survey to ensure that we conform to industry standards/expectations.

8 - Assessment of Proposed Solution

Our team has created a viable extension to the existing Hammer-io system that clients may use to deploy their code to their IoT devices in an automated fashion. The extension relies on the Hammer-io project's existing microservices architecture to be able to leverage easy integration and extensibility.

The fairly straightforward extension that we have proposed should provide the solution to our client's problem that they are looking for, a way to leverage the Hammer-io system to create a deployment pipeline to user's IoT devices.

Following the previous team's lead, our extension will provide an easy to use piece of functionality so that user's of the Hammer-io for will be able to both automate and simplify their deployment at an enterprise level.

Looking forward, we will continue to refine our design, thinking about the nuances of such an extension, hopefully ending next semester with a completely functional, easy to use, service for continuous deployment to IoT devices.

9 - Project Timeline

9.1 - First Semester

The following chart outlines the proposed timeline for the project's development into the first semester. The blue bars indicate project phases.

Jan				Feb				Mar				Apr				May			
W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4
			🟦 Req. Gathering	🟦	🟦	🟦	🟦												
						🟦 Research	🟦	🟦	🟦										
										🟦 Begin IoT Deployment Service and Manager	🟦	🟦	🟦	🟦					
															🟦 Demo	🟦			

9.2 - Second Semester

The following chart outlines the proposed timeline for the project's development into the second semester. The blue bars indicate project phases.

Aug				Sept				Oct				Nov				Dec			
W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4
🟦 Build IoT Deployment Service and Manager	🟦	🟦	🟦	🟦	🟦	🟦	🟦												
						🟦 Integration	🟦	🟦	🟦	🟦	🟦	🟦							
								🟦 Testing, Validation, Polishing	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	🟦	

10 - Challenges: Risks / Feasibility Assessment, Cost Considerations

- Risks
    - Integrating the new components, ‘IoT Deployment Service’ and ‘IoT Deployment Manager’ may take longer and be more challenging than expected
        - Chance of occuring: MEDIUM
        - Since we are extending an existing system by adding two new components to the architecture, we must integrate those components so they work with the system as a whole. This ‘integration’ will involve modification to the existing codebase, which our group did not have a part in developing. With this, comes the risk that the integration may take longer than anticipated in the project plan. This risk could have a major impact on the project
    - The scope of our extension is too broad
        - Chance of occuring: MEDIUM
        - The endeavor we are taking on in this extension is a large and complex one, involving the implementation of a vast range of functionality. The risk associated with this will flesh itself out once development really heats up, as we will have a better gauge as to the speed at which we are producing functionality.
    - The knowledge level of the team, in the given technologies, may lead to slow development
        - Chance of occuring: LOW
        - This risk is inherent with any team working on a project who does not have much experience using the technologies associated with the given project. For us, this risk applies specifically to our knowledge as a collective of NodeJS and Express, as well as the existing Hammer-io system.
- Feasibility
    - Is producing such an extension to Hammer-io feasible? We believe so, based on our assessment of the problem and the solution we have come up with. The reason we believe this, is because we have gotten a pretty good start on the actual development of our two new components and we feel confident that with more experience, coming into next semester, that we can achieve our goal.
- Cost Considerations
    - The cost of the development for the extension will be extremely low as we are only using free, open source software in our system. The only cost associated with system when complete, will be the cost to host the Hammer-io system, which will scale based on user traffic (microservices architecture facilitates this).

11 - Standards

For this project we will follow the ethical standards defined by the software development community, including those defined by IEEE. We will also follow common development standards, including ones that outline how sensitive information should be handled. These are particularly important as we are requiring a client to run our code on their device whilst simultaneously requiring that the client provide us with their credentials to external services.

The standards used during development are:

- Code Review
  - For deployment, we will push each new feature to a new branch in the git repository, otherwise known as a ‘feature branch’
  - For each feature branch to be merged with master, and therefore deployed, it must pass a code review by at least 2 developers
  - To pass a code review, a commit must contain new code that:
    - Is correct
    - Integrates with the existing code
    - Passes all unit, integration, and regression tests
    - Implements the desired functionality
    - Is free from any other errors (ex: syntactical errors)
  - During a code review, reviewers will communicate to the developer who wrote the code about any issues that they come across. It is the responsibility of the developer whose code is being reviewed to fix any issues with it.

- Version Control
  - We will version our software via Git. This is the most widely used versioning software on the market and is the industry standard. Using git will allow us to keep records of each change (commit) to a file in our repository, including the developer who made the change, when the change was made, and the message left by the developer of the commit about the changes made.
  - For documentation we will use Google Docs, allowing us to look back at our change history for each document.

- Testing Standards
  - Unit Tests
    - Each commit pushed to the remote repository should contain unit tests that correspond to the functionality added in that commit
    - The author of the new commit is also responsible for producing unit tests for its functionality. In order to pass code review, these tests must be verified by the reviewer to cover every line of the new code (100% code coverage).
  - Integration Tests
    - For each new component we will require that there be tests written to verify its functionality in the system as a whole. These will be important markers for how the system will function once deployed.
  - System and Acceptance Testing
    - Manual testing by developers will be the standard for System Testing. This will take place upon merging new code and upon completion of major portions of functionality.
    - Product demos to the client will be the standard for Acceptance Testing. These take place on a weekly basis at the team meeting.
  - Code Coverage
    - To verify that our unit tests cover every line of code, we will leverage an existing code coverage tool, likely an IDE plugin. The specific tool to be used will be consistent across the development team, the tool to be used will be decided next semester when the development ramps up.

12 - Test Plan: Functional Testing

12.1 - Manual Testing

Much manual testing will have to be done to verify the functionality of our extension, because of the nature of it. Such a thing is a bit hard to test using automated testing methods especially for the integration portion of it. We will perform automated regression testing on the existing system to make sure that our newly added pieces to endor, koma, and yggdrasil do not break any functionality, but beyond that we will have to go through the use cases of our extension manually to verify that the functionality of the system is what is expected. This manual testing will involve standing up the Hammer-io system, with our extensions active, and performing a deployment onto an IoT device. Of course for convenience sake, this IoT device will simply be a Docker container running Raspbian so that the dev team may have an easily reproducible IoT environment to test with.

12.2 - Automated Testing

Automated testing for such an extension will be done leveraging Mocha and its unit testing functionality. Such tests will be ran automatically before the new code is pushed to production to verify that the code will function as expected once deployed.

12.3 - Regression Testing

Regression Testing will be done every time a new build is created to make sure that changes to the code do not break the existing functionality. This testing will be done by executing all unit tests and verifying that they pass.

13 - Test Plan: Non-functional Testing

Much of the non-functional testing will be done manually, as it is difficult to measure many of these qualities with any sort of tool. For example, ensuring ease of use is not really something that we can test with a piece of software.

Looking specifically at 'IoT Deployment Service', it should be fairly simple to manually ensure that our API is easily extensible and easy to utilize. Supportability requirements are similar in this aspect, as it will be fairly telling to us that our systems do not work on the necessary systems should our testing environment stop working properly. In establishing that the service meets our standards for reliability, we will aim to be able to utilize load balancing techniques. Finally, as was previously stated, we will be utilizing HTTPS to guarantee secure data transmission, something that will be a simple process to test for.

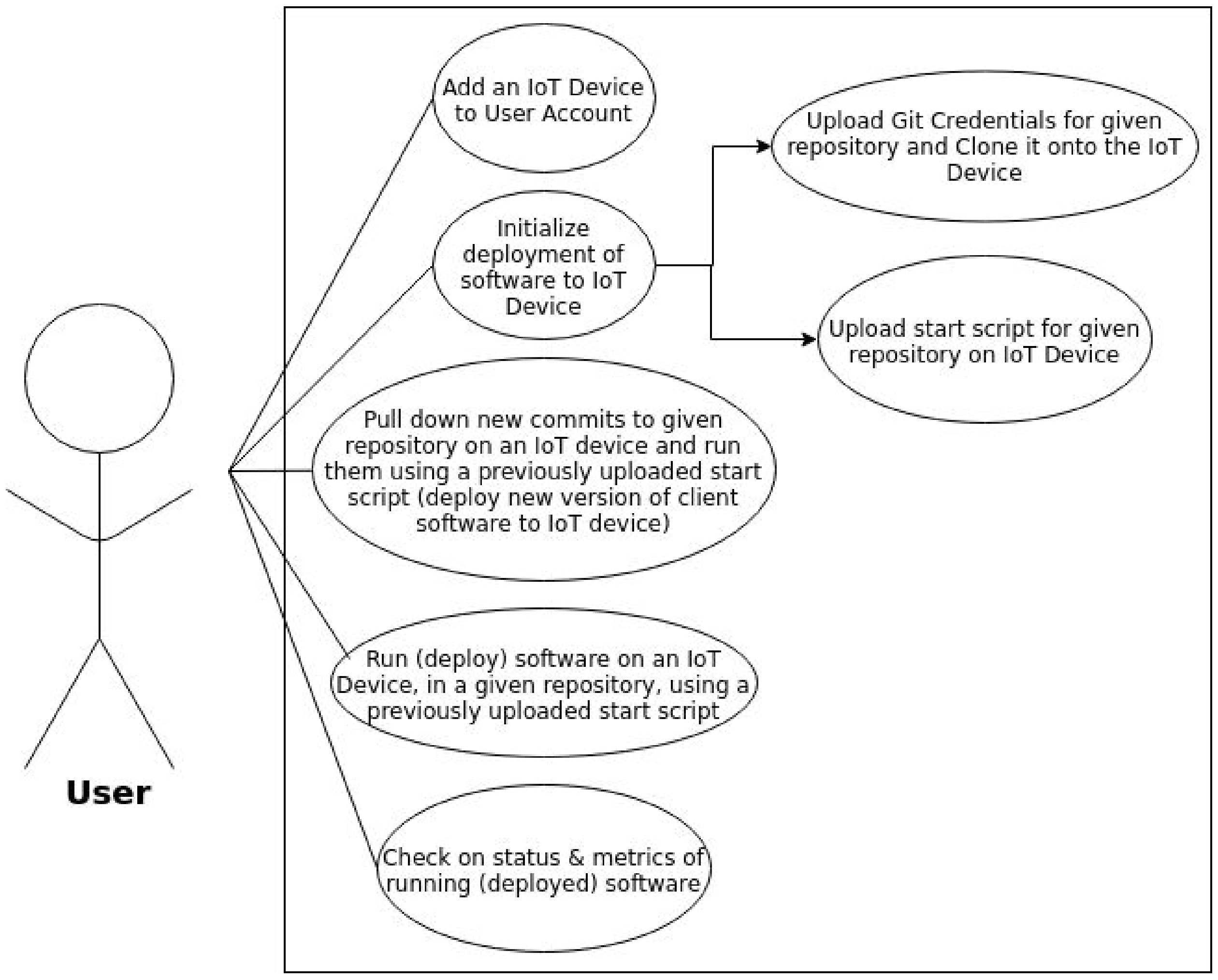
Moving on to 'IoT Deployment Manager', we see only a few differences. With regard to reliability, we will utilize a sort of heartbeat testing to confirm the uptime of the service. Finally we will be able to manually test the effectiveness of our authentication method, whether it be a whitelist or the use of an RSA key pair.

14 - Test Plan: Validation

This project is an extension for a DevOps system, we will be able to continually analyze if our solution is valid because of our agile approach. Validation with the client will take place at weekly meetings where we demo our progress and brief the client on the direction and planned development for the following week.

15 - Use Case Diagram For The IoT Deployment Extension

15.1 - Use Case Diagram


Add an IoT Device to User Account
Initialize deployment of software to IoT Device
Upload Git Credentials for given repository and Clone it onto the IoT Device
Upload start script for given repository on IoT Device
Pull down new commits to given repository on an IoT device and run them using a previously uploaded start script (deploy new version of client software to IoT device)
Run (deploy) software on an IoT Device, in a given repository, using a previously uploaded start script
Check on status & metrics of running (deployed) software
User

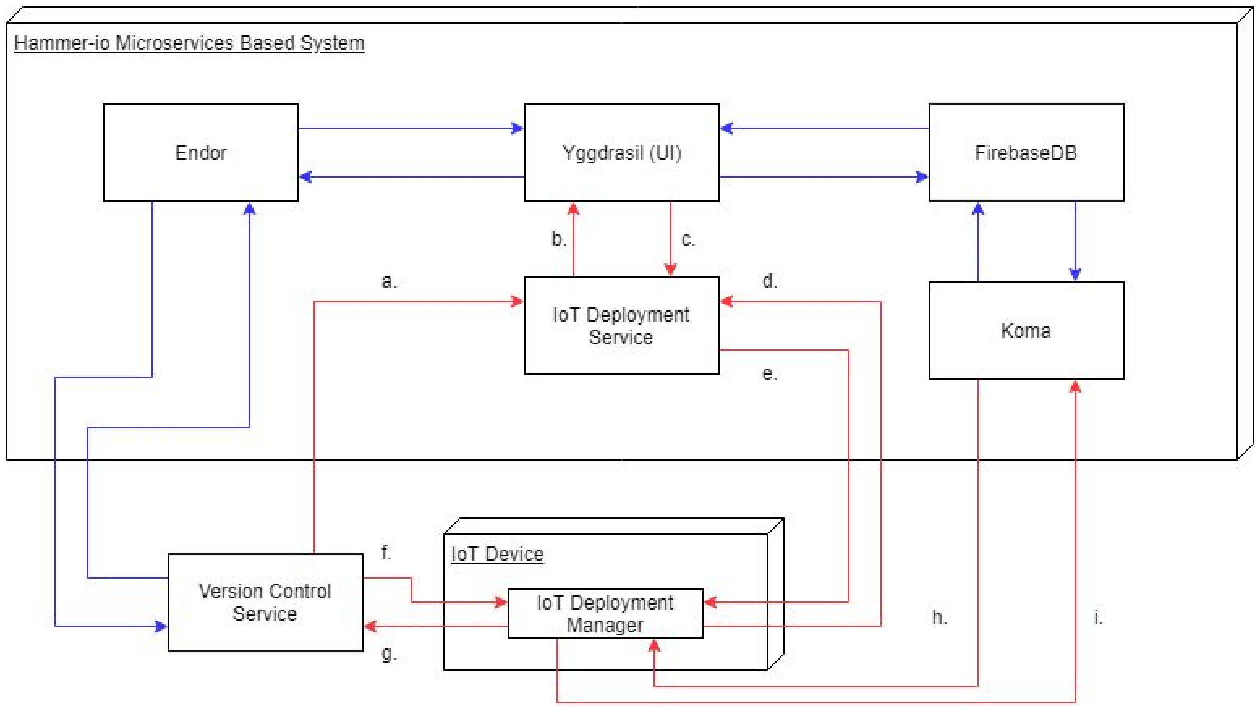
15.2 - Description of Use Case Diagram

- **Add an IoT Device to User Account**
    - This will involve the user adding the IP of the IoT Device they wish to add to their account. This device must have our IoT Deployment Manager running on it, serving on a publicly accessible port (or the client could whitelist the IoT Deployment Service's IP on the port). This will add the device's endpoint to the client's list of IoT devices within the IoT Deployment Service's database.
- **Initialize deployment of software to IoT Device**
    - **Upload Git Credentials for given repository and Clone it onto the IoT Device**
        - Here, the user will upload their git credentials for connecting to their private repository. These credentials should either be for use with an HTTPS or SSH connection to the remote repository.
    - **Upload start script for given repository on IoT Device**
        - Here, the user will upload the start script for their software. This script should be an executable bash script which is meant to be ran in the top level of the repository in which the software resides.
- **Pull down new commits to given repository on an IoT device and run them using a previously uploaded start script (deploy new version of client software to IoT device)**
    - This will be done via an HTTP request to the IoT Deployment Service specifying the repository in which pull down the new commits to. This HTTP request can be made either through the UI (Yggdrasil) or via Git Hooks in the remote repository. These Hooks will likely be placed on the pushing of new commits to the repository.

      When the IoT Deployment Service receives a request to run new code as detailed above, it will make a request to the existing Hammer-io system to verify the users credentials. Once verified, the Deployment Service will pull the client's IoT Device list from the database and proxy the update request from the user to the correct IoT Deployment Manager running on the correct IoT Device.
- **Run (deploy) software on an IoT Device, in a given repository, using a previously uploaded start script**
    - Here, the user will make a request via the UI (Yggdrasil) to start the code in a given repository. In order to do this the user must have previously uploaded a start script for the IoT Deployment Manager to use to start the code in the given repository.
- **Check on status & metrics of running (deployed) software**
    - This case will involve the user querying the IoT Deployment Service, via the UI, to get information about their deployed software. The IoT Deployment Service will call off to the IoT Deployment Manager running on the specified IoT Device to get information about the running processes started by the manager.

16 - Block Diagram For The IoT Deployment Extension

16.1 - Block Diagram



16.2 - Block Diagram Description

Blue Arrows: Existing HTTP communication
Red Arrows: New HTTP communication

New HTTP Communication Description

a. Git Hooks sending request to the Deployment service with user credentials. This request will kick off the continuous deployment process, calling for an update of the code running on the IoT device. This Git Hook will be used upon each new commit pushed to the master branch.

b. Response to requests from the User Interface, Yggdrasil. These will contain data that will then be displayed to the user.

c. Requests from Yggdrasil to the IoT Deployment Service. These will include all requests to the IoT Deployment Service API, used for managing and deploying software to the clients IoT devices.

d. Responses from the IoT Deployment Manager to the IoT Deployment Service. These will contain data pertaining to the state of the client's software running/to be ran on the IoT device.

e. Requests from the IoT Deployment Service to the IoT Deployment Manager running on the IoT Device. These will include all requests that pertain to the running and managing of the client's software on the IoT Device.

f. Responses from the remote repository controlled by the Version Control Service, which can be any Git based repository service such as GitHub or GitLab, to the IoT Deployment Manager. These will be the result of Git commands ran by the IoT Deployment Manager running on the IoT device.

g. Requests from the IoT Deployment Manager to the Version Control Service, these will be via NodeGit, a wrapper for the libgit2 GitHub git library written in C. These should just be basic Git HTTP requests such as clone, fetch, etc. The IoT Deployment manager will never push to the remote repository.

h. Responses from Koma upon the IoT Deployment Manager sending data to it. This data will contain statistics about the deployed software.

i. Requests to Koma from the IoT Deployment Manager pushing data about the state of the software running on the device. This will data will end up in Firebase for Yggdrasil to pull from and display to the user.

- **Yggdrasil**
    - Front end web service for Tyr. Everything that Tyr can do can also be done through Yggdrasil, such as setup and deploy a project. Yggdrasil, beyond deploying, can allow the user to monitor and view data from their systems (via Firebase, provided by Koma). Essentially, Yggdrasil is the 'DevOps' hub, similar to the Amazon Web Services Management Console Website.
- **Endor**
    - Backend service for Yggdrasil. Endor handles all of the user information from the Yggdrasil site as well as all user requests that do not involve simply viewing the data collected from their deployed system, as that is done via Firebase and Koma. Endor is like a wrapper to Tyr that takes web requests instead of direct commands.
- **Koma**
    - Koma collects the data from each deployed service, aggregates it based on the Hammer-io users account, and sends it off to be stored in Firebase.
- **Firebase**
    - Firebase receives real-time data from Koma about the deployed instances, and stores them for retrieval by Yggdrasil. From this, users can view data about their running instances in near real-time.
- **Version Control Service**
    - This will be the host of the Remote Git Repository that holds the client's code. Upon new commits pushed to the repository the git hooks placed in the repository should make a request to the IoT Deployment Service with information about the repository to pull new commits from. This request will then be proxied by the IoT Deployment Service to the correct IoT Device running an instance of our IoT Deployment Manager.
- **IoT Deployment Service**
    - The IoT Deployment Service is a new component that we have added to the Hammer-io system. This component will field HTTP requests corresponding to the deployment and management of software running on the IoT device. The Deployment Service will authenticate the requests coming in and proxy them to the correct IoT Deployment Manager running on the specified IoT Device.
- **IoT Deployment Manager**
    - The IoT Deployment Manager is another new component created, to be ran on the client's IoT Device. The 'Manager' will be a web server that fields requests to deploy and manage the client's software running on the IoT Device. Once authentication is passed, the 'Manager' will perform the given task, whether it be to clone a new repository, pull down new commits, start up the code in a given repository, etc.

Conclusion

The extension design that we have laid out above should provide a much needed way for DevOps engineers to deploy their services to the IoT devices of their choosing, while also keeping usability, reliability and security, simplistic and ensured. Such an extension to the existing DevOps framework, Hammer-io, should boost its enterprise compatibility, hopefully forcing it into the ever growing IoT space and allowing for our clients leverage it for a solution of their choosing.

Overall, the extension should affect the Hammer-io project as whole in a very positive manner, granting new clients and old, the ability to automate their IoT service deployment.

References

Github: https://developer.github.com/v4/
Hammer-IO: http://sdmay18-19.sd.ece.iastate.edu/docs/
Mocha: https://mochajs.org/api/mocha/
NodeJS: https://nodejs.org/en/docs/
NPM: https://docs.npmjs.com/
Docker: https://docs.docker.com/
NodeGit: https://www.nodegit.org/api/
Original Hammer-io project plan: https://hammer-io.github.io/docs/Project_Plan_v3.pdf