

Microservice Planning and Specifications

This chapter contains all specification documents for home made microservices

- [Base Microservice Specifications](#)
- [DevOps Topics](#)
- [Security Goals](#)
- [Infrastructure](#)
- [Cloud Configurations Repository Wiki](#)
- [Logging and Tracing](#)

Base Microservice Specifications

This document will describe basic functions all Microservices will have. The service's endpoints and controller files will be generated using OpenAPI's generator. As such, all microservices will follow OpenAPI's specification for code generation and documentation.

Software Topics

Base Image

The official [Debian](#) docker image will be used for all containers since the dev container will also be a Debian based image. The latest Java version will also be used for running the microservices

Versioning

To distinguish when containers were built, each container will have a buildinfo.json text file generated.

```
{
  "timestamp": "2021-10-16T14:45:51.208015",
  "base-version": "<insert base version here>",
  "application-version": "<insert application version here>"
}
```

In the keys "base-version" and "application-version", developers will be required to follow the semantic versioning method, a sequence versioning method that follows a vX.Y.Z format (eg: v1.2.3), where X is a major change, Y is a minor change and Z is a patch change. Additional information can be added on the end of the version number to distinguish between test builds, nightly builds, ect. Example: v1.1.2-20211209Nightly.

Logging

In effort to centralize logging, all containers must log to the standard out of the docker-compose process. (This shouldn't be an issue since if you just log everything to STD out, it'll log to the docker logs). One consideration to make is that logs will be sent to a centralized logging server so they should be easily grep-able.

Event Messaging

Microservices will have their own event messaging topics that other microservices can subscribe to. For instance, an authentication microservice might have a topic for notifying other services when tokens have expired.

Database/caching

Each microservice will have its own cache. There will be a group of other services that provide database functionality.

Well Known Endpoints

/health

Can return HTTP status 200 for healthy or 500 for not healthy.

This will not be available to end users

/docs

HTTP documentation generated with OpenAPI. **This will not be available to end users**

/build

Information about the application.

/operations

Returns information about running operations.

Consider the following, client POSTs a create endpoint to create an ambiguous object that take a long time to construct. The client will receive the following response if the operation is created:

```
202 Accepted

{
  "operationLocation": "https://example.com/operations/<operation-id>"
}
```

The client can then invoke a GET request on the operationLocation to get the status of the operation. If the server responds that the operation isn't ready, a Retry-After HTTP header will be sent.

```
200 OK
Retry-After: 30

{
  "createdDateTime": "2022-09-20T15:00:00.00Z",
  "status": "running",
  "estimatedCompleteTime": "2022-09-20T15:01:00.00Z"
}
```

```
}
```

```
200 OK
```

```
Retry-After: 30
```

```
{  
  "createdDateTime": "2015-06-19T12-01-03.4Z",  
  "status": "running",  
  "percentComplete": 0,  
}
```

When the request is complete, the operation will respond with the resource's location and other data

```
HTTP 200
```

```
{  
  "createdDateTime": "2022-09-20T15:00:00.00Z",  
  "lastActionDateTime": "2022-09-20T15:01:00.00Z",  
  "status": "succeeded",  
  "resourceLocation": "https://example.com/<some-resource>/<some-resource-id>"  
}
```

API Guidelines

All microservices will be required to follow the guidelines as closely as possible. This is to ensure stability of the ecosystem and to make developing client applications easier. The API guidelines will be a combination of the following existing guidelines:

- [Microsoft's API Guidelines](#)

Errors and Faults

An error occurs when the client passes invalid data to the server. Typically, errors are 4XX errors. When an error occurs, the service should be able to operate normally and not shut down.

A fault occurs when the service can not pass the correct response to the clients. Typically, faults are 5XX HTTP errors. Services should attempt to recover from a fault in order to avoid shutting down, however if they can not recover, the health of the service should be labeled as Unhealthy so it can be troubleshooted/restarted.

URL Structures

URLs should be easy to read by people. This includes the maximum length of a URL. While RFC7230 does not a specific URL length, the length of a URL should not exceed 2,083 characters (This is the maximum length for Internet Explorer).

Required Supported Methods

Services should support the required methods with no exception: GET, POST, DELETE, PUT, HEAD, OPTIONS. If Open API is used, all of these methods will be available for use. Below summarizes a table of the behavior of the methods on each resource from an ecommerce point of view:

Resource	POST	GET	PUT	DELETE
/customers	Create a new customer	Retrieve all customers	Bulk update of customers	Remove all customers
/customers/1	Error	Retrieve the details for customer 1	Update the details of customer 1 if it exists	Remove customer 1
/customers/1/orders	Create a new order for customer 1	Retrieve all orders for customer 1	Bulk update of orders for customer 1	Remove all orders for customer 1

POST Method

POST should be used for creating resources. A POST request should always return HTTP 201 (Created) with the new location of the object.

For example, consider an API for managing users, the client can POST data a URL to create users:

```
curl \
  -X POST \
  -d '{"user':'bob12345'}" \
  -H "Content-Type: application/json"
```

Should return

```
201 Created

{
  "location":"http://example.com/users/bob12345"
}
```

Additionally, the server may return the entire metadata for the object. The metadata object should have a "self" key to dictate the path to the object.

201 Created

```
{
  "username": "bob",
  "self": "http://example.com/users/bob",
  "creation_date": "2022-09-20T15:00:00.00Z"
}
```

Required Request Headers

The following request headers should be sent with every request:

Authorization: This allows the server to authenticate the request. (Note some services can leave this out if they have public facing data)

Date: The date the request was made. The date should follow ISO8601 and should always include a timezone. You can just use Z if using UTC timezone. The server should never assume the clients clock is accurate. If the time is vastly different than the servers time (+ or - 1 minute) the server should throw 403 with a reason of the clock being out of date.

Accept: Allows the server to see what type of content the client accepts

Accept-Encoding: Allows the server to see what encoding the client accepts

Accept-Charset: Allows the server to see what charset the client accepts (This will probably always be UTF-8)

Content-Type: Only used for POST or PUT calls, allows the server to handle incoming data

Prefer: Allows a client to change how much data the server returns. Can either be "return=minimal" or "return=representation"

Required Response Headers

Date: The date the request was sent to the user.

Content-Type: Content type of the response

Content-Encoding: Encoding the server encoded the response to

Optional Response Headers

Retry-After: Tells the client to retry the request after a date in time. This will primarily be used with long running operations. Note that while the HTTP specification states that this can return a number of seconds, for the purpose of the API, only the date will be used. The date should be formatted to fit ISO 8601 standard (eg: YYYY-MM-DDTHH:MM:SSSZ format)

Operation-Location: The location of the new operation. The location will be a link

Custom Headers

Custom headers can not be required for a service to work. All services should work within the required headers above. Custom headers should in a generic format or a scoped format. Custom header data should never be JSON objects

Using Query Parameters

Personally identifiable information should never be used in a URLs query parameters. Additionally query parameters should avoid using json objects.

Handling Errors

When an error occurs, the server should return some sort of message. At minimal the error object should contain a "code" key and a "message" key.

```
{
  "code": "someCode",
  "message": "some message"
}
```

More complex errors can contain the "target", "details" and "innererror" keys like so:

```
{
  "code": "someCode",
  "message": "some message",
  "target": "some target",
  "details": ["some", "details"],
  "innererror": {
    "some complex json object here"
  }
}
```

An example of a more complex error object is:

```
{
  "error": {
    "code": "BadArgument",
    "message": "Previous passwords may not be reused",
    "target": "password",
    "innererror": {
```

```
"code": "PasswordError",
"innererror": {
  "code": "PasswordDoesNotMeetPolicy",
  "minLength": "6",
  "maxLength": "64",
  "characterTypes": ["lowerCase", "upperCase", "number", "symbol"],
  "minDistinctCharacterTypes": "2",
  "innererror": {
    "code": "PasswordReuseNotAllowed"
  }
}
}
```

Special Note on Clients

When services are designed, they should be designed to be effortlessly used by simple HTTP tools such as curl or postman.

Collections

An inevitable topic that will come up during designing will be collections. A collection is just a simple group of similar objects. For example, a group of 'user' objects are 'users', a group of 'file' objects are 'files'. This should be used when designing endpoints.

For example, if you want to get a list of users, you might want to do the following:

```
curl \
  -X GET \
  -H "Content-Type: application/json" \
  https://example.com/users
```

If you want to get a singular user, you would do:

```
curl \
  -X GET \
  -H "Content-Type: application/json" \
  https://example.com/users/<user-id>
```

For querying and filtering the collection, URL headers should be used. Take care to not expose personal identifiable information!

```
curl \  
-X GET \  
-H "Content-Type: application/json" \  
https://example.com/users?$orderBy=name asc
```

```
curl \  
-X GET \  
-H "Content-Type: application/json" \  
https://example.com/users?$filter=name eq 'kyle'&orderBy=createdDate dec
```

Collections will always return a response with a "value" key like so:

```
{  
  "value": [  
    . . .  
  ]  
}
```

If no data is found, the HTTP 204 status code should be used

Nested Collections

Nested collections should also be designed around. For example, a user might have multiple friends:

```
curl \  
-X GET \  
-H "Content-Type: application/json" \  
https://example.com/users/<user-id>/friends
```

Nested collections will have to abide by the "value" key rule only if they are retrieved via their nested URL. Eg:

```
curl \  
-X GET \  
-H "Content-Type: application/json" \  
https://example.com/users/BillyTheKid05
```

Will return

```
{  
  "value": {
```

```
{
  "username": "BillyTheKid05",
  "firstName": "Billy",
  "lastName": "Smith",
  "friends": [
    {
      "username": "JohnApple",
      "firstName": "John",
      "lastName": "Appleseed"
    },
    {
      "username": "PaulB123",
      "firstName": "Paul",
      "lastName": "Bunyon"
    },
    ...
  ]
}
```

while

```
curl \
-X GET \
-H "Content-Type: application/json" \
https://example.com/users/BillyTheKid05/friends
```

Will return

```
{
  "value": [
    {
      "username": "JohnApple",
      "firstName": "John",
      "lastName": "Appleseed"
    },
    {
      "username": "PaulB123",
      "firstName": "Paul",
      "lastName": "Bunyon"
    }
  ]
}
```

```

    ....
  ]
}

```

Big Collections and Pagination

Pagination should be used on all results by default. Pagination parameters should be URL query parameters. paginated responses should have a "@nextlink" key that indicates the next link for the user or client to follow. The URL query parameters "skip" and "limit" should be used:

```

{
  "value": [
    . . .
  ]
  "@nextlink": "https://example.com/collection?$orderBy=name asc&skip=10&limit=10"
}

```

If skip either or limit are missing, an HTTP 401 should be sent

Filter Operations

At minimum, the filter should support the following operations:

Operator	Description	Example
Comparison Operators		
eq	Equal	city eq 'Redmond'
ne	Not equal	city ne 'London'
gt	Greater than	price gt 20
ge	Greater than or equal	price ge 10
lt	Less than	price lt 20
le	Less than or equal	price le 100
Logical Operators		
and	Logical and	price le 200 and price gt 3.5
or	Logical or	price le 3.5 or price gt 200
not	Logical negation	not price le 3.5
Grouping Operators		
()	Precedence grouping	(priority eq 1 or city eq 'Redmond') and price gt 100

Dates, Times and Intervals

As alluded to above, all dates will follow the ISO 8601 standard. If the date is in UTC time, there should be nothing after the Z

A standard date will look like `YYYY-MM-DD`

A standard date with time will look like `YYYY-MM-DDTHH:MM.SSSZ`

A duration will follow the format: `P[n]Y[n]M[n]DT[n]H[n]M[n]S` where

- P is the duration designator (historically called "period") placed at the start of the duration representation.
- Y is the year designator that follows the value for the number of years.
- M is the month designator that follows the value for the number of months.
- W is the week designator that follows the value for the number of weeks.
- D is the day designator that follows the value for the number of days.
- T is the time designator that precedes the time components of the representation.
- H is the hour designator that follows the value for the number of hours.
- M is the minute designator that follows the value for the number of minutes.
- S is the second designator that follows the value for the number of seconds

Long Running Operations

It is important that users are able to monitor operations that run for a long period of time. When a long run operation is invoked, the client will be able to poll the operation to get an idea of where the task is at. All operations will be given a task id and will be able to be polled at the /operations endpoint.

Returning Specific Fields

Some clients might not need the entire object that is given. To save on bandwidth, services can implement a fields URL parameter to specify the fields they need. For example, consider a user model:

```
{
  "id": "0",
  "firstName": "Bob",
  "lastName": "Smith",
  "birthday": "20220101",
  "phone": "(999) 999-9999"
}
```

Let's say you want just the first and last names of the user. You can specify that in the fields URL parameter:

```
curl \  
  -X GET \  
  -H "Content-Type: application/json" \  
  https://example.com/users/0?fields=firstName,lastName
```

This will return

```
{  
  "firstName": "Bob",  
  "lastName": "Smith"  
}
```

Likewise, on a collection:

```
curl \  
  -X GET \  
  -H "Content-Type: application/json" \  
  https://example.com/users?fields=firstName,lastName
```

```
{  
  "value": [  
    {  
      "firstName": "Bob",  
      "lastName": "Smith"  
    },  
    {  
      "firstName": "Sue",  
      "lastName": "Baker"  
    },  
    . . . . .  
  ]  
}
```

File System

Each microservice container will follow Linux's [Filesystem Hierarchy Standard](#) and will have a common file structure to simplify everything

High Level directories

Below are the important high-level directories.

Folder	Description	Mappable
<code>/opt/HeestandTech</code>	Application binaries and startup scripts	No
<code>/etc/HeestandTech</code>	Application configuration files	Yes
<code>/var/opt/HeestandTech</code>	Variable data from microservice (temporary upload files, mock data, ect)	Yes
<code>/var/cache/HeestandTech</code>	Locally generated data from long running I/O or a calculation. This folder will be able to be deleted with no negative effects to the program	No
<code>/var/log/HeestandTech</code>	Log file locations	Yes
<code>/var/lib/HeestandTech</code>	Persistent application data	Yes

DevOps Topics

Overview

The purpose of this document is to specify different devops topics from building, testing and deploying to developing in a container with docker.

Visual Studio Code

VS Code will be used for all major development. For the most up to date IDE, it is encouraged that users develop on the cloud development machine. The Remote-Containers and Remote-SSH extensions must be installed. Any additional extensions will be automatically installed when setting up the remote environment

Setup

- On the local machine with VS Code, you will need to generate a keypair for Gitea

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

- Be sure to add it to the agent with

```
eval "$(ssh-agent -s)"
```

```
ssh-add <path-to-ssh-key>
```

Once your key has been added to Gitea, you can develop either on a remote machine or your local machine.

On same local machine

- On the local machine with VS Code, generate another keypair to put on the remote machine

```
ssh-keygen -t ed25519 -C "your_email@example.com"
```

- On the local machine add the keypair to your ssh agent

```
eval "$(ssh-agent -s)"
```

```
ssh-add <path-to-ssh-key>
```

- On the remote machine, you can put the **public key** in your `~/.authorized_keys` file
- Open the remote location in VS Code
- You should be prompted in the bottom left corner to reopen the project in a container

On a remote machine

- Make sure docker is installed and the docker plugin is installed in VS Code
- Open the folder in VS Code
- You should be prompted in the bottom left corner to reopen the project in a container

Starting a new project

To start a new project, you will need to make a template repository with the Hello World Microservice repository. This can easily be done with Gitea by adding

[kheestand/HelloWorldMicroservice](#) as a template repository when creating the repository.

Once you have the HelloWorldMicroservice added as a base repository, you will need to modify the .env file located at /workspace/.openapi/.env. Your .env file should look something like this:

```
{  
  "schemaLink" : "<link to openapi schema>"  
}
```

Note: you will need a link to a valid openapi schema. You can use the [pet store api](#) if you just want to test these instructions.

Once your .env file is ready, you can select the **Start new Project** task in VS code by pressing Ctrl + Shift + P on your keyboard and selecting *Tasks: Run Tasks* and then *Start New Project*.

When the project is generated you should also edit the name of the devcontainer environment to something other than "Microservice v1.0.0.0". This can be done by navigating to the /workspace/.devcontainer/devcontainer.json file and editing the "name" field at the top of the file.

Updating the template repository

In VS code you can use the following commands to update the template

```
git fetch --all  
git merge template/master
```

After running git fetch you might need to commit the changes to finish the merge. This can be done through VS Code's Source Control menu. You also might need to allow unrelated histories which you can do by modifying the git merge command to

Viewing Schema in SwaggerUI

The [OpenAPI Preview](#) extension should be installed when the devcontainer is set up. If it is, go to the directory `/workspace/server/openapi_server/openapi/openapi.yaml` then press `Ctrl + Shift + P`. Then select or type in "OpenAPI Preview" from the menu

Run, Test, Build and Deploy

Running the Code for Debugging

Code can be ran in the following ways:

- Press F5
- Go to the Run and Debug menu (`Ctrl + Shift + D`) and select "Launch OpenAPI2SpringBoot"

Running Tests

In progress...

Building

Jenkins will be used for building all microservices. Each repo will have a dockerfile located in the root directory that will tell the system how to build the image. An example can be found on the HelloMicroservice gitea repo. A sample keystore.jks file is included in the `.crypto` folder and is used to sign the jar during development. The password for the keystore is test123

Note that microservices that are built in production will be signed using the jarsigner utility with the production keystore.

Deploying

In progress...

What's included in the Dev Container?

Docker-compose environment

MariaDB

Default username `root` and password `mariadb`

[Docker Hub Link](#)

VS Code Extensions

VSCoDe-MySQL-Client

[VS Code Store Link](#)

OpenApi-Preview

[VS Code Store Link](#)

Security Goals

Overview

The purpose of this document is to outline security goals for the entire system

Goals

There are multiple goals that need to be met to provide users with the most secure platform as possible. We will be defending against many common attacks such as:

- Session hijacking and forging
- Database leaks
- Attackers gain knowledge of the internal system
- Attackers gaining access to the internal system and stealing data

End to End Encryption

All data must be encrypted with some sort of encryption while in transport. The microservices must only accept data from HTTPS sources. Data that is going from the microservice to the database must be encrypted with TLS and data that is at rest or not in use will be encrypted. Messages from one microservice to another will also be encrypted with TLS.

Verifiable Data

Certain data should be signed by the user so other users can verify it was them. Some examples include messages, files, commits, ect.

Authentic Microservices

We should be able to verify microservices before they start that it was signed by a trusted user. Moreover, users should be able to see who has signed the microservice. This problem can be solved by using PGP keys. See the infrastructure page for more information on how it is set up

Cryptic but Useful Errors

Having general errors displayed to the user allows for people to see what went wrong and prevents bad actors from gaining inside knowledge about the system. For example, if a file can not be found we would not display:

```
Traceback (most recent call last):
File "C:\Users\sauris\Desktop\pyfile\folder-read.py", line 13, in <module>
File "C:\Users\sauris\AppData\Local\Programs\Python\Python36-32\lib\bz2.py", l
ine 96, in __init__
self._fp = _builtin_open(filename, mode)
FileNotFoundError: [Errno 2] No such file or directory: 'sampledatafile2.txt.bz2'
```

Instead, we would just display:

```
sampledatafile2.txt.bz2 was not found
```

Non-root Containers

All container/microservices should not be run as the root user. This is to prevent someone from getting access to the container and compromising it.

Post Quantum Encryption

Recently [NIST published](#) the chosen algorithms for post quantum cryptography! These will be worked in to the software as seen fit

Infrastructure

Overview

This page will outline the specifications for the infrastructure. It is very important that a strong infrastructure is used to facilitate better scalability, better developer experience and a better user experience. Most infrastructure will remain inaccessible to the general public. The infrastructure will be split up into different "stacks".

All infrastructure containers will be the official images or will be based off of the official docker images

Logging/Metrics

Logging

For logging, the GELF stack will be used (Graylog, elasticsearch, mongodb).

Metrics

For metric capturing the GIT stack will be used (Grafana, InfluxDB, Telegraf)

Identity/Authentication

OpenLDAP and Keycloak will be used for both authentication and authorization. Accounts will be created in OpenLDAP and not Keycloak in most cases

Database

Postgres will be the database of choice for all infrastructure services and application containers. Redis will also be use for caching

File storage

Minio will be used for storing and retrieving files. It is an S3 compliant file system

Cloud Configurations Repository Wiki

How to Contribute

Anyone is welcome to contribute to this project. Be sure to follow the guidelines below and get started!

Folder Structure

The following diagram displays how a developer should set up their API folder structure:

```
APIs/  
├─ API_NAME/  
│   ├─ master.yaml  
│   ├─ models/  
│   │   └─ someModel1.yaml  
│   └─ paths/  
│       └─ somePath1.yaml  
CommonFragments/  
CommonModels/
```

Note that master.yaml contains the info about the API, specific components and the path definitions. An example can be found below:

```
openapi: 3.0.3  
info:  
  title: "Sample API"  
  description: "Information about your API, what it does and how it works"  
  version: 0.0.0  
servers:  
  - url: https://development.service.example.com/  
    description: "Development server, breaking changes are often made here"  
  - url: https://staging.service.example.com/  
    description: "Staging server, breaking changes should rarely happen here"  
  - url: https://service.example.com/
```

```
    description: "Production server, breaking changes should never happen here"
paths:
  /path/resource:
    - $ref: ./paths/path-resource.yaml
  /path/something:
    - $ref: ./paths/path-something.yaml
components:
  someComponent ...
security:
  someSecurity ...
tags:
  someTags ...
externalDocs:
  description: Find more info here
  url: https://docs.example.com/someServiceDoc.html
```

Common Fragments

Common fragments are very small blobs of yaml code that can be shared amongst any services. For example, it is standard that a username has a minimum and maximum length. Therefore, we can create a fragment that we can reuse across all yaml files.

Common Models

Common models are yaml object files that contain code that is shared amongst any services. An example of this is the standard response one might get from the health and version api. Other examples are general HTTP error codes

Logging and Tracing

Purpose

The purpose of this document is to describe how microservices should log. As a note, this excludes infrastructure containers since they all have their own log formats. The goal of having a standard logging format for containers is to allow for easier debugging.

Log Levels

The following standard log levels will be used. A description of each level will also be provided

FATAL	Very sever errors that will lead to the application crashing.
ERROR	Events that could be fatal but also allow the application to continue running.
WARN	Events that could lead to potentially harmful situations.
INFO	Informational messages that highlight the current state of the application
DEBUG	Events that will be useful for a developer when debugging
TRACE	Events that provides finer detail than DEBUG. This could be a dump of a data structure, progress of a task ect.

Standard Logging Format

The standard log format for all services will be a JSON object:

```
{
  "timestamp": "2022-09-22T10:33:20.123456",
  "level": "FATAL|ERROR|WARN|INFO|DEBUG|TRACE",
  "service": "my-microservice",
  "session_id": "7cddeb94-4d9b-4231-bb5a-2c9ce889926c",
  "user_agent": "User agent here",
  "method": {
    "name": "someMethod",
    "args": ["some", "args"]
  }
}
```

```
    },
    "message": "some super cool message",
    "stacktrace": "the stacktrace here if there is an error"
  }
}
```

PII (Personally Identifiable Information)

Personally Identifiable Information shall be removed from all logs. Some PII includes:

- Username
- Passwords
- API Tokens
- API Keys
- Names
- Age
- Addresses
- Phones
- SSN
- Bank account numbers
- Credit card numbers
- Biometric data
- Geographic data
- Date of birth
- ect

If PII should be put in a log message it should be changed to have only 5 * characters. For instance, the full name "Kyle Heestand" would become "*****". This is to not only protect against the data in the database, but also to protect against people guessing who/what the data is based on the size of it. As an example, take a standard login message:

```
{
  "timestamp": "2022-09-22T10:33:20.123456",
  "level": "INFO",
  "service": "LoginService",
  "session_id": "7cddeb94-4d9b-4231-bb5a-2c9ce889926c",
  "user_agent": "User agent here",
  "method": {
    "name": "login",
    "args": ["kheestand", "some-password"]
  },
  "message": "Kyle Heestand has logged in"
}
```

After removing the PII, we will get something like this:

```
{
  "timestamp": "2022-09-22T10:33:20.123456",
  "level": "INFO",
  "service": "LoginService",
  "session_id": "7cddeb94-4d9b-4231-bb5a-2c9ce889926c",
  "user_agent": "User agent here",
  "method": {
    "name": "login",
    "args": ["*****", "*****"]
  },
  "message": "***** has logged in"
}
```

Obviously, you would more than likely have a different message, but we can see that the user's username, password and full name has been removed to preserve their privacy.