



# National Urban Digital Mission

Building cities that work for people

## Customizing backend Service

## Outline

1. Writing a new Service
2. Enhancing existing Service
3. Customizing PDF Receipts & Certificates
4. Persister and Indexer changes
5. Writing a new consumer

## Writing a new Service

















APIs developed on UPYOG follow certain conventions and principles. Follow the below principles when writing a new service.

- Always define the Yaml for your APIs as the first thing using Open API 3 Standard (<https://swagger.io/specification/>)
- APIs path should be standardized as follows:
  - **/\_{service}/{entity}/{version}/\_create**: This endpoint should be used to create the entity
  - **/\_{service}/{entity}/{version}/\_update**: This endpoint should be used to edit a entity which is already existing
  - **/\_{service}/{entity}/{version}/\_search**: This endpoint should be used to provide search on the entity based on certain criteria
  - **/\_{service}/{entity}/{version}/\_count**: This endpoint should be provided to give count of entities that match a given search criteria
- Always use POST for each of the endpoints
- Take most search parameters in POST body only
- If query params for search needs to be supported then make sure to have same parameters in POST body also and POST body should take priority over query params
- Provide additionalDetails objects for **\_create** and **\_update** APIs so that custom requirements can use these fields
- Each API should have a [RequestInfo](#) object in request body at the top level
- Each API should have a [ResponseInfo](#) object in response body at the top level
- Mandatory fields should be minimum for the APIs.
- minLength and maxLength should be defined for each attribute
- Read only fields should be called out
- Use common models already available in the platform in your APIs. Ex -
  - [Address](#)
  - [User](#)(Citizen or Employee or Owner)
  - [Role](#)

- [AuditDetails](#)
- [ErrorRes](#) (Response sent in case of errors)
- TODO: Add all the models here
- For receiving files in an API, don't use binary file data. Instead accept the file store ids
- If there is only one file to be uploaded and no persistence is needed, and no additional json data is to be posted, you can consider using direct file upload instead of using filestore id

## Enhancing existing Service

In UPYOG, most of the applications are RESTful services which follow the below project structure

- ▼ src
  - ▼ main
    - ▼ java
      - ▼ org.egov.pgr
        - > config
        - > consumer
        - > producer
        - > repository
        - > service
        - > util
        - > validator
      - ▼ web
        - > controllers
        - > models
      -  PGRApp
    - ▼ resources
      - ▼ db
        - ▼ migration.main
          -  V20200717133931\_\_create\_table.sql
          -  V20200810112036\_\_add\_index.sql
          -  Dockerfile
          -  migrate.sh
          -  application.properties
          -  contractForCodeGen.yml
          -  mockData.json
          -  pgr-services-persister.yml
          -  swagger-contract.yml
          -  workflowConfig.json
  - > test
- > target
  -  CHANGELOG.md
  -  LOCALSETUP.md
  -  pgr-services.iml
  -  pom.xml
  -  README.md

- **The structure consists of the following packages:-**

**Config:** This package consists of configuration code and data for the application. The class defined here will read the value from the property file. While enhancing the service suppose new values are added in the property file those values are accessed through the config file defined in this package.

- **Consumer:** This package consists of the Kafka Consumer program, which consumes the published messages from the producer. The consumer consumes the message by subscribing to the topic. A new consumer must be added to this package only.
- **Producer:** This Package consists of the Kafka producer program, which pushes the data into the topic and the consumer consumes the message by subscribing to that same topic.
- **Repository:** This package is a data access layer, which is responsible for retrieving data for the domain model. It only depends on the model layer. The classes define here could get data from the database or other Microservices by RESTful call. Any update in the data retrieval query has to be done in the classes present in the repository package.
- **Service & Util:** This package is a service layer of the application. The implementation classes are defined here which consists of the business logic of the application or functionality of the APIs. Any modification in the functionality of API or business logic has to be done in the classes present in this package.  
**E.g:-** you have an API call which you need to modify then you create one more class or util function in a new file in service or util package and just call that function to do the work you want in the API code instead of writing code directly in the API file.
- **Validation:** This package consists of a validation class for better processing of the application. While enhancing a service if a new validation method is required it has to be added to this package.
- **Models:** The various models of the application are organised under the *models* package. This package is a domain module layer, which has domain structs. Pojo's classes will be present here and these classes will be updated on the update of the contract of the service. And accordingly rowmapper class needs to update.
- **Controller:** The most important part is the controller layer. It binds everything together right from the moment a request is intercepted until the response is prepared and sent back. The controller layer is present in the controller package, the best practices suggest that we keep this layer versioned to support multiple versions of the application. It expects to have different versions having different features.

- **DB.migration.main:** This package consists of flyway migration scripts related to setting up an application, for example, database scripts. For any changes in the database table of this service, a script has to be added to this package.

- **Note:**

- 1) Never overwrite the previously created scripts always create a new one as per the requirements.

- 2) The file name should follow this naming convention

- V<timestamp>\_\_<purpose>.sql***

- ex: V20200717125510\_\_create\_table.sql***

Any changes here, if it is required to change the retrieval query then accordingly the classes present in the repository package have to update. And also the persister file associated with these services needs to be updated.

- **application.properties:** The application.properties file is just straightforward key-value stockpiling for configuration properties. You can package the configuration file in your application jar or put the file in the file system of the runtime environment and load it on Spring Boot startup.

This file contains the value of

- server port
  - server context path
  - Kafka server configuration value
  - Kafka producer and the consumer configuration value
  - Kafka topic
  - External service path and many more.

Whatever changes are done in application.properties file the same changes need to be reflected in the values.yaml file (for reference only) of a particular service.

- After enhancing the service, the details about the new feature must be mentioned in README.md, LOCALSETUP.md and [CHANGELOG](#).md file. And the version of service in the pom.xml file need to be incremented.

**Eg:** 1.0.0 to 1.0.1 or 1.0.0 to 1.1.0 depend upon the level of the enhancement

## Customizing PDF Receipts & Certificates

The objective of PDF generation service is to bulk generate pdf as per requirement.

### Configuration Details

**Config file:** A json config file which contains the configuration for pdf requirement. For any pdf requirements we have to add two config file to the service.

- **Format Config file:** This config file define the format of PDF. In format config we define the UI structure ex: css, layout etc. for pdf as per PDFMake syntax of pdf. In PDF UI, the places where values are to be picked from request body are written as “{{variableName}}” as per ‘mustache.js’ standard and are replaced by this templating engine. Ex:  
<https://github.com/nugp-digit/nugp-configs/tree/master/configs/pdf-service/format-config>

PDF generation service read these such files at start-up to support PDF generation for all configured module.

**Data Config file:** This file contains mapping to pick data from request body, external service call body if there is any and the variable which defines where this value is to be replaced in format by the templating engines (mustache.js). The variable which is declared in format config file must be defined in data config file. Ex:

<https://github.com/nugp-digit/nugp-configs/tree/master/configs/pdf-service/data-config>

## Persister and Indexer changes

Persister service provides a framework to persist data in a transactional fashion with low latency based on a config file. Removes repetitive and time-consuming persistence code from other services.

Provides a framework to persist data in transactional fashion with low latency based on a config file. Removes repetitive and time consuming persistence code from other services

Refer: <https://github.com/nugp-digit/nugp-configs/tree/master/configs/egov-persister>

### Functionality:

- [Persist](#) data asynchronously using kafka providing very low latency
- Data is persisted in batch
- All operations are transactional
- Values in prepared statement placeholder are fetched using JsonPath

- Easy reference to parent object using '{x}' in jsonPath which substitutes the value of the variable x in the JsonPath with value of x for the child object.(explained in detail below in doc)
- Supported data types **ARRAY("ARRAY")**, **STRING("STRING")**, **INT("INT")**,**DOUBLE("DOUBLE")**, **FLOAT("FLOAT")**, **DATE("DATE")**, **LONG("LONG")**,**BOOLEAN("BOOLEAN")**,**JSONB("JSONB")**

## How to Use:

1. **Configuration:** Persister uses configuration file to persist data. The key variables are described below:
  - **serviceName:** Name of the service to which this configuration belongs.
  - **description:** Description of the service.
  - **version:** the version of the configuration.
  - **fromTopic:** The kafka topic from which data is fetched
  - **queryMaps:** Contains the list of queries to be executed for the given data.

**query:** The query to be executed in form of prepared statement:

1. **basePath:**
2. **jsonMaps:** Contains the list of jsonPaths for the values in placeholders.
  1. **jsonPath:** The jsonPath to fetch the variable value.

## Bulk Persister:

To persist a large quantity of data bulk setting in persister can be used. It is mainly used when we migrate data from one system to another. The bulk persister have the following two settings:

<code>persister.bulk.enabled</code>	false	Switch to turn on or off the bulk kafka consumer
-------------------------------------	-------	--



<code>persister.batch.size</code>	100	The batch size for bulk update
-----------------------------------	-----	--------------------------------

Any kafka topic containing data which has to be bulk persisted should have '-batch' appended at the end of topic name example: [save-pt-assessment-batch](#)

**Indexer** uses a config file per module to store all the configurations pertaining to that module. Indexer reads multiple such files at start-up to support indexing for all the configured modules. In config we define source and, destination elastic search index name, custom mappings for data transformation and mappings for data enrichment. Below in the sample configuration for indexing TL application creation data into elastic search.

Refer: <https://github.com/nugp-digit/nugp-configs/tree/master/configs/egov-indexer>

### The configuration file contains following keys:-

Variable Name	Description
serviceName	Name of the module to which this configuration belongs.
summary	Summary of the module.
version	Version of the configuration.

mappings	List of definitions within the module. Every definition corresponds to one index requirement. Which means, every object received onto the kafka queue can be used to create multiple indexes, each of these indexes will need configuration, all such configurations belonging to one topic forms one entry in the mappings list. The keys listed henceforth together form one definition and multiple such definitions are part of this mappings key.
topic	Topic on which the data is to be received to activate this particular configuration.
configKey	Key to identify to what type of job is this config for. values: INDEX, REINDEX, LEGACYINDEX. INDEX: LiveIndex, REINDEX: Reindex, LEGACYINDEX: LegacyIndex.
indexes	Key to configure multiple index configurations for the data received on the particular topic. Multiple indexes based on different requirement can be created using the same object.
name	Index name on the elasticsearch. (Index will be created if it doesn't exist with this name.)
type	Document type within that index to which the index json has to go. (Elasticsearch uses the structure of index/type/docId to locate any file within index/type with id = docId)

id	Takes comma separated JsonPaths. The JSONPath is applied on the record received on the queue, the values hence obtained are appended and used as id for the record.
isBulk	Boolean key to identify whether the JSON received on the Queue is from a Bulk API. In simple words, whether the JSON contains a list at the top level.
jsonPath	Key to be used in case of indexing a part of the input JSON and in case of indexing a custom json where the values for custom json are to be fetched from this part of the input.
timestampField	JSONPath of the field in the input which can be used to obtain the timestamp of the input.
fieldsToBeMasked	A list of JSONPaths of the fields of the input to be masked in the index.
customJsonMapping	Key to be used while building an entirely different object using the input JSON on the queue
indexMapping	A skeleton/mapping of the JSON that is to be indexed. Note that, this JSON must always contain a key called "Data" at the top-level and the custom mapping begins within this key. This is only a convention to smoothen dashboarding on Kibana when data from multiple indexes have to be fetched for a single dashboard.

fieldMapping	Contains a list of configurations. Each configuration contains keys to identify the field of the input JSON that has to be mapped to the fields of the index json which is mentioned in the key 'indexMapping' in the config.
inJsonPath	JSONPath of the field from the input.
outJsonPath	JSONPath of the field of the index json.
externalUriMapping	Contains a list of configurations. Each configuration contains keys to identify the field of the input JSON that are to be enriched using APIs from the external services. The configuration for those APIs also is a part of this.
path	URI of the API to be used. (it should be POST/_search API.)
queryParam	Configuration of the query params to be used for the API call. It is a comma separated key-value pair, where key is the parameter name as per the API contract and value is the JSONPath of the field to be equated against this paramter.
apiRequest	Request Body of the API. (Since we only use _search APIs, it should be only RequestInfo.)
uriResponseMapping	Contains a list of configuration. Each configuration contains two keys: One is a JSONPath to identify the field from response, Second is also a JSONPath to map the response field to a field of the index json mentioned in the key 'indexMapping'.

mdmsMapping	Contains a list of configurations. Each configuration contains keys to identify the field of the input JSON that are to be denormalized using APIs from the mdms service. The configuration for those mdms APIs also is a part of this.
path	URI of the API to be used. (it should be POST/_search API.)
moduleName	Module Name from MDMS.
masterName	Master Name from MDMS.
tenantId	Tenant id to be used.
filter	Filter to be applied on the data to be fetched.
filterMapping	Maps the field of input json to variables in the filter
variable	Variable in the filter
valueJsonpath	JSONPath of the input to be mapped to the variable.

## Writing a new consumer

Kafka producer publishes messages on a given topic. Kafka Consumer is a program, which consumes the published messages from the producer. The consumer consumes the message by subscribing to the topic. A single consumer can subscribe to multiple topics. Whenever the topic receives a new message it can process that message by calling defined functions. The following snippet is a sample of code which defines a class called TradeLicenseConsumer which contains function called listen() which is subscribed to save-tl-tradelicense topic and calls a function to generate notification whenever any new message is received by the consumer.

```
@Slf4j
@Component
public class TradeLicenseConsumer {

    private TLNotificationService notificationService;

    @Autowired
```

```
public TradeLicenseConsumer(TLNotificationService notificationService) {  
    this.notificationService = notificationService;  
}  
  
    @KafkaListener(topics = {"save-tl-tradelicense"})  
    public void listen(final HashMap<String, Object> record,  
    @Header(KafkaHeaders.RECEIVED_TOPIC) String topic) {  
        notificationService.sendNotification(record);  
    }  
}
```

@KafkaListener annotation is used to create a consumer. Whenever any function has this annotation on it, it will act as a kafka consumer and the message will go through the flow defined inside of this function. The topic name should be picked up from application.properties file. This can be done as showed below:

```
@KafkaListener(topics = {"${persister.update.tradelicense.topic}"})
```

where persister.update.tradelicense.topic is the key for the topic name in application.properties

Whenever any new message is published on this topic the message will be consumed by the listen() function and will call the function sendNotification() with the message as the argument. The deserialization is controlled by the following two properties in application.properties:

```
spring.kafka.consumer.value-deserializer
```

```
spring.kafka.consumer.key-deserializer
```

The first property sets the deserializer for value while the second one sets it for key. Depending on the deserializer we have set we can expect the argument in that format in our consumer function. For example if we set the value deserializer to HashMapDeserializer and key deserializer to string like below:

```
spring.kafka.consumer.value-deserializer=org.egov.tracer.kafka.deserializer.HashMapDeserializer
```

```
spring.kafka.consumer.key-deserializer=org.apache.kafka.common.serialization.StringDeserializer
```

Then we can write our consumer function expecting HashMap as argument like below:

```
public void listen(final HashMap<String, Object> record){...}
```