Tapis API Development with Python: Best Practices In Scientific REST API Implementation

Experience implementing a distributed Stream API

SEAN B. CLEVELAND, University of Hawaii
ANAGHA JAMTHE, Texas Advanced Computing Center
SMRUTI PADHY, Texas Advanced Computing Center
JOE STUBBS, Texas Advanced Computing Center
MICHAEL PACKARD, Texas Advanced Computing Center
JULIA LOONEY, Texas Advanced Computing Center
STEVE TERRY, Texas Advanced Computing Center
RICHARD CARDONE, Texas Advanced Computing Center
MAYTAL DAHAN, Texas Advanced Computing Center
GWEN A. JACOBS, University of Hawaii - System, USA

In the last decade, the rise of hosted Software-as-a-Service (SaaS) application programming interfaces (APIs) across both academia and industry has exploded, and simultaneously, microservice architectures have replaced monolithic application platforms for the flexibility and maintainability they offer. These SaaS APIs rely on small, independent and reusable microservices that can be assembled relatively easily into more complex applications. As a result, developers can focus on their own unique functionality and surround it with fully functional, distributed processes developed by other specialists, which they access through APIs. The Tapis framework, a NSF funded project, provides SaaS APIs to allow researchers to achieve faster scientific results, by eliminating the need to set up a complex infrastructure stack. In this paper, we describe the best practices followed to create Tapis APIs using Python and the Stream API as an example implementation illustrating authorization and authentication with the Tapis Security Kernel, Tenants and Tokens APIs, leveraging OpenAPI v3 specification for the API definitions and docker containerization. Finally, we discuss our deployment strategy with Kubernetes, which is an emerging orchestration technology and the early adopter use cases of the Streams API service.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability; Network reliability.

Additional Key Words and Phrases: API, real-time streaming data, Tapis, CHORDS, Abaco

ACM Reference Format:
Sean B. Cleveland, Anagha Jamthe, Smruti Padhy, Joe Stubbs, Michael Packard, Julia Looney, Steve Terry, Richard Cardone, Maytal Dahan, and Gwen A. Jacobs. 2020. Tapis API Development with Python: Best Practices In Scientific REST API Implementation: Experience implementing a distributed Stream API. In *Practice and Experience in Advanced Research Computing (PEARC '20), July 26–30, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3311790.3396647

1 INTRODUCTION

As more scientific research moves towards distributed and cloud computing, the need for software tools and services to communicate effectively has become ever more important. Application Programming Interfaces (APIs) are how software applications can communicate and exchange information. REST APIs have become the defacto standard for many applications that need to communicate over the internet to support this distributed form of communication. Robust and trustworthy APIs are required to support increasingly complex technology stacks that supports cutting edge science, requirements for the services to be rolled into scientific gateways and middleware frameworks to enable easy access for the broader research communities. Tapis [10], an open-source Software-as-a-Service API framework, provides a rich set of production quality APIs to support distributed scientific computational experiments. As Tapis has evolved, the best practices for the development of these APIs has become a critical aspect of the implementation. Tapis APIs can be developed in any programming language, however, this paper focuses on the Python implementation of a Tapis API. In this paper, we will introduce the best practices for the development and deployment of Tapis APIs that leverage other Tapis services, with a focus on Python and Tapis development libraries using the new Streams API as an example use case.

2 BACKGROUND AND MOTIVATION

2.1 APIs & Microservices

An API is a software intermediary that allows two applications to talk to each other. It is a mechanism that takes user requests and tells the system what actions to perform and brings the response back to the user [7]. An API defines functionalities that are independent of their respective implementations, which allows those implementations and to vary without impacting consumers of the API. Most of the applications you access utilize APIs to allow your laptop or mobile device to access data and services hosted somewhere else - most of the Web is powered by APIs.

Microservices, also known as the microservice architecture, is an architectural style that structures an application as a collection of services. The main idea is that the services are easily maintainable, testable, loosely coupled to other services and independently deployable. Microservices typically are HTTP-based APIs built on open standards such as REST. All leading cloud providers, including Amazon AWS, Google Cloud Platform, and Microsoft Azure provide such services. Microservice architectures have significantly reduced barriers to distributed application development and have enabled new usage patterns across industry and academia. In contrast to the traditional monolithic service architecture, Tapis has adopted the microservice architecture to build resiliency and scalability in the platform. Services are loosely coupled with each other, thereby allowing easy maintainability and independent development schedules. End users and applications interact with Tapis by making authenticated HTTP requests to Tapis's public endpoints.

2.2 REpresentational State Transfer (REST)

REST, unlike SOAP, is not a WS (web service) standard but an architectural style for web applications. REST was devised by Roy Fielding in his doctoral dissertation [16]. REST makes use of existing web standards (HTTP, URL, XML, JSON, MIME types). Although it is most often used in the context of HTTP, REST is an architectural design pattern and not a communication protocol. Since REST is not a formally defined protocol, there are many opinions on the details of implementing REST APIs. However, the following five criteria must be fulfilled for any application to be considered RESTful:

- *Client-Server* A client-server architecture allows a clear separation of concerns. The client is responsible for requesting and displaying the data while the server handles data storage and application logic.
- *Statelessness* Communication between client and server is stateless. This means that every client request contains all information necessary for the server to process the request.
- *Caching* Stateless client-server communication can increase server load since some information may have to be transferred several times, so requests that only retrieve data should be cache-able.
- *Layered system* A feature of most networked systems, for REST this means that a client can not distinguish, nor care if it is directly communicating with the server or an intermediate proxy.
- *Uniform interface* REST defines a set of well-defined operations that can be executed on a resource.

A key concept of REST (and the HTTP protocol) are resources. A resource can be anything that is uniquely addressable, for example, in the Streams API, a project, site, instrument, and variable are all examples of resources (Table 1). The HTTP protocol defines a set of operations that can be executed on a single or multiple resources:

- *GET* A safe read-only method that reads a single resource or a list of resources.
- *HEAD* Same as GET but only returns the header and no data.
- *POST* Creates a new resource.
- *PUT* Completely replaces the resource(s) at the given location with the new data.
- *PATCH* Merges the resource(s) at the given location with the new data.
- *DELETE* Deletes the resource(s) at a location.

REST assumes the methods GET, HEAD, PUT, DELETE to be idempotent (invoking the method multiple times on a specific resource has the same effect as invoking it once. In order to support best practices for a REST API Tapis APIs implement the above criteria.

2.3 JavaScript Object Notation (JSON)

Tapis uses JSON to describe request and response data for all of its APIs. JSON is a lightweight data-interchange format on the web. The data exchange mainly takes place between a server and a browser. It is easy for humans to read and write and also it is easy for machines to parse and generate. A JSON object is organized as comma-separated key value pairs, and the entire object is wrapped up in curly braces. JSON objects are very useful to transmit and deliver data using REST APIs [7].

2.4 Containers

The use of containers for application packaging has grown exponentially in recent years. It is one of the most highly adopted technologies used by software professionals. Use of the Docker platform, to package applications along with all its assets such as software libraries and dependencies is a standard deployment practice followed lately [19]. This bundling of the application assets into a single package minimizes the dependencies on the execution environment, and containers can be executed from the image on any machine where the container run-time is available. Although containers provide portability and reproducibility for general applications, containers cannot easily encapsulate all dependencies of scientific code running in customized, heterogeneous environments. Scientific computations rely on specialized hardware and custom software installed across a computing infrastructure, including MPI, networking, mathematical and GPU libraries, and specialized workflow managers, that cannot be captured within a container. The

Tapis platform therefore aims to augment container technology with metadata captured by the platform that describes the requirements of applications as well as capabilities of different execution environments [21].

2.5 The Tapis Framework

Tapis is an open source, NSF funded Application Program Interface (API) platform for distributed computation. It provides production-grade capabilities to enable researchers to 1) securely execute workflows that span geographically distributed providers, 2) store and retrieve streaming/sensor data for real-time and batch job processing, with support for temporal and spatial indexes and queries (Figure 1), 3) leverage containerized codes to enable portability, and reduce the overall time-to-solution by utilizing data locality and other "smart scheduling" techniques, 4) improve repeatability and reproducibility of computations with history and provenance tracking built into the API, and 5) manage access to data and results through a fine-grained permissions model, so that digital assets can be securely shared with colleagues or the community at large.

The Tapis framework is RESTful API platform that can be leveraged as a hosted solution or distributed between various institutions. The central hosted instance is currently hosted by the Texas Advanced Computing Center (TACC) at the University of Texas at Austin. Other institutions, such as the University of Hawaii (UH), have a hybrid deployment with subsets of Tapis API services deployed locally while leveraging others hosted at TACC. Tapis is multi-tenant, meaning that there can be a number of organizations using the same set of Tapis API services but persisting data in logically separate, secure, namespace. A tenant is typically a grouping of users, such as an institution, lab or project.

The goal of Tapis is to provide a simple, unified API that enables researchers to access computational and data intensive computing in a secure, scalable, and reproducible way allowing domain experts to focus on their research instead of the technology needed to accomplish it. Researchers and applications are able to interact with Tapis by making authenticated HTTP requests to Tapis's public endpoints. In response to requests, Tapis's network of microservices interact with a vast array of physical resources on behalf of users including high performance and high throughput computing clusters file servers and other storage systems, databases, bare metal, and virtual servers. Tapis aims to be the underlying cyberinfrastructure for a diverse set of research projects: from large scale science gateways built to serve entire communities, to smaller projects and individual labs wanting to automate one or more components of their process.

2.6 Motivation

Defining standards for Tapis APIs is required for the internal project development to present a unified and logical API experience for Tapis API consumers. However, as long as API responses are articulated properly, there can be flexibility in the implementation. To minimize the amount of initial work need to on-board developers and bootstrap the development of a new Tapis API internally, a base package was necessary. Here we present the Tapis team's best practices in developing Python based APIs for Tapis and an example implementation of the new Streams API using these practices.

3 API DEVELOPMENT FOR TAPIS IN PYTHON

In this section, we will discuss the design and implementation of a Tapis API implemented in Python. We start the discussion by describing various components required for the API implementation followed by a deep dive into the Streams API services design, data management, and hosting of the Streams service in a distributed manner and conclude with details on deploying the service with Kubernetes.

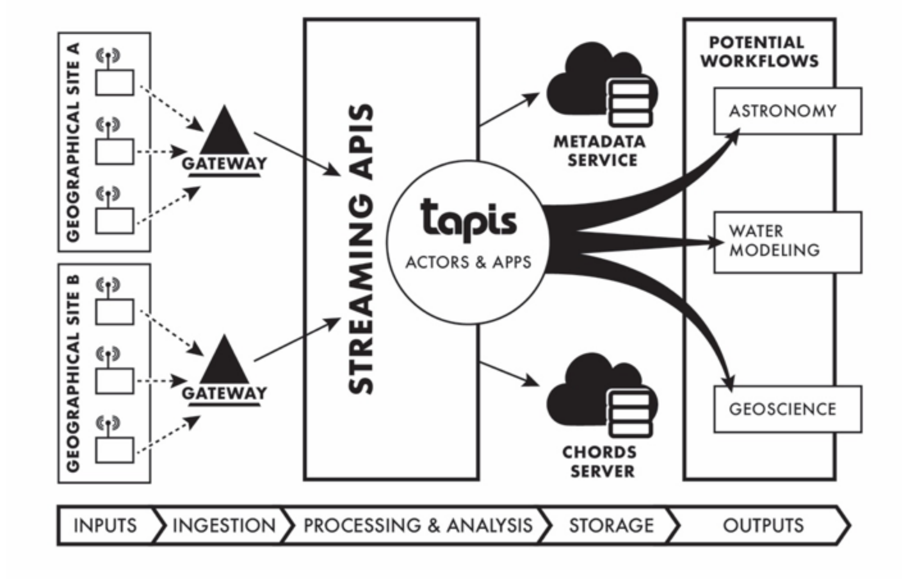

Fig. 1. Tapis workflow diagram illustrating support for actor and application integration streaming data workflows.

3.1 Components

3.1.1 Web Framework - Python Flaskbase. The Tapis Flaskbase [12] is a Python library maintained by the Tapis team. This library provides Python modules for integrating Tapis, Dockerfiles, and build scripts to standardize and streamline the development process for scientific APIs. This library makes building APIs faster, easier, more secure, and provides a way to update all core functionality across APIs that use it. This base library is opinionated in that it uses Python Flask [12] as the web framework. Flask was chosen for being light-weight with few dependencies and has been tested as dependable as part of the Abaco project [20]. The base library also incorporates common functions for leveraging Tapis Token and Authentication and Authorization services that are required for each API endpoint call. Additionally, appropriate organization of code folders, files, and database migrations is also provided to enable easier maintenance and help across the Tapis community. This base package also utilizes docker to provide a central Tapis Python API base image [13]. The use of a base image enables updating the core Tapis Python libraries from the development team as simple as rebuilding the docker container.

3.1.2 OpenAPI Specification. In building APIs and microservices a current best practice is to define a specification describing the service. Within the Tapis development group the OpenAPI 3.0 Specification (OAS) has been adopted [8]. It defines a standard, language-agnostic interface to RESTful APIs, which allows both humans and computers to discover and understand the capabilities of the service without access to source code or documentation. When properly defined, a consumer can understand and interact with the remote service with a minimal amount of implementation logic. API specifications can be written either in YAML or JSON format and include details such as, API endpoints, operations, input parameters, response codes, etc.

A key feature of the OpenAPI definition is that is can be used by a number of third-party tools to automate various tasks, including: documentation generation tools to automatically create interactive websites documenting the API, code generation tools to generate servers and clients in various programming languages, as well as testing tools. Using OAS for Tapis APIs also allow developers to leverage Python tools like the dynamic Tapis API Software Development Kit (SDK), DynaTapy developed by the Tapis team to dynamically generate and validate some of the API logic code. So not only does this process make the API more readable and accessible it can also ensure the endpoints are compliant with what the API documentation, it decreases the amount of coding and documentation effort overall. All Tapis APIs leverage the OAS to automatically generate and deploy interactive documentation as part of the continuous integration process, ensuring uniform and up-to-date information is maintained across the platform.

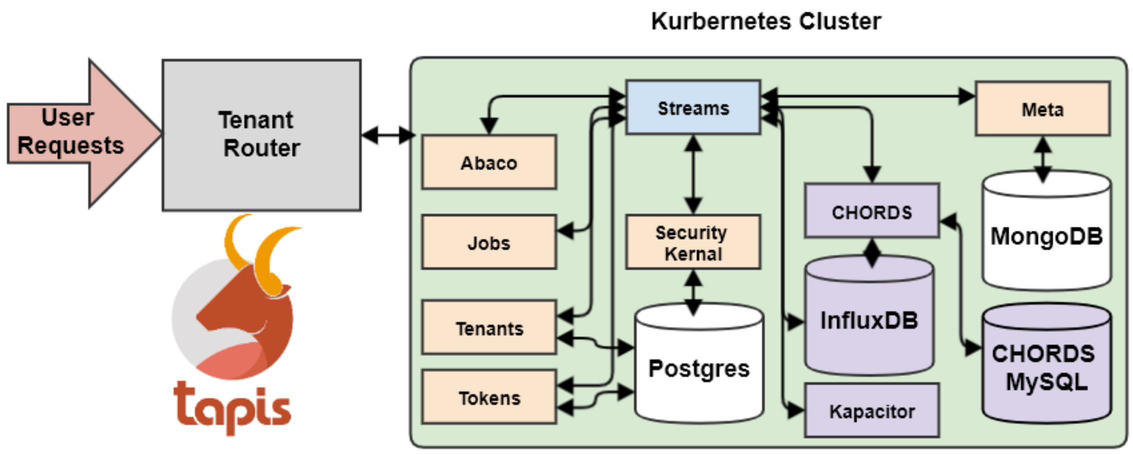

Fig. 2. Tapis Streams API diagram - Tapis tenant router(gray) routes all user requests to the appropriate API in the Kubernetes cluster(green). The Streams API service (blue) interacts with other Tapis APIs in orange and the CHORDS service (purple). Persistence for the Stream API data is stored in influxDB with metadata in MongoDB.

3.1.3 Authentication and Authorization. The Tapis authentication subsystem provides two mandatory components together with a set of optional services to enable institutions to integrate their existing identity providers and related systems. Tapis authentication is closely related to the notion of *tenancy*; a *tenant* in the Tapis framework represents a logical separation of Tapis entities (i.e., apps, jobs, actors, etc.) as well as a high-level authentication and security configuration. Signed JSON Web Tokens (JWT, [1]) represent the sole mechanism for proving identity to any Tapis API. As each tenant is configured with its own public key for token signatures, the entire authentication system can be customized on a per-tenant basis.

The Tapis authentication subsystem is comprised of the four primary components namely, Tokens API, Authentication Server, Tenants API and API Router. The Tokens API is a stateless mircoservice, which provides a reference implementation for generating a well-formatted, signed JWT. The Authentication server is OAuth2 and OIDC complaint web server, which is capable of authenticating and generating signed JWT's using the Tokens service for end users. It is capable of integrating with an institution's LDAP server. The Tenant's API is used by system administrators to manage the registry of tenants distributed globally in a given Tapis installation, and by Tapis services for various operational tasks. These tenant specific properties, such as the public key that is used to sign JWTs, the locations of other Tapis services, etc., can be extracted from public endpoints provided by the API. Finally, the API Router which acts as a load balancer and "'edge" router and can route requests to back-end services distributed across sites.

The Tapis Security Kernel plays an important role in the authentication and authorization process and is a distributed subsystem comprised of open-source software components tied together by a unifying API. The Tapis Security Kernel builds upon Shiro's [4] security framework to create a scalable, fine-grained authorization facility by extending its permissions model with representations of file path names. It has a secrets store, Vault [6], which manages all secrets in the system, including all passwords and keys, using a fault-tolerant, scalable, highly available cluster of VMs. The security kernel provides an API that the platform's other microservices use to securely interact with users' storage and execution resources and with each other.

The security kernel's API presents a unified interface to secrets and permissions management. It provides a scalable solution to the problem of fine-grained authorization checking across a virtually unlimited, distributed namespace. Access to secrets is possible by a microservice if the user on whose behalf the microservice is making a request is authorized by Shiro. Users can achieve fine-grained control over multitude of resources such as files, apps, projects, etc. by defining specific roles and permissions in the security kernel.

3.2 The Streams API - a Tapis API implemented in Python

3.2.1 Streaming Data Support Use Case. The explosion of IoT devices and sensors in recent years has lead to a demand for support of storing, processing, and analyzing time-series data. Further, as more ML and AI systems are developed and come online the need for re-enforcement of data and datasets that can be used for training as well as input for driving events is increasing. A number of technologies have sprung up for storing large volumes of logging information (Elasticsearch, Splunk, Apache Flume) and process data streams (Apache Kafka [3], Apache Storm [5], Apache Flink [2]). Geoscience researchers have been using time-series data stores such as the Hydroserver [22], Virtual Observatory and Ecological Informatics System (VOEIS) [18], and the Cloud-Hosted Real-time Data Service (CHORDS) [17]. Many of these tools can be powerful but also require a great deal of infrastructure overhead to deploy and expertise to manage and scale. To support streaming data for science, UH and TACC developed APIs and infrastructure, the Streams API [11], to enable the integration of streaming data workflows, storage, and retrieval with temporal and spatial support (Figure 1).

The Tapis Streams API utilizes the components described in the previous sections in its design. Specifically, openAPI 3.0 specification has been used to design REST API. The specification document is available in the Tapis Streams API github repository [14]. A livedocs based on the openAPI specification is also available for easy viewing of all the REST end-points [15]. We started the implementation of the API conforming to the openAPI specification. The Tapis Streams micro-service leverages other Tapis micro-services towards providing a secure, scalable, and reproducible service. It uses security kernel to check if the users or the devices are authorized and authenticated for the requested action, tokens service to obtain signed service-JWT to authorize itself to security kernel and metadata service for micro-service to micro-service communications, and metadata service as the back-end database to store metadata about the different Streams service resources such as projects, sites, instruments and variables. The Tapis Streams API also integrates CHORDS [17] streaming data services into Tapis to leverage its streaming data model and streaming data capability. CHORDS uses the time-series influxDB database to store the measurement values obtained for the variables. The Streams API service utilizes an administrative account, alternatively called a service account, to communicate with the CHORDS service, which acts as the service account for all metadata and data. Figure 3 shows the architecture of the

Tapis streams API. The segregation of metadata and data across Tapis tenants and users is enabled by using the Tapis Security Kernel service permissions to restrict access, allowing a single CHORDS service to serve multiple tenants.

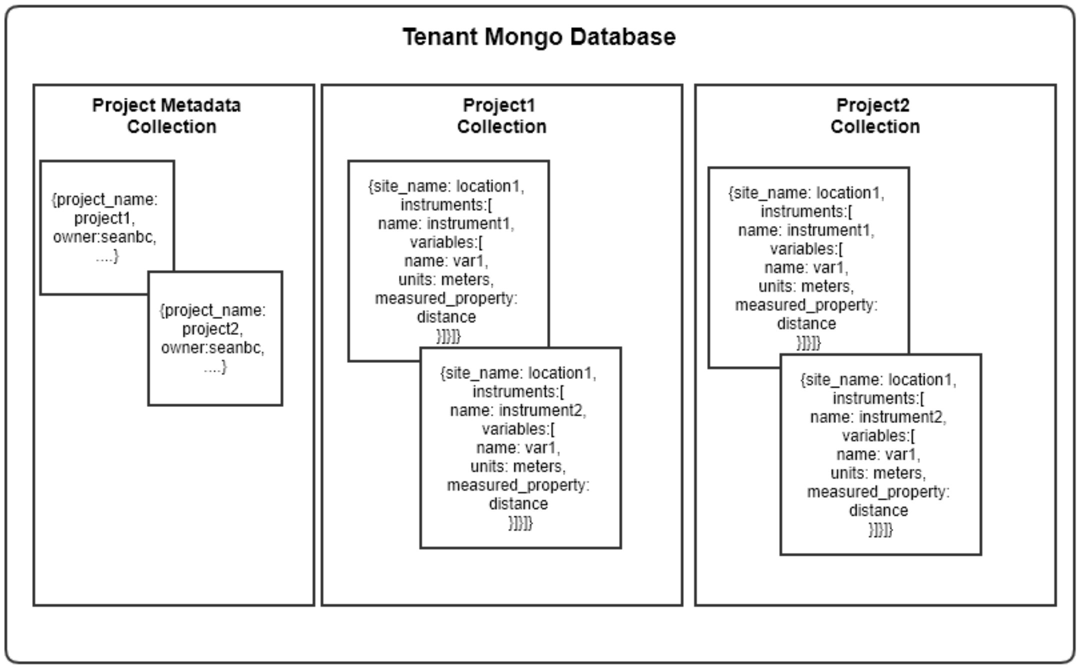
Tenant Mongo Database
Project Metadata Collection
{project_name: project1, owner:seanbc, ...}
{project_name: project2, owner:seanbc, ...}
Project1 Collection
{site_name: location1, instruments:[name: instrument1, variables:[name: var1, units: meters, measured_property: distance }]}]}
{site_name: location1, instruments:[name: instrument2, variables:[name: var1, units: meters, measured_property: distance }]}]}
Project2 Collection
{site_name: location1, instruments:[name: instrument1, variables:[name: var1, units: meters, measured_property: distance }]}]}
{site_name: location1, instruments:[name: instrument2, variables:[name: var1, units: meters, measured_property: distance }]}]}

Fig. 3. Stream API MongodDB Data Model

3.2.2 Tapis Metadata Integration. The Tapis Meta API service utilizes MongoDB as the database for storing metadata JSON documents in a MongoDB. The Streams API stores JSON embedded documents that encapsulate the CHORDS concepts of sites, instruments, and variables for describing where and how measurement data is captured and is organized within a project collection. The Tapis Meta 3.0.0 service is integrated with the RestHeart (RH) service [9], which offers a REST API to access and manipulate a MongoDB hosted database. Customizing the plugin provided with RH server allows security integration with the Tapis security kernel for authorizations of access to JSON data documents.

The Streams API currently utilizes permissions at a project level and manages users' actions based on their granted roles stored in the Tapis security kernel. The Stream API "project" concept is implemented where metadata is stored across two collections. One collection contains metadata about the project, and the second is a collection to hold the projects embedded document objects that hold information about sites, instruments, and variables. Upon the creation of a project, a new document is created in the metadata collection, and a new collection is created for that project's sites, instruments, and variable metadata (Figure 3).

The Meta MongoDB offers geospatial query support for data stored as GeoJSON objects, which the Tapis team has leveraged in other projects. Leveraging these capabilities simply require adding a 2dsphere index on the project

Table 1. Streams API endpoints

Resource Name	Method & Endpoint	Description
Project	GET, POST, PUT, DEL streams/v3/projects	Logical grouping of sites
Site	GET, POST, PUT, DEL streams/v3/sites	Geographical location of instruments
Instrument	GET, POST, PUT, DEL streams/v3/instruments	A source of related measurements
Variable	GET, POST, PUT, DEL streams/v2/variables	A measurement made by an instrument
Measurement	GET, POST, PUT, DEL streams/v3/measurements	A single observation of a variable
Channel	GET, POST, PUT, DEL streams/v3/channels	Pre-processing of measurements to trigger alerts

collection for an established schema field; this enables MongoDB to support complex geospatial queries across all JSON documents that contain the field in the collection. The Stream API adds these indices in response to creating a new project, and it creates a new collection for that project.

3.2.3 Local site implementation. Deploying the Streams API service requires an existing Tapis platform deployment to utilize, which does not have to be local, and a CHORDS service stack (CHORDS web application, MYSQL, Kapacitor and INFLUXDB). The UH and TACC implementation have deployed the CHORDS stack and Streams API as docker containers. The UH and TACC implementations have the CHORDS services configured as only accessible to the Streams API service to protect the data, this allows security to remain at the Tapis level when the Streams API service is configured as part of the Tapis API management system, which sits in front of all the Tapis APIs.

3.2.4 Deployment. In a production Tapis instance, the Streams service is packaged into a Docker image and deployed using Kubernetes, a popular container orchestration technology. Kubernetes supports important features including: service discovery and networking between different services in the same namespace, persistent data storage via Ceph-backed persistent volume claims (PVC), easy startup and tear down of the services, and basic service health monitoring capability. Applications are deployed in "Pods", which can be easily created or destroyed. Pods are defined similarly to Docker Compose files and codify the relationships between apps, arguments, storage, etc. "Services" control access to Pods both from inside and outside the cluster. Services can remain up even if the underlying Pods are down. Each service has an IP address and a port, which does not change, thereby allowing the clients (or a proxy server) to reach Pods without knowing their individual addresses. Kubernetes treats the Pods as transient and will restart them if they crash. For the Streams API, we have five Kubernetes applications each running in their own Pod: Streams API, CHORDS, InfluxDB, Kapacitor and MySQL. Configuration is provided to the Pods via "ConfigMap" objects, which contain files that can be mounted into Pod containers. All Kubernetes objects are described in YAML files. All of these services have specific ports associated with them. For example CHORDS app is accessible on port 80, InfluxDB on port 8086, Kapacitor on 9092 and Chords-MySQL on 3306. These services have been deployed in Tapis dev environment. Users subscribed to Streams API and having specific permissions on projects can access and modify the resources, utilizing these underlying services. Easy set up and tear down of these services is possible with the help of automated scripts developed by the Tapis team.

4 CONCLUSION

The Tapis Python API design provides a foundation for fast development of web APIs that integrate with the Tapis Framework. The best practices presented by the Tapis development team illustrate a successful and secure method for integrating and scaling microservices to support reproducible scientific research.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation Office of Advanced CyberInfrastructure[grant numbers 1931439 and 1931575], RII Track 1: ʻIke Wai Securing Hawaiʻi's Water Future NSF OIA #1557349, Collaborative Research: SS2-SSI: The Agave Platform: An Open Science-As-A-Service Cloud Platform for Reproducible Science NSF ACI #145041

REFERENCES

[1] 2018. *JSON Web Token (JWT).* Retrieved Oct 30, 2019 from https://tools.ietf.org/html/rfc7519
[2] 2019. *Apache Flink.* Retrieved Feb 17, 2020 from https://flink.apache.org
[3] 2019. *Apache Kafka.* Retrieved Feb 17, 2020 from https://kafka.apache.org
[4] 2019. *Apache Shiro.* Retrieved Oct 30, 2019 from http://shiro.apache.org/
[5] 2019. *Apache Storm.* Retrieved Feb 17, 2020 from http://storm.apache.org
[6] 2019. *HashiCorp Vault.* Retrieved Oct 30, 2019 from https://www.vaultproject.io/
[7] 2019. *Introduction to JSON and Restful APIs(coding bootcamp series).* Retrieved Feb 11, 2020 from https://blog.usejournal.com/introduction-to-json-and-restful-apis-coding-bootcamp-series-7ad6aa294c89
[8] 2019. *OpenAPI specification 3.0.* Retrieved Feb 12, 2020 from https://github.com/OAI/OpenAPI-Specification
[9] 2019. *RestHeart API.* Retrieved Feb 17, 2020 from https://restheart.org
[10] 2019. *Tapis.* Retrieved Feb 17, 2020 from https://tapis-project.org/
[11] 2019. *Tapis-CHORDS Integration: Time-Series Data Support in Science Gateway Infrastructure.*
[12] 2019. *Tapis Flaskbase.* Retrieved Feb 17, 2020 from https://github.com/tapis-project/flaskbase
[13] 2019. *Tapis Flaskbase Docker Container.* Retrieved Feb 17, 2020 from https://hub.docker.com/r/tapis/flaskbase
[14] 2019. *Tapis Streams API.* Retrieved May 9, 2020 from https://github.com/tapis-project/streams-api
[15] 2020. *Tapis Live Docs.* Retrieved Feb 17, 2020 from https://tapis-project.github.io/live-docs/
[16] Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. Dissertation. University of California, Irvine.
[17] Branko Kerkez et al. 2016. Cloud Hosted Real-time Data Services for the Geosciences (CHORDS). Geoscience Data Journal, 2–4.
[18] S.J.K. Mason et al. 2014. A centralized tool for managing, archiving, and serving point-in-time data in ecological research laboratories. *Environmental Modeling & Software* 51, 59–69.
[19] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241
[20] Joe Stubbs et al. 2018. Rapid Development of Scalable, Distributed Computation with Abaco. Science Gateways Community Institute, 10th International Workshop on Science Gateways.
[21] Joe Stubbs, Richard Cardone, Mike Packard, Anagha Jamthe, Smruti Padhy, Steve Terry, Julia Looney, Joseph Meiring, Steve Black, Maytal Dahan, Sean Cleveland, and Gwen Jacobs. 2020. Tapis: An API Platform for Reproducible, Distributed Computational Research. (2020). submitted.
[22] D.G. Tarboton et al. 2009. Development of a community hydrologic information system. In *Proceedings of 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation, Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation).* 988–994.