A picture containing drawing

Description automatically generatedA picture containing scene, laser, stage

Description automatically generated

Project Parivartan API and UI/UX Standards

NSE API Standards and Guidelines

The document outlines API standards, error codes, URL and resource structures, naming conventions for Classes that can be adhered to for consistency of API definitions across project teams and BUs.

**URL structure**

All endpoints must be prefixed with a version number /v1/. Pattern: /v1/...

Collections of resources are referenced by their resource name (plural)

Pattern: /v1/:resource name

Example: /v1/accounts

Individual resources are referenced their resource name (plural) followed by the uuid Pattern: /v1/:resource name/:uuid

Example: /v1/accounts/25fe21b8-8de2-40d0-93b0-c819101d1a11

**Resources**

A resource represents an individual object within the system, such as an account or a branch. It is represented as a JSON object.

A resource **MUST** contain the following fields:

uuid: a unique identifier for the resource

created\_at: an ISO8601 compatible date and time that the resource was created

updated\_at: an ISO8601 compatible date and time that the resource was last updated

A resource **may** contain additional fields which are the attributes describing the resource.

A resource **MAY** include a self link object in the links field.

**Example**

{

"uuid": "a-b-c",

"created\_at": "2015\*07\*06T23:22:56Z",

"updated\_at": "2015\*07\*08T23:22:56Z",

"name": "school account",

"description": "an example account",

"links": {

"self": {

"href": "/v1/accounts/a-b-c"

}

}

}

# Using HTTP Methods for RESTful Services

The following table summarizes the sample scenarios for using different types of HTTP Verbs

|  |  |  |  |
| --- | --- | --- | --- |
| HTTP Verb | CRUD | Entire Collection (e.g. /customers) | Specific Item (e.g. /customers/{id}) |
| POST | Create | 201 (Created), 'Location' header with link to /customers/{id} containing new ID. | 404 (Not Found), 409 (Conflict) if resource already exists.. |
| GET | Read | 200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists. | 200 (OK), single customer. 404 (Not Found), if ID not found or invalid. |
| PUT | Update/Replace | 405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| PATCH | Update/Modify | 405 (Method Not Allowed), unless you want to modify the collection itself. | 200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid. |
| DELETE | Delete | 405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable. | 200 (OK). 404 (Not Found), if ID not found or invalid. |

**HTTP Methods**

**GET**

Used to retrieve a single resource or a list of resources.

* GET requests **may** include query parameters
* GET requests **must NOT** include a request body

**Example**

Show individual resource:

GET /v1/accounts/:uuid

**Responses (Resource)**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Code** | **Body** |
| Authorized User | 200 | Resource |
| Resource is Missing | 404 | Error |

List collection of resources:

GET /v1/accounts/

**Responses (Collection)**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Code** | **Body** |
| User With Complete Visibility | 200 | List of All Resources |
| User With Partial Visibility | 200 | List of Visible Resources |
| User With No Visibility | 200 | Empty List |

**POST**

Used to create a resource.

* POST requests **must NOT** include query parameters
* POST requests **may** include a request body

**Examples**

Create a resource:

POST /v1/accounts/ {

"customer\_uuid": "ab09cd29-9420-f021-g20d-123431420768",

"account\_number": "12345678",

"brach\_uuid": "uuid-bd7369a8-deed-ff1a-2315-77410293a922",

...

}

**Responses**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Code(s)** | **Body** |
| Authorized User (sync) | 201 | Created Resource |
| Authorized User (async) | 202 | Empty w/ Location Header -> Job |
| Unauthorized User | 401 | Error |
| Forbidden User | 403 | Error |
| Read-only User | 405 | Error |

**PUT**

Used to trigger an action, basically to update a resource

* PUT requests **must NOT** include query parameters
* PUT requests **may** include a request body
* PUT requests **MUST be** idempotent

**Examples**

Partially update a resource:

PUT /v1/accounts/:uuid

{

"alias": "new\_alias\_name"

}

**Responses**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Code(s)** | **Body** |
| Authorized User (sync) | 200 | Empty |
| Authorized User (async) | 202 | Empty w/ Location Header -> Job |
| Unauthorized User | 401 | Error |
| Forbidden User | 403 | Error |
| Resource is Missing | 404 | Error |
| Read-only User | 405 | Error |

**DELETE**

Used to delete a resource.

* DELETE requests **may** include query parameters
* DELETE requests **must NOT** include a request body

**Examples**

Delete a resource:

DELETE /v1/accounts/:uuid

**Responses**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Code(s)** | **Body** |
| Authorized User (sync) | 204 | N/A |
| Authorized User (async) | 202 | Empty w/ Location Header -> Job |
| Unauthorized User | 401 | Error |
| Forbidden User | 403 | Error |
| Resource is Missing | 404 | Error |
| Read-only User | 405 | Error |

**Naming Conventions**

In the world of APIs, there are URIs, resources, and representations. A URI is what allows you to access a resource. What is returned, in response, is a representation of that resource. Since we’re a bank, let’s take “account” as an example, you may have:

**url**

http://.../accounts/12345

**resource**

/accounts

**representation**

{

    "accounts": [{

    "balance": {

    "available": 350.15,

    "current": 149.85

    },

    "meta": {

    "limit": 500.00,

    "name": "My Credit Card",

    "number": "12345"

    },

    "institutionType": "NSE",

    "type": "credit"

    }]

}

APIs should only have two URIs that represent a given resource. In the example above, the API URIs to either fetch all accounts or fetch a specific account, respectively, would look something like:

/accounts

or

/accounts/123456

Note that many times there are cases where a specific account cannot be fetched without some other relationship tied to it. It’s, after all, almost certainly going to be associated with some account owner. In these scenarios, the following syntax should be used:

/{resource}/{id}/{resource}/{id}

As a result, account lookup API would look something like:

/customers/ABC123/accounts/123456

For simplicity’s sake, resources should rarely go deeper than what is outlined above. But, if necessary, the same /{resource}/{id} pattern should be followed.

Lastly, when assigning the resource name, it is better to use concrete names as opposed to making the resource overly abstract. For example, a resource of /locations may be too abstract. Does it refer to ATM Locations? Branches? Restrooms? Rather, more suitable resource names could be /atmLocations or /branchLocations. The exception, of course, would be if a parent resource is already present in the hierarchy. In that case, one might expect to see something like

/branches/locations

For a set of APIs, the key is keeping the number of resources at a manageable level for the end-user. Apigee suggests a target of approximately 12-24 resources names for a set of APIs. If we need to go much beyond that, we should revisit the API taxonomy.

**Methods**

In the world of REST APIs, **using verbs within the resource becomes cumbersome to navigate as an end user and MUST be avoided**. In account example the list can quickly grow if VERBS are incorporated:

/getAccount

/createAccount

/deleteAccount

/updateAccountStatus

/getActiveAccounts

/validateAccount

/getAllAccounts

/updateExpirationDate

/updateActiveCustomers

/increaseCreditLimit

However, the use of nouns combined with HTTP methods allows us to narrow this down to a single resource and 4 HTTP verbs:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operation** | **HTTP Method** | **Resource** | **Comments** |
| Create | POST | /accounts | Creates an account with the parameters passed in the message body |
| Read | GET | /accounts or /accounts/{uuid} | Fetches all accounts or a specific account’s details, respectively |
| Update | PUT | page9image7124480  /accounts/{uuid} | Updates an account with the information passed message body (*Full* resource) |
| Update | PATCH | /accounts/{uuid} | Updates an account (or accounts) with the information passed message body (*Partial* resource) |
| Delete | DELETE | /accounts/{uuid} | Moves the account(s) to a “deleted” status |

**Consistency across all API**

We want to keep consistency across APIs. If we have a series of APIs which return an “account” object, that object should be the same across those APIs. The only exception would be in the case of the same API being used both publicly and privately to NSE. In this case, only a subset of the “account” elements should be sent to external parties. However, the “account” object returned should be consistent.

We also need to take into account the context of the API request. For example, an /accounts request which returns a list of accounts, may return a subset of the “account” object returned when making an /accounts/1234 call which would return account details.

**Resource and Enumeration Naming Conventions**

The purpose of this living section is to define a strawman for standards and conventions to be followed for NSE Information Model aligned objects and elements in Digital APIs. Projects should conform to the standards. The Information Engineering team will have active involvement in participating in governance processes for all digital APIs and the intent is to evolve the standards to best satisfy the objective of promoting reuse across the Digital APIs.

**Conventions for Class names**

* Class names should be nouns
* Class names should be simple and descriptive
* Class names should follow UpperCamelCase with following considerations: -
  + Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used
* Class names should contain characters only. No numeric and special characters are allowed Class names should not contain non-approved abbreviations.
* Class names should not exceed 35 characters. If length exceeds 35 characters, then approved abbreviations should be used consistently from last word backward. Approved abbreviations can be found in appendix below
* Class names should not end with ‘Request’ and ‘Response’ since these are used to indicate root level classes (see next section)

*Example:* PartyRole, Address, DestinationAccount

**Conventions for Class attribute names**

* Class attribute names should be simple and descriptive
* Class attribute names should follow lowerCamelCase with following considerations: -
  + Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used e.g. use ‘url’ *instead of* ‘uniformResourceLocator’
* Class attribute names should contain alphanumeric characters only. No special characters are allowed Class attribute names should start with an alphabet and never with a numeric character.
* Class attribute names should not contain non-approved abbreviations. Class attribute name should not start with the following words: ‘get’ and ‘set’
* If a class attribute represents an array, then the attribute name should depict plural e.g. ‘balances’

*Example:* accountNumber, statusFlag

**Conventions for path parameter and query parameter names**

* Parameter names should be simple
* Parameter names should follow lowerCamelCase with following considerations:
  + Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used e.g. Use ‘url’ *instead of* ‘uniformResourceLocator’
* Parameter names should contain alphanumeric characters only. No special characters are allowed.
* Parameter names should start with an alphabet and never with a numeric character.
* Parameter names should not contain non-approved abbreviations.
* Query parameters that accept multiple values **MUST** be pluralized e.g. ‘accounts’
* All query parameters **MUST** be properly url-encoded. If a query parameter value includes the comma (,) character, the comma **MUST** be double encoded. Note that for readability purposes, the examples throughout this document do not show encoded query strings.

*Example:* accountNumber, statusFlag

**Data Types**

Swagger uses several known formats to more finely define the data type being used. However, the format property is an open string -valued property and can have any value to support documentation needs. Formats such as "email", "uuid", etc., can be used even though they are not defined by this specification. Types that are not accompanied by a property follow their definition from the JSON Schema. The formats defined by the Swagger Specification are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Common Name** | **type** | **Format** | **Comments** |
| integer | integer | int32 | signed 32 bits |
| long | integer | int64 | signed 64 bits |
| float | number | Float |  |
| double | number | Double |  |
| string | string |  |  |
| byte | string | Byte | base64 encoded characters |
| binary | string | Binary | any sequence of octets |
| boolean | boolean |  |  |
| date | string | Date | As defined by full-date - RFC3339 |
| dateTime | string | date-time | As defined by date-time - RFC3339 |
| password | string | password | Used to hint UIs the input needs to be obscured. |

You may have noticed that the examples above use lower camelCase in the resource naming conventions. There are many case variations out in the programming world. We settled on lower camelCase for several reasons:

* camelCase is more familiar to JavaScript developers and easier to parse
* camelCase is better aligned with Java Object Data Models in use by NSE

**One List to Rule Them All**

* Conventions for path parameter and query parameter names:\*
* Parameter names should be simple
* Parameter names should follow lowerCamelCase with following considerations:
  + Wherever standard acronyms are available and applicable, fully defined names or non-standard acronyms/abbreviations for the same should not be used e.g. Use ‘url’ *instead of* ‘uniformResourceLocator’
* Parameter names should contain alphanumeric characters only. No special characters are allowed.
* Parameter names should start with an alphabet and never with a numeric character.

**Collections**

A collection is a list of multiple Resources. A collection is represented as a JSON object.

* A collection **MUST** contain a resources field. The resources field is an array containing multiple Resources
* A collection **MAY** contain a field containing a pagination object.

**Example**

{

"resources": [{

"uuid": "a-b-c",

"created\_at": "2015-07-06T23:22:56Z",

"updated\_at": "2015-07-08T23:22:56Z",

"links": {

"self": {

"href": "/v1/accounts/a-b-c"

}

}

}, {

"uuid": "d-e-f ",

"created\_at": "2015-07-06 T23:22:56Z",

"updated\_at": "2015-07-08 T23:22:56Z",

"links": {

"self": {

"href": "/v1/accounts/d-e-f"

}

}

}],

"pagination": {

"total\_results": 2,

"total\_pages": 1,

"first": {

"href": "/v1/accounts?page=1&per\_page=10"

},

"last": {

"href": "/v1/accounts?page=1&per\_page=10"

},

"next": null,

"previous": null

}

}

**Pagination**

Pagination **may** be used by Collections to limit the number of resources returned at a time. Pagination is requested by a client through the use of query parameters. Pagination is represented as a JSON object.

Pagination **MUST** include a total\_results field with an integer value of the total number of records in the collection.

Pagination **MUST** include the following fields for pagination links:

* first : the first page of resources
* last : the last page of resources
* previous : the previous page of resources
* next : the next page of resources

Pagination links may be null . For example, if the page currently being displayed is the first page, then previous link will be null.

When pagination links contain a URL, they **MUST** be a JSON object with a field named href containing a string with the URL for the next page.

The URL **MUST** include all query parameters required to maintain consistency with the original pagination request. For example, if the client requested for the collection to be returned in a specific order direction via a query parameter, then the pagination links must include the proper query parameter to maintain the requested direction.

The following query parameters **MUST** be used for pagination:

* page : the page number of resources to return (default: 1)
* per\_page : the number of resources to return in a paginated collection request (default: 50)
* order\_by : a field on the resource to order the collection by; each collection may choose a subset of fields that it can be sorted by

When collections are ordered by a subset of fields, each field **MAY** be prepended "-" to indicate descending order direction. If the field is not prepended, the ordering will default to ascending.

If there are additional pagination query parameters, the parameters **MUST** have names that conform to the acceptable query parameter names.

**Example**

"pagination": {

"total\_results": 20,

"total\_pages": 2,

"first": {

"href": "/v1/accounts?order\_by=\*created\_at&page=1&per\_page=10"

},

"last": {

"href": "/v1/accounts?order\_by=\*created\_at&page=2&per\_page=10"

},

"next": {

"href": "/v1/accounts?order\_by=\*created\_at&page=2&per\_page=10"

},

"previous": null

}

**Errors**

**Status Codes**

The HTTP status code returned for errors **MUST** be included in the documented status codes.

This proposal includes code which would be an internal unique identifier of a class of error. The method for maintaining a list of these codes and their meanings would need to be determined

**Response Codes**

**Successful Requests**

|  |  |  |
| --- | --- | --- |
| **Status Code** | **Description** | **Verbs** |
| 200 OK | This status **MUST** be returned for synchronous requests that complete successfully and have a response body. This must only be used if there is not a more appropriate 2XX response code. | GET, PATCH, PUT |
| 201  Created | This status **MUST** be returned for synchronous requests that result in the creation of a new resource. | POST |
| 202  Accepted | This status **MUST** be returned for requests that have been successfully accepted and will be asynchronously completed at a later time. See more in the async section. | POST, PATCH, PUT, DELETE |
| 204 No  Content | This status **MUST** be returned for synchronous requests that complete successfully and have no response body. | DELETE |

**Redirection**

|  |  |  |
| --- | --- | --- |
| **Status Code** | **Description** | **Verbs** |
| 302  Found | This status **MUST** be returned when the API Gateway redirects to another location. | GET |
| 303  See Other | This status **MUST** be returned when an async job finishes. It must include a location header containing the resource link. See more in the async section. | GET |

**Client Errors**

|  |  |  |
| --- | --- | --- |
| **Status Code** | **Description** | **Verbs** |
| 400 Bad Request | This status **MUST** be returned for requests that provide malformed or invalid data. Examples: invalid JSON, unexpected query parameters or request fields. | GET, PATCH, POST, PUT, DELETE |
| 401  Unauthenticated | This status **MUST** be returned if the requested resource requires an authenticated user but there is no OAuth token provided, or the OAuth token provided is invalid. | GET, POST, PATCH, DELETE, PUT |
| 403 Forbidden | This status **MUST** be returned if the request cannot be performed by the user due to lack of permissions. Example: User with read-only permissions to a resource tries to update it. | POST, PATCH, DELETE, PUT |
| 404 Not Found | This status **MUST** be returned if the requested resource does not exist. | GET, POST, PATCH, PUT, DELETE |
| 405 Method Not Allowed | This status **MUST** be returned if the user requesting the resource has insufficient permissions to view the resource. | GET, POST, PATCH, PUT, DELETE |
| 422  Unprocessable Entity | This status **MUST** be returned if the request is semantically valid, but performing the requested operation would result in a invalid state.  Example: Attempting to start an app without assigning a droplet. | POST, PATCH, PUT |

**Server Errors**

|  |  |
| --- | --- |
| **Status Code** | **Description** |
| 500 Internal Server Error | This status **MUST** be returned when an unexpected error occurs. |
| 502 Bad Gateway | This status **MUST** be returned when an upstream service failure causes a request to fail. Example: Being unable to reach requested service broker. |
| 503 Service Unavailable | This status MUST be returned when the server is not ready to handle the request |
| 504 Gateway Timeout | This status MUST be returned when an upstream service did not get a response in time. |

**Asynchronicity**

Endpoints are responsible for behaving either asynchronously (return 202) or synchronously (don't return 202). For async endpoints:

POST /v1/resource

The CC will return a 202 with a location header pointing to the job. Depending on the resource, it may also return a skeletal body containing the partial resource.

202 Accepted

Location: /v1/jobs / 123

{

"message": "Task queued"

}

Before the job has completed, GET requests made to the job endpoint will return 200 with information about the status of the job.

GET /v1/jobs/123

200 OK

{

"status": "in progress"

}

When the job has completed, GET request made to the job endpoint will return 303 and a location header to the resource (assuming it still exists).

GET / v1 / jobs / 123

303 See Other

Location: /v1/resource /:uuid

{}

Note that for asynchronous deletes, the redirect location will be to a no-longer-existent resource.

# Microservice Framework Components

This section defines the components that are required in a typical microservice architecture. Provided below is a logical view of the microservice architecture.

A screenshot of a cell phone

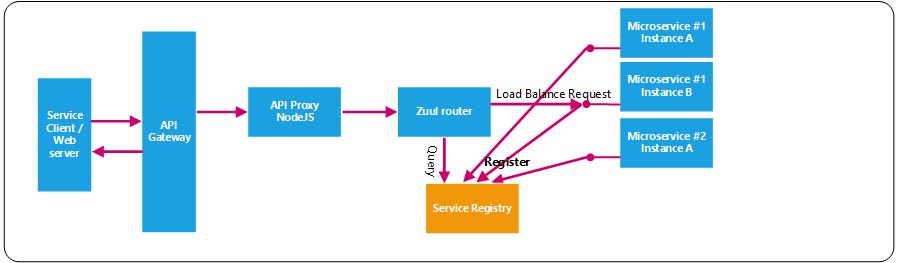
Description automatically generated

# Service Registry

A Discovery Server, based on Spring Cloud Netflix Eureka, which holds the information about all client-service applications. All microservices register into the Eureka Server. The server allows services to find and communicate with each other due to the dynamic nature of service instance (in terms of number of instances and assigned IP address).

**Usage of framework component:**

* On all requests coming in from Apigee Gateway, Zuul proxy server performs a lookup on Service Registry for healthy service instances, before routing to service instances
* For microservice to microservice synchronous invocation, Netflix Ribbon performs an internal lookup on Service Registry



**Framework Component:**

Spring Cloud Netflix Eureka

**Significant Annotations & Properties:**

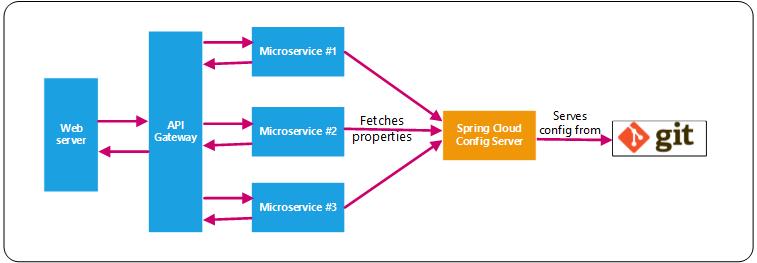
|  |  |  |
| --- | --- | --- |
| Annotation / Properties | Component | Dependencies |
| @EnableEurekaServer | Service Registry server | spring-cloud-starter-netflix-eureka-server |
| @EnableDiscoveryClient | Microservice (client) apps | spring-cloud-starter-netflix-eureka-client |
| @FeignClient | Microservice client app to invoke target microservice | spring-cloud-starter-openfeign |
| bootstrap.properties  eureka.client.serviceUrl.defaultZone=  =${EUREKA\_SERVER:http://localhost:8761 /eureka} | Specifies location of eureka server to microservice apps |  |

# Config Server

A Config Server is developed for microservices to consume externalized configuration on startup, and then refresh the configuration without restart. It provides a central place to manage external properties for applications across all environments such as dev, prod etc.

**Usage of framework component:**

* Config server to point to git (bitbucket) repository that contains microservices’ property sources
* Some of the property values to be mapped from TAS environment variables
* Microservice application to point to config server for properties



**Framework Component:**

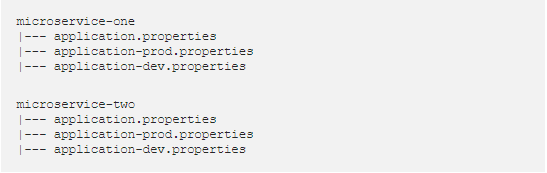
Spring Cloud Config Server

**Significant Annotations & Properties:**

|  |  |  |
| --- | --- | --- |
| Annotation / Properties | Component | Dependencies |
| @EnableConfigServer | Config server | spring-cloud-config-server |
| ConfigServer’ application.properties  spring.cloud.config.server.git.uri=https://  toolstack.nseindia.com/stash/scm/~joseph.kevin/test-config.git | Git repository containing property sources |  |
| The below are defined on the microservice applications | | |
| @RefreshScope | To refresh the configuration of microservice without a restart with the execution of POST request | spring-boot-starter-actuator |
| bootstrap.properties  spring.application.name=<service-name>  spring.profiles.active=production  spring.cloud.config.uri=${CONFIG\_SERVER: http://localhost:8888}  spring.cloud.config.fail-fast=true | Specifies location of config server to microservice apps | spring-cloud-config-server |
| @ConfigurationProperties or  @Value | To access any property from the Config Server |  |

**Config server Git repository**

The git repository linked by Config server can be structured as follows:



# Auto-scaling

To automatically scale individual microservice applications based on rules that are set. This configuration is defined in a manifest file within the microservice project.

This service is part of the TAS platform with no additional licensing required. Open question - Need to check if it is possible to apply a set of rules on a group of microservices instead of individually for each microservice.

**Usage of framework component:**

* Create an autoscaler\_manifest.yml for individual microservice that will define auto-scaling parameters.

Provided below is an example which sets minimum number of instances as one and maximum number of instances as two. When the http\_throughput crosses 20, the number of instances scale up by one, and if it goes below 20, the number of instances scale down by one.



**Framework Component:**

PCF provided marketplace service

# Hystrix Circuit Breaker

As services collaborate when handling requests, the failure of one service, either unavailable or exhibiting high latency (and unusable), can potentially cascade to other services throughout the application. This also causes resource exhaustion, an example is threads being consumed while waiting for called services to respond, which makes the calling service unable to handle other requests.

To enable fault tolerance in the application, it is recommended to apply the circuit breaker design pattern where by the execution falls back to a different path automatically in case of any failure. In this manner, the services handle the failure of the services that they invoke.

**Usage of framework component:**

* A template of the FeignClient and the circuit breaker pattern with Hystrix has been created.
* It can be cloned by developers and modified as per user story

**Framework Component:**

Spring Cloud Netflix Hystrix

**Significant Annotations & Properties:**

|  |  |  |
| --- | --- | --- |
| Annotation / Properties | Component | Dependencies |
| @EnableDiscoveryClient | Service Registry used by Netflix Ribbon to locate the microservice for invocation | spring-cloud-starter-netflix-eureka-client |
| @EnableFeignClients  @FeignClient | A REST client that makes writing web service client easier | spring-cloud-starter-openfeign |
| @HystrixCommand | Enables method calls to be executed using a circuit breaker implemented with Hystrix | spring-cloud-starter-netflix-hystrix |
| @EnableHystrixDashboard | To monitor the Circuit Breaker status on a dashboard | spring-cloud-starter-netflix-hystrix-dashboard |

**Significant Components:**

1. Usage of Feign Client for a synchronous microservice to microservice communication, which internally uses Netflix Ribbon (client-side) load balancer. It is integrated with Eureka for dynamic service discovery.

Provided below is an example of the invocation of the common Audit microservice.

@FeignClient(value = "audit-microservice",configuration = FeignClientsConfiguration.class, fallback = AuditServiceFallback.class)

public interface AuditService {

@RequestMapping(value = "/v1/audit/{id}", method = RequestMethod.GET)

public ResponseEntity<Optional<AuditEvent>> getAuditById(@PathVariable(name = "id") Integer id);

@RequestMapping(value = "/v1/audit", method = RequestMethod.GET)

public ResponseEntity<List<AuditEvent>> getAudit();

@RequestMapping(value = "/v1/audit", method = RequestMethod.POST)

public ResponseEntity<AuditEvent> addAudit(@RequestBody AuditEvent audit) throws InterruptedException, ExecutionException, JsonProcessingException;

}

1. In the event of a failure, a circuit breaker is implemented with Hystrix.

As an example, the synchronous invocation is replaced with a non-blocking push of audit message to Kafka topic.

@Component

public class AuditServiceFallback implements AuditService {

@Autowired

private KafkaProducer kafkaProducer;

@Override

public ResponseEntity<Optional<AuditEvent>> getAuditById(Integer id) {

AuditEvent defaultAudit = new AuditEvent();

defaultAudit.setUser("DefaultUser");

………………………………………………….

defaultAudit.setUrl("DefaultUrl");

return new ResponseEntity<>(Optional.ofNullable(defaultAudit),HttpStatus.OK);

}

@Override

public ResponseEntity<List<AuditEvent>> getAudit() {

return new ResponseEntity<>(Collections.emptyList(),HttpStatus.NOT\_FOUND);

}

@Override

public ResponseEntity<AuditEvent> addAudit(AuditEvent audit) throws InterruptedException, ExecutionException, JsonProcessingException {

//push audit as event to Kafka

kafkaProducer.publish(audit);

return new ResponseEntity<>(audit,HttpStatus.OK);

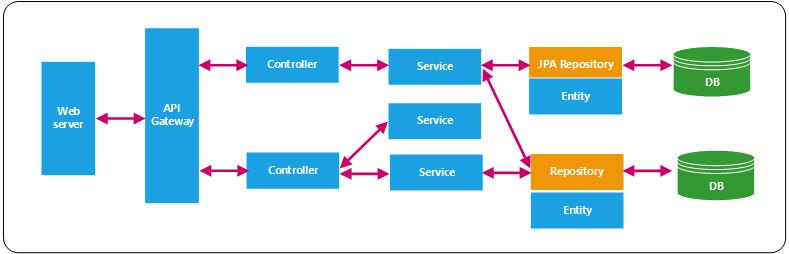
}

}

# Spring Data JPA

Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the boilerplate code to be written. The developer writes the repository interfaces, including custom finder methods, and Spring provides the implementation automatically.

The Lombok project further reduces boilerplate code. It is used to avoid writing getters and setters, constructors, toString, etc. from the POJOs. It also provides Logback logging features without explicitly calling Logger factory method of Slf4J.



**Usage of framework component:**

It is a template that is to be cloned by developers while working with Oracle DB, and modified as per user story

**Framework Component:**

Spring Data JPA

**Significant Components:**

1. Define Entity, mapping to Oracle RDBMS table

**@Data //Lombok project usage  
@Entity**public class Inventory implements Serializable {  
  
 @Id  
 **@GeneratedValue(strategy = GenerationType.*AUTO*)**  
 private int id;  
 private String productCode;  
 @OneToMany(mappedBy = "inventory", cascade = CascadeType.*ALL*, fetch =

FetchTy pe.*LAZY*)  
 @JsonManagedReference  
private List<Vendor> vendors;  
}

1. For audit requirements, model classes can be audited with @CreatedBy, @LastModifiedBy, @CreatedDate, @LastModifiedDate annotations. Spring automatically populates the annotated fields with the principal who created the object, last modified it, and the date of creation, and last modification.

public class <ClassName> {

// ….

@CreatedBy

User creator;

@LastModifiedBy

User modifier;

@CreatedDate

Date createdAt;

@LastModifiedDate

Date modifiedAt;

// ….

}

1. Define Repository interface by extending JPARepository<T, ID>, which provides various pre-defined methods for CRUD operations - findAll(), findById(), findOneById(), findOne(), getOne(), count(), deleteById(), save(), etc.

@Repository

@Transactional(readOnly = true)

**public interface** InventoryRepository **extends** JpaRepository<Inventory,Integer>{}

1. Customized methods can be defined in the interface based on the columns specified in the Entity<T>

**@Repository**public interface InventoryRepository extends JpaRepository<Inventory,Integer> {  
  
 public Optional<Inventory> findByProductCode(String productCode);

public Optional<Inventory> findByVendorName(String vendorName);

public void deleteByVendorName(String vendorName);

**}**

1. Named and Native Queries can be further defined if required
2. Properties for connecting to a local instance of Oracle database

spring.datasource.url= jdbc:oracle:thin:@3.7.251.1:1521:PRVTNDB

spring.datasource.username=

spring.datasource.password=

spring.datasource.hikari.poolName = Hikari\_P

spring.datasource.hikari.auto-commit = false

spring.datasource.jpa.show-sql = true

spring.jpa.properties.hibernate.format\_sql=true

#logging.level.org.hibernate.type=trace

#logging.level.org.hibernate.SQL=DEBUG

#logging.level.org.hibernate.type.descriptor.sql.BasicBinder=TRACE

# Spring Data MongoDB

MongoDB is a popular open source, schema less, secure, highly scalable, NoSQL database. Spring Data MongoDB project is to be used to access data from MongoDB.

**Usage of framework component:**

It is a template that is to be cloned by developers while working with MongoDB, and modified as per user story

**Framework Component:**

Spring Data MongoDB

Spring Data MongoRepository provides common functionalities for easy plug-in and usage.

1. Mark a class as a domain object with the @Document annotation, for persisting to the Collection. The @Field annotation is used to configure the name of the field to be used when MongoDB persists the document.

**@Data //Lombok project usage  
@Document(collection=”product”)**public class Product {  
  
 @Id  
 private int id;

@Field(”code”)  
 private String code;

@Field(”name”)  
 private String name;

**@PersistenceConstructor**

public Product (String code, String name) {

this.setCode(code);

this.setName(name);

}

}

1. For audit requirements, model classes can be audited with @CreatedBy, @LastModifiedBy, @CreatedDate, @LastModifiedDate annotations. Spring automatically populates the annotated fields with the principal who created the object, last modified it, and the date of creation, and last modification.
2. Define the Repository interface with Spring Data providing auto-generated queries out of method names.

@Repository

public interface ProductRepository extends MongoRepository<Product, Integer> {

List<Product> findByCode(String code);

List<Product> findByNameStartingWith(String regexp);

Product save(Product product);

}

1. The @Query annotation can also be used to specify a raw query as a Mongo JSON query string.

**@Repository**public interface ProductRepository extends MongoRepository <Product,Integer> {

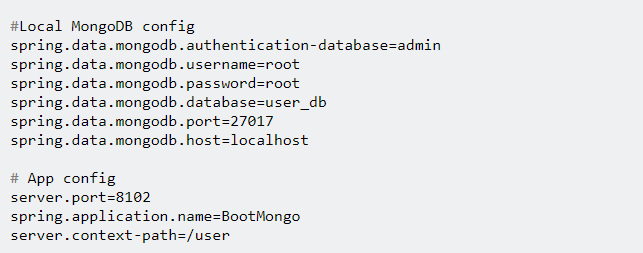
//not the best of example!

@Query(“{ ‘$or’ : [{‘code’ : ?0},{‘name’ : ?1}]}”)

List<Product> findByCodeOrName(String code, String name)

**}**

1. Properties for connecting to a local instance of MongoDB



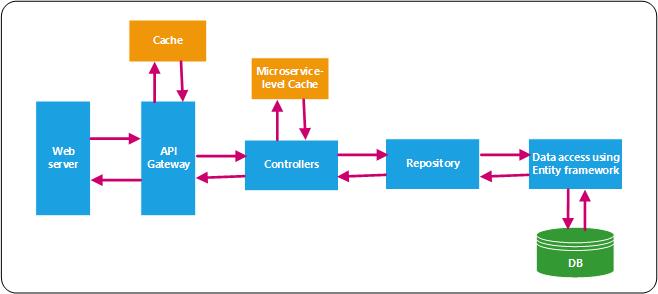
**Design Considerations:**

* To Identify appropriate user stories for implementation of MongoDB document database

# Redis Caching

The primary benefits of caching is to reduce load on web services & database, and increase performance as the latency to the database is removed.

* Caching can be configured on API Gateway for caching certain responses for faster response.
* At an individual microservice level, server-side caching can be implemented where in server can cache responses that are expensive to generate but don’t change very often. Server-side caching can be implemented with Spring’s built-in cache management capability and Redis, a noSQL datastore used as a cache



**Usage of framework component:**

* It is a template that is to be cloned by developers while working with server-side caching requirements (Redis), and modified as per the user story

**Framework Component:**

Spring Data Redis

**Design Considerations:**

* Careful planning required to identify cacheable requests

**Significant Annotations & Properties:**

|  |  |
| --- | --- |
| Annotation / Properties | Component |
| @EnableCaching | Enables Spring’s built-in cache management capabilities. |
| @Cacheable(value = "owners", key = "#ownerId")  @GetMapping(value = "/{ownerId}") | It denotes that the results of invoking this method will be cached. If the method is called again with the same arguments, the results will be retrieved from the cache instead of invoking the method. |
| @CachePut(value = "owners", key = "#ownerId")  @PutMapping(value = "/{ownerId}") | It denotes that a method will trigger the cache put operation. |
| @CacheEvict(value = "owners", key = "#ownerId")  @DeleteMapping(value = "/{ownerId}") | It denotes that a method will trigger the cache evict operation. |
| application.properties  spring.cache.type=redis  spring.redis.host=<ip-address>  spring.redis.port=6379  spring.redis.password= | Redis connection parameters |

# Event-driven architecture based on Kafka

The request-response design pattern can result in network congestion as the microservice call paths might be deep and one slow service can further aggravate the issue. Therefore, it is suggested to always consider an event-driven architecture when appropriate for a use case.

The main “event” based design patterns can be categorized into one of the following:

* Event-Carried State Transfer: This is the most commonly used event based design pattern, wherein the microservice publish events with events that contain details of the data that changed, so that subscribing microservice don’t need to communicate with the publishing microservice in order to do its work.

An example is the invocation of the common Audit microservice.

* Event Notification: This happens when a microservice sends event messages to notify other services of a change in its domain. There is a marked separation between the logic flow that sends the event and any logic flow that responds to some reaction to that event.

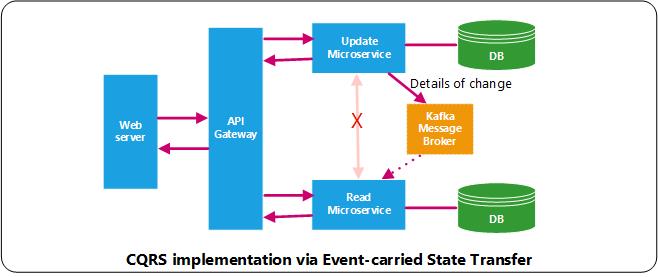
To be noted here, is that the subscribing microservice needs to fetch the state from the publishing microservice.

* Event-Sourcing: The core idea is that whenever a change is made to the state of a system, the state change is recorded as an event in an event store. The event store becomes the principal source of truth, and the system state is purely derived from it.
* CQRS: Command Query Responsibility Segregation (CQRS) is the notion of having separate data structures for reading and writing information.

In an event-driven architecture,

* A microservice publishes domain events when the domain model is created or updated, which are consumed by other command microservices.
* The command microservice receives the events and can update its own data stores to keep data in sync.

Apache Kafka, a distributed and fault-tolerant stream processing platform, has been selected as the message broker.



**Usage of framework component**

* A common library containing Kafka publish and subscribe methods

Publish

* Include the jar in garage-specific microservices that are implementing a non-blocking microservice call.
* Define the Kafka topic in the config server for the publish microservice, and invoke the Kafka publish bean with the message for publish to the topic

Subscribe

* Include the jar in garage-specific microservices that need to subscribe to a Kafka topic
* Define the Kafka topic in the config server for the publish microservice, and invoke the Kafka publish bean with the message for publish to the topic

**Framework Component:**

Spring Kafka

**Design Considerations:**

* Identify use cases to be implemented with an event-driven architecture

**Significant Annotations & Properties:**

1. The Kafka publish class in common library

public class Producer {

private static final Logger LOGGER = LoggerFactory.getLogger(Producer.class);

private ObjectMapper objectMapper = new ObjectMapper();

private KafkaProducer<String, String> producer;

private final KafkaConfiguration kafkaConfiguration;

private final String topic;

public Producer(KafkaConfiguration kafkaConfiguration,@Value("${kafka.producer.topic}") String topic) {

this.kafkaConfiguration = kafkaConfiguration;

this.producer = new KafkaProducer<>(kafkaConfiguration.getProducerProps());

this.topic = topic;

}

public void publish(AuditEvent auditEvent) throws ExecutionException, InterruptedException, JsonProcessingException {

String key = UUID.randomUUID().toString();

String message = objectMapper.writeValueAsString(auditEvent);

LOGGER.debug("Sending to Kafka topic {} with key {} the message : {}", topic, key, message);

producer.send(new ProducerRecord<>(topic, key, message)).get();

}

}

1. In the publish microservice, import the common jar, and invoke the method with the event data for publish to the topic.

public class AuditServiceFallback implements AuditService {

@Autowired

private Producer producer;

@Override

public ResponseEntity<AuditEvent> addAudit(AuditEvent audit) throws

InterruptedException, ExecutionException, JsonProcessingException {

//push audit as event to Kafka

producer.publish(audit);

return new ResponseEntity<>(audit,HttpStatus.OK);

}

}

1. Similarly, define a Kafka consumer class in common library, which is imported in the subscribe microservice

public class Consumer {

….

}

1. In the subscribe microservice, process the event message.

@SpringBootApplication

public class AuditMicroservice {

@Autowired

Consumer consumer;

public static void main(String[] args) {

SpringApplication.run(AuditMicroservice.class, args);

}

@PostConstruct

public void startConsumer() {

consumer.consume();

}

}

@Component

public class AuditEventProcesser {

public void processAuditEvent(String eventLog) throws JsonProcessingException {

AuditEvent auditEvent = objectMapper.readValue(eventLog,AuditEvent.class);

auditService.add(buildAudit(auditEvent));

}

private Audit buildAudit(AuditEvent event) {

Audit audit = new Audit();

audit.setCreatedDate(event.getCreatedDate());

return audit;

}

}

# Microservice Cross-cutting Components

This section addresses cross-cutting concerns of Access Management, Application Logging, Auditing, Server-side Validation and Exception Handling that affects the entire application.

# Access Management

There are various levels at which access management concerns are to be addressed for a digital platform:

1. Access to the enterprise’s domain or intranet via LDAP and/or OTP sign-in
2. Achieve Single Sign-On (SSO) via SAML, enabling user login to the web applications or native apps
3. OAuth (Open Authorization) for the protected server-side resources - REST APIs
4. Access requirements for the various integrations with the core systems

The scope of this document is to address the Authentication & Authorization requirements for server-side REST APIs (Point 3) and integrations with the core systems (Point 4).

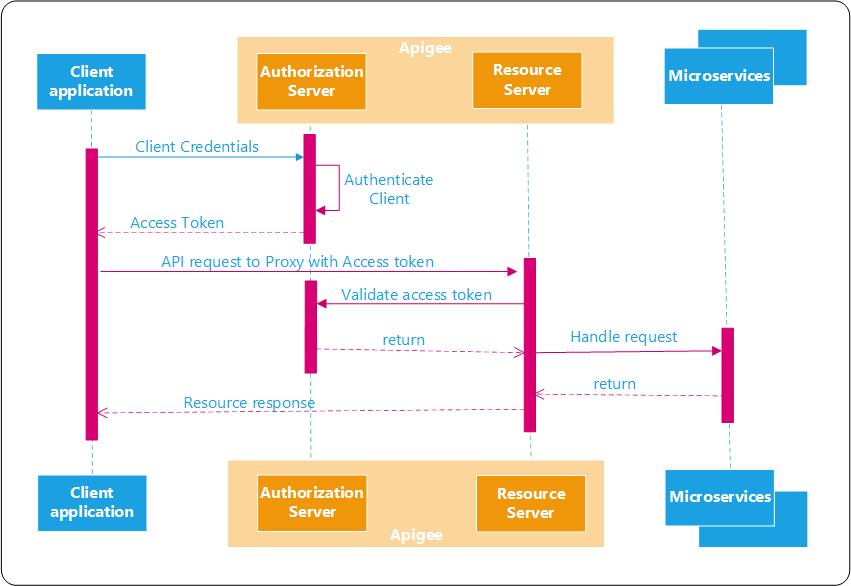
The server side resources, REST APIs, are to be protected with OAuth, enabling token-based authentication and authorization.

* It provides a way for end applications to gain limited access to a protected resource without the need for the client to divulge their login credentials to the microservices
* For the microservices, it allows a client to exchange some credentials for a token, and that token gives the client access to the API.

Though OAuth is an incredibly useful security standard, it’s made up of a complex family of specs, and there are numerous ways to use it. To make leveraging OAuth simpler and remove the complexity from individual microservices, it is to be implemented as follows:

* All the microservices are to be behind a firewall, and accessible only through (Apigee) API proxy which authenticates requests and forwards them to microservices, which might in turn invoke other services.
* The API Management platform, Apigee will be used to generate & validate OAuth access tokens (JSON Web Token) and apply granular control over what a token is allowed to do.
* The access token securely identifies the requestor in each request to the microservice, which can be propagated in requests it makes to other microservices.
* The microservices can also verify that the requestor is authorized to perform an operation based on the access token, as Header can include further metadata attributes such as App id, User id, Role, etc.
* OAuth Grant Type selected depends on use case or client app, as some grant types are more secure than others. Most of the NSE web and mobile app API invocations will use the Client Credentials grant type, as the client app is also the resource owner.

**OAuth workflow based on Grant Type = Client Credentials**



**Usage of framework component:**

1. Include the configuration WebSecurityConfig that extends WebSecurityConfigurerAdapter
2. Include a common component to validate session id and client IP

**Framework Component:**

API Management platform, Apigee

**Significant Annotations:**

1. Microservice Authentication and Authorization requirements are handled on the API Management platform
2. Enable XSS protection with Java Configuration:

@EnableWebSecurity

@Configuration

public class WebSecurityConfig extends

WebSecurityConfigurerAdapter {

@Override

protected void configure(HttpSecurity http) throws Exception {

http

.headers()

.xssProtection()

.and()

...;

}

}

1. A common component to (Work in Progress):

* Persist and Validate session id
* Validate client IP address

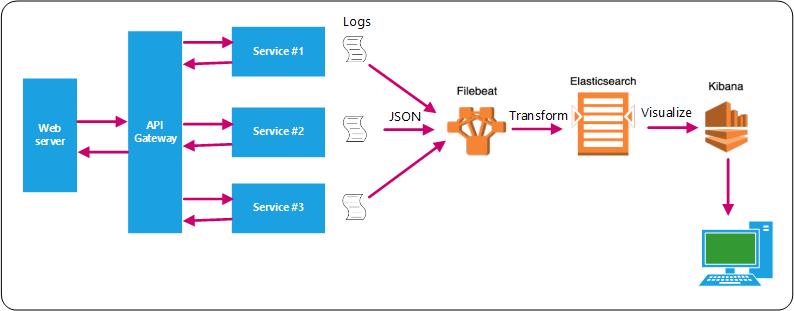
# Application Logging

Log Management and archival –

* What about logs from other layers such as web, gateway, database, etc. Feed into ELK?
* Elasticsearch archival process?

It is critical that each individual microservice produces structured log messages that indicate every action and event. A Filebeat service running on TAS can subsequently ship the log messages for aggregation based on Tracing IDs, which can assemble full pictures of events in the system for analysis and action.

* Analyze an issue in production
* Used to identify ongoing problems and predict potential issues



1. This section covers application logging
2. The next section - Distributed Tracing addresses enabling of Tracing ids in the logs
3. The subsequent section - Elastic Stack addresses Log aggregation and analysis

**Usage**

1. Common Spring AOP based logging project developed. For e.g. to log at the beginning and exit of method execution in specific layer
2. To be included as a dependency in microservice applications
3. Custom logging annotation can be used in microservice applications as required. For e.g. @CalculateTime to calculate method execution

**Implementation steps**

1. SLF4J provides a standardized API that is implemented by the popular Logging frameworks. It enables to change the logging framework without changing application code, only change the dependency to a different framework that implements the SLF4J interfaces.

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**@Data //Lombok project usage  
@Entity**public class ConsumerImpl implements Consumer {  
  
 private static final Logger logger =LoggerFactory.getLogger(ConsumerImpl.class);  
 logger.info(“{}{}”,param1, param2);

}

1. Among others, the popular Logging framework options are Logback and Log4j2. Log4j2 is the more advanced and faster logging framework, with the dependencies listed below.

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-slf4j-impl</artifactId>

</dependency>

<!-- For handling JSON conversion -->

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.10.0</version>

</dependency>

1. Configure Log4j2.xml to structure log data into a standard format like JSON (Log4j2.xml)

<configuration>

<!-- Appender to log in a JSON format -->

<Appenders>

<Console name="ConsoleJSONAppender" target="SYSTEM\_OUT">

<JsonLayout complete="false" compact="false">

<KeyValuePair key="myCustomField1" value="myCustomValue1" />

<KeyValuePair key="myCustomField2" value="myCustomValue2" />

</JsonLayout>

</Console>

</Appenders>

<Logger name="CONSOLE\_JSON\_APPENDER" level="TRACE" additivity="false">

<AppenderRef ref="ConsoleJSONAppender" />

</Logger>

​

</configuration>

1. Microservices to log below attributes in JSON format to provide application insights

* Trace\_id (in response payload if possible)
* Span\_id / Parent\_span\_id ?
* Server\_ip\_address
* Timestamp
* Service\_name
* Service\_version
* URI
* Http\_Method
* Class\_Name
* Method\_name
* External\_service\_name (database, etc)
* File\_name
* LogLevel
* Http\_Code
* Esception\_object
* User\_id
* User\_role

Log message structure

1. Logging points

* Logger interceptor at entry and exit of method calls
* Invocation of other services
* Publish and Subscribe of events
* Exceptions
* …

**Reference from Architecture document:**

Section 15.4 Best practices for high availability and scalability -> Use ELK for centralized logging

**Framework Component:**

SLF4J and Log4j2

**Design Considerations:**

* Review fields to log and points of invocation

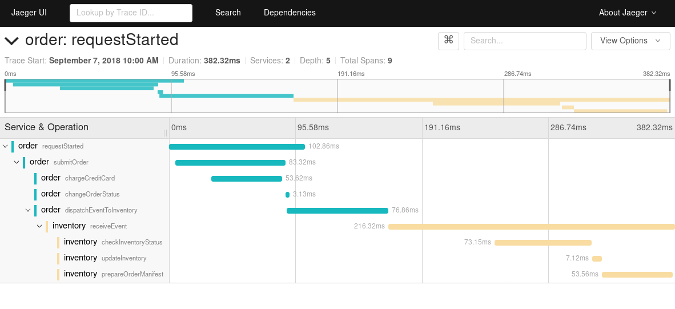
# Distributed Tracing

Distributed Tracing is crucial for troubleshooting and understanding microservices. It enables a Trace Id in the logs, which is useful for tracking the request passing through multiple microservices. Distributed Tracing can also be used to measure the performance of the microservices.

It is easy to identify which microservice has failed or has performance issues whenever there are multiple service calls within a single request.

**Zipkin UI**

It provides a view of the microservice call chain and related execution time.



**Reference from Architecture document:**

It is recommended in an Event-driven architecture that has been suggested in Section 1.10

**Framework Component:**

Spring Cloud Starter Zipkin

**Significant Annotations & Properties:**

|  |  |  |
| --- | --- | --- |
| Annotation / Properties | Component | Dependencies |
| application.properties  spring.zipkin.baseUrl = http://localhost:9411/  spring.zipkin.enabled = true | Specifies location of Zipkin server to microservice apps | spring-cloud-starter-zipkin  **Transitive dependences**  spring.cloud.starter.sleuth  spring.cloud.sleuth.zipkin |

# Elastic Stack for Log Aggregation

This section describes the usage of the Elastic stack and builds upon the previous two sections, where in we define the logging approach and the addition of a trace Id to the logs.

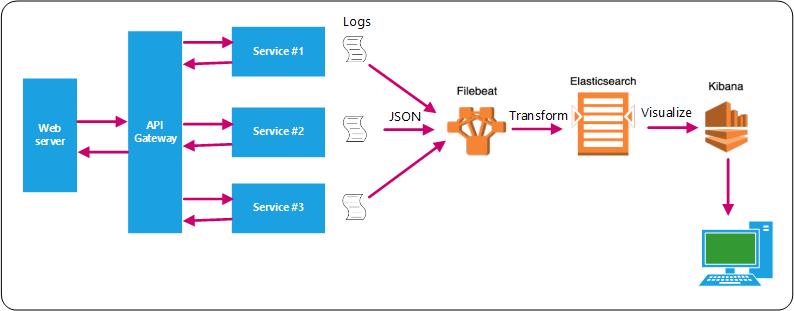
Elastic stack, a group of open source products from Elastic, provides the following features:

1. Filebeat ships JSON logs real-time from the multiple microservices to Elasticsearch, either directly or through Logstash. It is usually preferred to keep all logs so as to not lose context or metadata from the log itself.

* Filebeat is a “log shipper” that is installed on servers as agents
* Logstash is a data collection engines that unifies and normalizes data from disparate sources
* Elasticsearch is a RESTful distributed JSON based search engine

1. Kibana is a data visualization and exploration tool. It is aided by tracing IDs that help assemble full pictures of events in the system. The ability to filter on the metadata, and visualize the log message becomes a huge improvement for human comprehension.

* Log aggregation and visualization can be useful for debugging in a microservices architecture that involves multiple remote processes. As the calls are disjointed by their very nature, log aggregation provides a better picture of the system as a whole.
* Operations can leverage the visualization of logs when troubleshooting as well as inspecting system health. The operational value of visualizations of aggregated logs cannot be underestimated.



**Reference from Architecture document:**

Section 15.4 Best practices for high availability and scalability -> Use ELK for centralized logging

**Framework Component:**

Elastic stack, which includes Filebeat, Logstash, Elasticsearch and Kibana open source products

**Setup of Elastic stack:**

1. Setup Filebeat to ship microservice JSON log files to Elasticsearch as a JSON object. Need to analyze how to do the same from PCF to on-premise / cloud based Elastic stack.

#=========================== Filebeat inputs =============================

# Configure the input to access loggregator to forward the log events.

filebeat.inputs:

- &cloudfoundry

type: cloudfoundry

client\_id: ${CLIENT\_ID:filebeat}

client\_secret: ${CLIENT\_SECRET:changeme}

api\_address: https://api.sys.parivartandev1.com

#doppler\_address: ${DOPPLER\_ADDRESS}

#uaa\_address: ${UAA\_ADDRESS}

#rlp\_address: ${RLP\_ADDRESS}

#shard\_id: ${SHARD\_ID}

type: log

# Change to true to enable this input configuration.

enabled: true

# Paths that should be crawled and fetched. Glob based paths.

# To fetch all ".log" files from a specific level of subdirectories

# /var/log/\*/\*.log can be used.

# For each file found under this path, a harvester is started.

# Make sure not file is defined twice as this can lead to unexpected behaviour.

paths:

- /var/log/\*.log

#================================ Outputs =====================================

# Configure the Elasticsearch output either to a specific host

#-------------------------- Elasticsearch output ------------------------------

output.elasticsearch:

# Array of hosts to connect to.

hosts: ["10.166.16.57:9200"]

# Protocol - either `http` (default) or `https`.

#protocol: "https"

# Authentication credentials - either API key or username/password.

#api\_key: "id:api\_key"

#username: "elastic"

#password: "changeme"

#================================ Processors ====================================

# Configure processors to enhance or manipulate events generated by the beat.

processors:

- add\_cloudfoundry\_metadata:

<<: \*cloudfoundry

1. Open browser and navigate to Kibana dashboard to view the logs.
   * Create index pattern
   * Search on trace id
   * Create Alerts based on a pattern

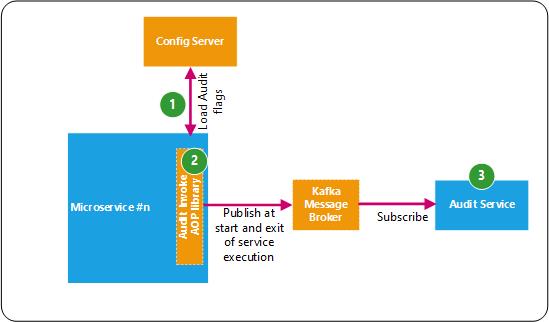
# Auditing

Auditing requirement is applicable on the microservice layer only. Auditing to be developed as a microservice, which can be invoked from other microservice either via:

* the standard synchronous Request-Response design pattern
* a non-blocking asynchronous event via Kafka

A Spring AOP based implementation is being considered for invoking the Audit component during entry and exit to the controller layer of the microservices and during exception processing.

For every invocation of a microservice, there will be a record (row) inserted in the Audit database, with the table partitioned on CreatedDate. The need for Oracle table sharding needs to be evaluated.



**Sequence:**

1. The config of a microservice will be loaded during container start-up. It will contain flags that will help control the switch on-off the audit requirement of the microservice.
2. Spring AOP aspect to be developed, with a Pointcut to intercept the call during entry and exit to the controller layer and exception processing. The Advice will check if the audit flag is enabled, and accordingly prepare an auditEvent and publish the auditEvent on to the Kafka broker.

The Spring AOP aspect will be developed as a library and included as a dependency in the microservice, thereby providing a consistent behaviour across all microservices.

1. The Audit microservice will subscribe to the topic and insert or update the Audit table record.

The API endpoints are as follows

|  |  |
| --- | --- |
| **API endpoints** | **Descriptions** |
| POST /v1/audit | Create an entry in the Audit table on start of microservice processing |
| PUT /v1/audit | Update the entry in the Audit table on exit of microservice processing, including during error |
| GET /v1/audit/{id} | Retrieve audit record based on Audit table primary key |
| GET /v1/audit/user/{user}/timewindow | Retrieve audit record by user performing the transaction based on a time window |
| GET /v1/audit/traceId/{traceId} | Retrieve audit record by trace id (generated by Zipkin) |
| GET /v1/audit/serviceId/{serviceId}/timewindow | Retrieve audit record based on Audit table primary key |

# Server-side Validation

The validation of input data is a common requirement in most applications, and the de-facto standard for performing validation is Hibernate Validator, the Bean Validation framework’s reference implementation. It ensures that the properties of a domain object meet specific criteria using:

* Built-in constraint annotations such as @NotNull, @Min, @Max and many, many more.
* Custom validation annotation, where in constraint annotation are applied on a new annotation
* Custom constraint validator for handling advanced logic, for example, to restrict a field to a specific list

**Usage**

* A common library containing custom validation annotataions. For e.g. validation of IP address
* Include the jar in garage-specific microservices
* Use the custom build validation annotation in microservices
* Any custom validations required by the microservices should be developed in the common library

**Implementation approach**

1. The validation starter adds in a dependency to a compatible version of hibernate validator, which is the most widely used implementation of the Bean Validation specification.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-validation</artifactId>

</dependency>

1. Add in the required validation constraints on the domain class. Bean Validation provides many handy constraints such as @Email, @Future, @Pattern, @Size, and many more. Custom validation annotations and constraint validators can also be created and used.

@Data //Lombok project usage  
@Entity  
public class Inventory implements Serializable {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private int id;

**@NotBlank**(message = “Product Code is mandatory”)  
 private String productCode;

**@Pattern(regexp = “^[A-Za-z]\*$”, message = “Must be alphabets only”)**

Private String manufacturer;

}

1. In the layer that assigns values to an object, the object can be passed into a Validator which checks if the constraints are satisfied. When Spring Boot finds an argument annotated with @Valid, it automatically bootstraps the Hibernate Validator and validates the argument.

@RestController

public class InventoryController {

@PostMapping("/products")

ResponseEntity<Inventory> addProduct(@**Valid** @RequestBody Inventory inventory) {

return new ResponseEntity<>(inventoryService.add(inventory), HttpStatus.CREATED);

}

}

When the target argument fails to pass the validation, Spring Boot throws a MethodArgumentNotValidException exception, which is detailed out in the next section.

1. For validation of method parameter such as path variables, that are of types int, Integer, String;

* Constraint validation is added to the method parameter
* Spring’s @Validated annotation is added at class level for Spring to evaluate constraint annotations on method parameters

@RestController

**@Validated**

public class InventoryController {

@GetMapping("/products/{Id}")

ResponseEntity<Inventory> getInventoryById(@PathVariable(“Id”)**@NotNull** int id) {

return new ResponseEntity<>(inventoryService.findById(id), HttpStatus.OK);

}

}

In contrast to validation on Request Body, a failed validation will trigger a ConstraintViolationException instead of a MethodArgumentNotValidException. Spring does not register a default exception handler for this exception, so it will by default cause a response with HTTP status 500 (Internal Server Error).

1. Instead of (or additionally to) validating input on the controller level, data can be validated to any Spring components. This is done with a combination of the @Validated and @Valid annotations. By default, Spring Data uses Hibernate underneath, which supports Bean Validation out of the box.

Though, Persistence Layer isn’t the right Place for Validation. The validation should always be applied server side and be the first thing executed, therefore input to service layer should be valid already.

Even though validations can be applied on the client side to save the server call, it still must be executed server side. The reason is that the server can be reached without the actual webpage. So never rely on client side input validation only.

# Exception Handling

As Exception Handling is a cross-cutting concern better handled in dedicated code, the global @ExceptionHandler is to be used with the @ControllerAdvice annotation to throw business and technical related exceptions.

The @ControllerAdvice annotation allows to consolidate multiple @ExceptionHandlers into a single, global error handling component. That is, the @ControllerAdvice annotation provides a centralized place for the exceptions thrown from all of the Controllers.

This mechanism is simple but also very flexible. It provides:

* Full control over the body of the response as well as the status code
* Mapping of several exceptions to the same method, to be handled together, and
* It makes good use of the newer RESTful ResposeEntity response

**Usage**

* A common library containing possible exceptions
* Include the jar in microservices
* When the microservice throws exceptions, it will be caught by the library code, which will log the exception and return back to front-end a standard error response
* All generic exceptions required by the microservices, should be developed in the common library

Response Exception message returned to frontend

Some of the possible exception handling scenarios are listed below.

1. When a validation fails, a meaningful error message is to be returned to the client in order to enable the client to display a helpful error message to the user. A data structure should be returned containing an error message for each validation that failed.

A global ControllerAdvice is created that handles all ConstraintViolationExceptions that bubble up to the controller level. In order to catch validation errors for request bodies as well, MethodArgumentNotValidExceptions are to be handled.

@ControllerAdvice

class ErrorHandlingControllerAdvice {

@ExceptionHandler(ConstraintViolationException.class)

@ResponseBody

public ResponseEntity<Object> onConstraintValidationException(

ConstraintViolationException e) {

ValidationErrorResponse error = new ValidationErrorResponse();

for (ConstraintViolation violation : e.getConstraintViolations()) {

error.getViolations().add(

new Violation(violation.getPropertyPath().toString(), violation.getMessage()));

}

return new ResponseEntity(error, HttpStatus.BAD\_REQUEST);

}

@ExceptionHandler(MethodArgumentNotValidException.class)

@ResponseBody

public ResponseEntity<Object> onMethodArgumentNotValidException(

MethodArgumentNotValidException e) {

ValidationErrorResponse error = new ValidationErrorResponse();

for (FieldError fieldError : e.getBindingResult().getFieldErrors()) {

error.getViolations().add(

new Violation(fieldError.getField(), fieldError.getDefaultMessage()));

}

return new ResponseEntity(error, HttpStatus.BAD\_REQUEST);

}

1. Each of the other business exceptions that are thrown can be handled in a similar manner.

@ExceptionHandler(Exception.class)

public ResponseEntity<Object> handleAllExceptions(Exception ex, WebRequest request) {

return new ResponseEntity(ex.getMessage(), HttpStatus.INTERNAL\_SERVER\_ERROR);

}

@ExceptionHandler(DuplicateRecordException.class)

public ResponseEntity<Object> handleDuplicateRecordException(DuplicateRecordException ex, WebRequest request) {

return new ResponseEntity(ex.getMessage(), HttpStatus.CONFLICT);

}

@ExceptionHandler(EmployeeNotFoundException.class)

public ResponseEntity<Object> handleEmployeeNotFoundException(EmployeeNotFoundException ex, WebRequest request) {

return new ResponseEntity(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

@ExceptionHandler(NoEmployeeRecordsException.class)

public ResponseEntity<Object> handleNoEmployeeRecordsException(NoEmployeeRecordsException ex, WebRequest request) {

return new ResponseEntity(ex.getMessage(), HttpStatus.NOT\_FOUND);

}

In Spring 5, the ResponseStatusException class has been introduced, whereby an instance of it can be created providing an HttpStatus and optionally a reason and a cause. Its main benefits are:

* Excellent for prototyping: A basic solution can be implemented quite fast
* One type, multiple status codes: One exception type can lead to multiple different responses. This reduces tight coupling compared to the @ExceptionHandler
* More control over exception handling since the exceptions can be created programmatically

However it comes with tradeoffs:

* There's no unified way of exception handling: It's more difficult to enforce application-wide conventions, as opposed to @ControllerAdvice which provides a global approach
* Code duplication: Code replication in multiple controllers

It is possible to combine different approaches within a single application. For example, implement a @ControllerAdvice globally, but also ResponseStatusExceptions locally. However, it needs to be considered with caution if the same exception is to be handled in multiple ways.

A possible convention is to handle one specific kind of exception always in one way.

## 12-Factor Apps Design Principles for Microservices:

|  |  |  |
| --- | --- | --- |
| Design principle | Criticality | Description |
| Codebase | Mandatory | All application code for a service lives in one repository |
| Dependencies | Mandatory | Explicitly declare and isolate dependencies |
| Config | Mandatory | Store config in the config server |
| Backing Services | Mandatory | Treat backing services as attached resources |
| Build, release, run | Recommended | Strictly separate build and run stages |
| Processes | Mandatory | Execute the app as one or more stateless processes |
| Port binding | Mandatory | Export services via port binding |
| Concurrency | Recommended | Scale out via the process model |
| Disposability | Recommended | Maximize robustness with fast startup and graceful shutdown |
| Dev/prod parity | Recommended | Keep development, staging, and production as similar as possible |
| Logs | Mandatory | Treat logs as event streams |
| Admin processes | Recommended | Run admin/management tasks as one-off processes |

Codebase — one codebase tracked in revision control, many deploys

Criticality: Mandatory

All application code for a service lives in one repository. A codebase is run by developers on their local machines, and deployed to any number of other environments, like a set of testing machines, and the live production servers.

Dependencies — explicitly declare and isolate dependencies

Criticality: Mandatory

All the environments code runs in need to have some dependencies, like a database, or an image-processing library, or a command-line tool. Applications must not assume those things will be in place on a given machine. Most languages and frameworks provide a natural way to do this. List all the versions of all the libraries and when the code is deployed, a command is run to download all the right versions and put them in place.

Config — Store config in the environment

Criticality: Mandatory

The twelve-factor compliant app stores config in environment variables. Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms such as Java System Properties, they are a language- and OS-agnostic standard.

Apps sometimes store config as constants in the code. This is a violation of twelve-factor, which requires strict separation of config from code. Config varies substantially across deploys, code does not.

Backing Services — Treat backing services as attached resources

Criticality: Mandatory

A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as [MySQL](http://dev.mysql.com/) or [CouchDB](http://couchdb.apache.org/)), messaging/queueing systems (such as [RabbitMQ](http://www.rabbitmq.com/) or Kafka), SMTP services for outbound email (such as [Postfix](http://www.postfix.org/)), and caching systems (such as Redis).

Backing services like the database are traditionally managed by the same systems administrators as the app’s runtime deploy. In addition to these locally-managed services, the app may also have services provided and managed by third parties. Examples include SMTP services (such as [Postmark](http://postmarkapp.com/)), metrics-gathering services (such as [New Relic](http://newrelic.com/) or Loggly), binary asset services (such as [Amazon S3](http://aws.amazon.com/s3/)), and even API-accessible consumer services (such as [Twitter](http://dev.twitter.com/), [Google Maps](https://developers.google.com/maps/), or [Last.fm](http://www.last.fm/api)).

The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the [config](https://12factor.net/config). A [deploy](https://12factor.net/codebase) of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as [Amazon RDS](http://aws.amazon.com/rds/)) without any changes to the app’s code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change. Each distinct backing service is a resource.

This is another case where defining dependencies cleanly keeps the system flexible and each part is abstracted from the complexities of the other, a core tenet of good architecture.

Resources can be attached and detached to deploys at will. For example, if the app’s database is misbehaving due to a hardware issue, the app’s administrator might spin up a new database server restored from a recent backup. The current production database could be detached, and the new database attached – all without any code changes.

Build, release, run — strictly separate build and run stages

Criticality: Recommended

The process of turning the code into a bundle of scripts, assets and binaries that run the code is the build. The release sends that code to a server in a fresh package together with the nicely separate config files for that environment. Then the code is run so the application is available on those servers. The idea here is that the build stage does a lot of heavy lifting, and developers manage it. The run stage should be simple and bullet-proof so that the team can be confident knowing that the application is running well, and that if a machine gets restarted (say, a power failure happens) that the app will start up again on launch without the need for human intervention.

Processes — execute the app as one or more stateless processes

Criticality: Mandatory

It is likely that an application will be running on many servers because that makes it more fault tolerant and support more traffic. As a rule, each of those instances of running code should be stateless. In other words, the state of the system is completely defined by the databases and shared storage, and not by each individual running application instance.

For example, a signup workflow, where a user has to enter 3 screens of information to create their profile. One (wrong) model would be to store each intermediate state in the running code and direct the user back to the same server until the signup process is complete. The right approach is to store intermediate data in a database or persistent key-value store such as Gemfire, so even if the web server goes down in the middle of the user’s signup, another web server can handle the traffic, and the system is none-the-wiser.

Port binding — Export services via port binding

Criticality: Mandatory

The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port and listening to requests coming in on that port.

In a local development environment, the developer visits a service URL like http://localhost:5000/ to access the service exported by their app. In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.

This is typically implemented by using [dependency declaration](https://12factor.net/dependencies) to add a webserver library to the app, such as [Tornado](http://www.tornadoweb.org/) for Python, [Thin](http://code.macournoyer.com/thin/) for Ruby, or [Jetty](http://www.eclipse.org/jetty/) for Java and other JVM-based languages, Express for Node.js. This happens entirely in user space, that is, within the app’s code. The contract with the execution environment is binding to a port to serve requests.

Concurrency — Scale out via the process model

Criticality: Recommended

When running the code, the idea is that many little processes are handling specific needs. Therefore, there might be dozens of handlers ready to process web requests, and another dozen to handle API calls for enterprise users. Still another half-dozen processing background welcome-emails going to new users or sending tweets for users sharing things on a social media service.

By keeping all these small parts working independently and running them as separate processes (in a low-level technical sense), the application will scale better. More stuff is done concurrently, by smoothly adding additional servers or additional CPU/RAM and taking full advantage of it with more of these small, independent processes.

# 

Disposability — Maximize robustness with fast startup and graceful shutdown

Criticality: Recommended

When new code is deployed, ideally the new version should launch right away and start to handle traffic. If an application has to do 20 seconds of work (say, loading giant mapping files into RAM) before it is ready to handle real traffic, makes it harder to rapidly release code, and more churn is introduced on the system to stop/start independent processes.

With the proliferation of so many third party libraries in today’s software systems, sub–1-second startup times are less and less common. However, beyond loading code, an application should have everything it needs waiting in high-speed databases or caches, so it can start up snappily and be ready to serve requests.

Further, the application should be robust against crashing. Meaning, if it does crash, it should always be able to start back up cleanly. One should never do any mandatory “cleanup” tasks when the app shuts down that might cause problems if they failed to run in a crash scenario.

Dev/prod parity — Keep development, staging, and production as similar as possible

Criticality: Recommended

It has become in vogue in recent years to have a much more rapid cycle between developing a change to an app and deploying that change into production. For many companies, this happens in a matter of hours. In order to facilitate that shorter cycle, and the risk that something breaks when entering production, it is desirable to keep a developer’s local environment as similar as possible to production.

This means using the same backing services, the same configuration management techniques, the same versions of software libraries, and so on.

This is often accomplished by letting developers use a tool like [Vagrant](http://www.vagrantup.com/) to manage their own personal virtual server that is configured just like production servers.

Logs — Treat logs as event streams

Criticality: Mandatory

A twelve-factor app should not concern itself with routing or storage of its output stream. It should not attempt to write to or manage log files. Instead, each running process writes its event stream, unbuffered, to stdout. During local development, the developer will view this stream in the foreground of their terminal to observe the app’s behavior.

In staging or production deploys, each process’ stream will be captured by the execution environment, collated together with all other streams from the app, and routed to one or more final destinations for viewing and long-term archival. These archival destinations are not visible to or configurable by the app, and instead are completely managed by the execution environment. Open-source log routers (such as Logplex and [Fluent](https://github.com/fluent/fluentd)) are available for this purpose.

The event stream for an app can be routed to a file or watched via real-time tail in a terminal. Most significantly, the stream can be sent to a log indexing and analysis system such as [ELK](http://www.splunk.com/), or a general-purpose data warehousing system such as [Hadoop/Hive](http://hive.apache.org/). These systems allow for great power and flexibility for introspecting an app’s behavior over time, including:

* Finding specific events in the past.
* Large-scale graphing of trends (such as requests per minute).
* Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

Admin processes — Run admin/management tasks as one-off processes

Criticality: Recommended

There may be need of lots of one-off administrative tasks once an app is live. For example, doing data cleanup on bad data discovered, running analytics, or turning on and off features for A/B testing.

Usually a developer will run these tasks, and when they do, they should be doing it from a machine in the production environment that is running the latest version of the production code. In other words, run one-off admin tasks from an identical environment as production. Do not run updates directly against a database; do not run them from a local terminal window.

# UI/UX Mobile Standards and Guidelines

Installation Guide: It helps developer to install the angular application and start working on the development

**Setup the development environment for Angular**

**Angular CLI: -** The Angular CLI is a command line interface tool that can create a project, add files, and perform a variety of ongoing development tasks such as testing, bundling, and deployment.

**NodeJS :** Node installation version should be 5.X + otherwise it throw an error

Install angular cli globally

npm install -g @angular/cli (version 6.0.8)

**Create a new project**

Open a terminal window type cmd on RUN.

Create a new project by running the following command:

ng new my-app

The Angular CLI installs all the necessary npm packages, creates the project files, and populates the project.

**Run the application**

cd my-app ng

serve –open

The ng serve command launches the server, watches your files, and rebuilds the app.

Using the --open (or just -o) option will automatically open your browser on <http://localhost:4200/>.

## Installation of ng-bootstrap

Ng-Bootstrap contains a set of native Angular directives based on Bootstrap’s markup and CSS. As a result no dependency on jQuery or Bootstrap’s JavaScript is required.

**Run the command**

npm install --save @ng-bootstrap/ng-bootstrap

Once the module is installed by npm, need to be entered in main.ts module

import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

import module where we need use it

import {NgbModule} from '@ng-bootstrap/ng-bootstrap';

@NgModule({

declarations: [OtherComponent, ...],

imports: [NgbModule, ...]

})

export class OtherModule {

}

Bootstrap grid is to make the page responsive

// Small devices (landscape phones, 576px and up)

@media (min-width: 576px) { ... }

// Medium devices (tablets, 768px and up)

@media (min-width: 768px) { ... }

// Large devices (desktops, 992px and up)

@media (min-width: 992px) { ... }

// Extra large devices (large desktops, 1200px and up)

@media (min-width: 1200px) { ... }

## Folder structure

|-- app  
 |-- modules  
 |-- Login  
 |-- [-] components  
 |-- [+] pages  
 |-- login-routing.module.ts  
 |-- login.module.ts  
 |-- core  
 |-- [+] authentication  
 |-- [+] header  
 |-- [+] footer  
 |-- [+] http  
 |-- [+] services  
 |-- [+] charts  
 |-- core.module.ts  
   
 |  
 |-- shared  
 |-- [+] components  
 |-- [+] directives  
 |-- [+] pipes  
 |  
 |-- [+] configs  
|-- assets  
 |-- css   
 |-- \_base.css  
 |-- styles.css

## Tools and plugins :-

**Charts :-** For charting we will use primeng library its open source and easy customizable below is the URL

<https://primefaces.org/primeng/#/chart/pie>

**Data grid**

https://primefaces.org/primeng/#/treetable

below is the article to configure primeNg, its available at npm

below are the commands

npm install primeng --save

npm install primeicons --save

for more details below’s the link for documentation

<https://primefaces.org/primeng/#/setup>

**Build with angular**

ng build –aot

ng serve –aot

Aot compiles the app at build time

## Code review and quality checks

For angular type script should be checked and review by ts lint, install tslint by using installing the below commands on the local machine, code should be check and reviewed by ts lint compiler.

npm install tslint typescript –save-dev

*# or*

yarn add tslint typescript --dev

## Accessibility

To implement accessibility with angular use angular services “ngAria”, ngAria hooks into standard AngularJS directives and quietly injects accessibility support into your application at runtime.

Install ngAria

npm install --save angular-aria@X.Y.Z

for table below’s the example

<table role="presentation">

<thead>

<th>for table header</th>

</thead>

<tbody><td></td></tbody>

Site should be accessible by the keyboard user can navigate with mouse as well as keyboard.

Image tag should be define should contain alt and title tag

Header should be define in a proper order should start with h1 to h6.

Hyperlinks should be define properly with proper description.

Examples

<img src=”” alt=”information about image” title=”image title” />

<a href=”” title=”about the link”></a>