< **Browse Recipes**

API First Development

api swagger yaml json api first design first contract first

An extension of "Design-First" methodology is "Contract-First" where the developer defines the contract of the API before the implementation is coded. Just-in-time and just-enough design is critical to agility. Applying design-first / contact-first methodology to the world of API development results in "API-First" development. The basic idea is that you create an API contract before implementing the full code and plumbing. The human-readable API specification becomes the quick first deliverable that allows fast, quick feedback from various stakeholders. These rapid iterations are a game changer because changing the API specification takes minutes not hours or days! Once the team determines that the API specification is "good enough" a full maven Spring Boot Application can be generated. Many other languages are supported as well – more on that later. The developer codes the business logic implementation in the server stub. While the developer is coding, the QA teams can be handed the spec as well so they get a head start on service testing.

API First with Open API Spec 2.0

The Open API Specification is a language and vendor neutral API specification standard. There are many permutations of tools, technologies, languages to approach API design and development, most new server-side development happens in the Java programming language. This document will walk through a real world scenario of putting 'API-First in Practice' with Open API for the first time. The goal is to provide a clear path to achieving agile API development and being effective at it.

Read about the **Open API Initiative**.

API-First Development Workflow

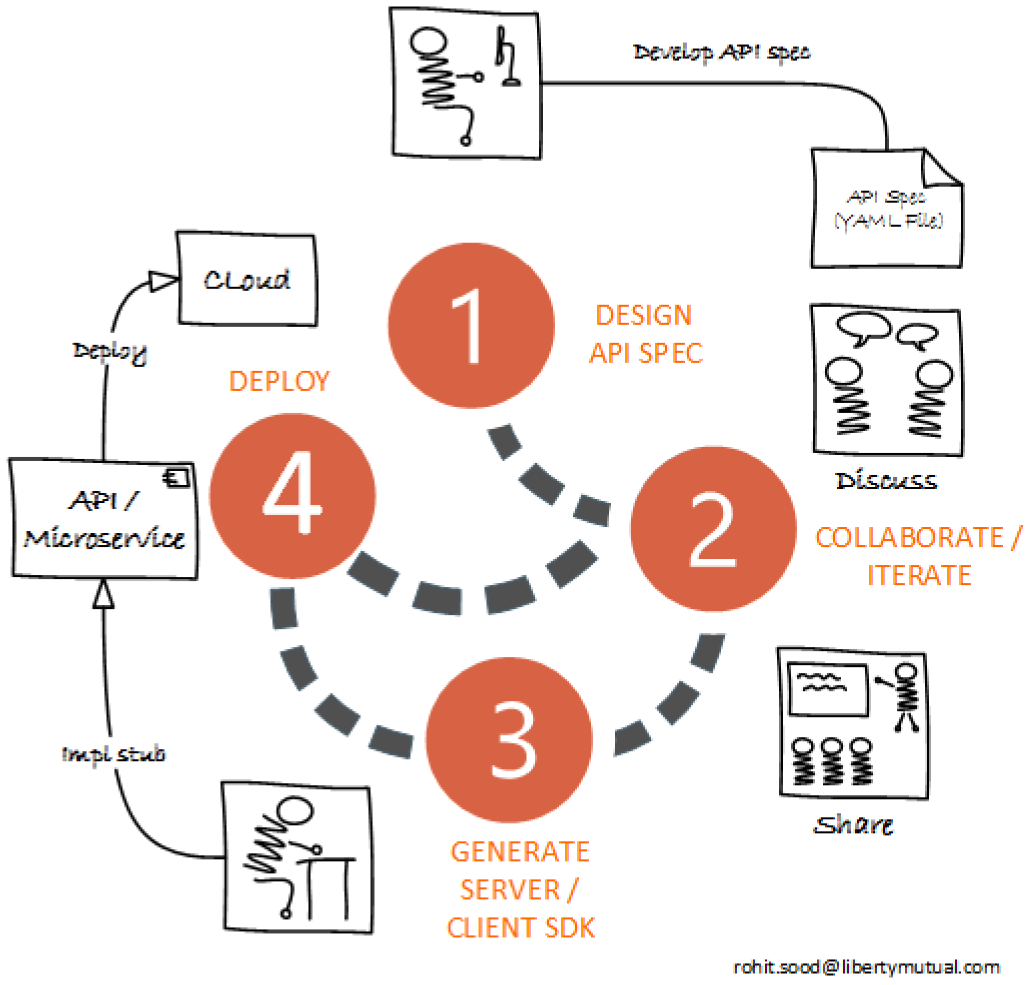
There are 4 high-level steps:

1. Create API
2. Collaborate

2. Collaborate

3. Generate Server

4. Deploy to Cloud



The first step is essentially starting the API development using Open API Spec 2.0. Begin working on evolving and iterating the API specification by collaborating with the product owner, service consumers and other stakeholders. Once the team is comfortable that the API specification is 'good enough' - generate the server in Spring Boot. Deploy the app to the cloud or locally. At this point the round-trip engineering starts. All updates to the API in the code automatically updates the API specification that is co-deployed via annotations.

Step 1: Create the API

Use Open API 2.0 specification standard and the YAML format to describe the API as follows:

The high-level context of the API:

```
swagger: '2.0'
info:
  title: LOB Request Write-Side API
  description: Transaction Request aggregate Write-Side API to manage the
overall transaction context. /transactions is the aggregate root through
which this API will be modeled. Basically, the business user will start and
manage transactions explicitly (or implicitly). Past-tense events will be
published after the transaction completes. The model depicted in this API
is exposed for write operations only per the CQRS pattern, it is possible
that the entities encapsulated within the service implementation are
further enriched to perform the business logic that's expected.
  version: "1.0.0"
schemes:
  - https
basePath: /v1
produces:
  - application/json
  - application/xml
```

Expect the consumers of the API to read the info sections, paths, verbs and model information to fully understand the API and how to interact with it.

- 'swagger' - This snippet is the defacto standard - the swagger version (which is Open API Spec 2.0)
- 'info' - this specifies high level information about the api
- 'title' - Ensure that the title is clear and broad enough that it covers the API as whole.

Defining the Paths

```
paths:
  /transactions:
    post:
      summary: |
        Starts a transaction. It can start either a Multi-line App or a
Mono-line App Transaction.
        All  types of transaction types are supported via this API.
Business rules/logic will be executed to ensure this operation can be
executed.
        Transactions are a container that tracks and manages the overall
processing.
      description: |
        This is the aggregate root of the API. If only one LOB request
exists under the transaction then a monoline app request is created.
        A multi-line app request contains more than one 'lob requests'.
Each 'lob request' has an associated policy app/lob. This end-point
contains a single request or multiple requests - and executes the
StartRequestCommand.
        The Multiline Request endpoint creates a new multi-line app request
(and child requests if provided). This API should be used for all requests
(monoline app or multi-line apps). This is the  Multi-Line App Request
Aggregate Root.

      consumes:
        - application/json
        - application/xml
      produces:
        - application/json
        - application/xml

      parameters:
        - in: body
          name: transaction
          description: |
            If a transaction_id is not provided, one will be created by the
service.
            Account is optional.
            Agency is optional.
            These can be added to the transaction later.
          required: true
```

```
            schema:
              $ref: "#/definitions/Transaction"
        tags:
          - Transaction
        responses:
          200:
            description: Successful addition of a transaction.
          400:
            description: Invalid request received - please check the API
Contract.
          403:
            description: Forbidden. Client is not permitted to perform this
operation.
          409:
            description: Conflict. Resource already exists.

          default:
            description: Unexpected error
            schema:
              $ref: '#/definitions/Error'
```

The path parameters define the shape of the API via URIs. Each URI definition has multiple HTTP verbs that can act on it.

- POST: has a summary of what the API expects and what it responds with on a creation.
- Summary and description fields are important to document so that the consumer of the API knows how to use it.
- Consumes & Produces - specifies the data formats the API will accept and produce. the Parameters specify where and how the data will enter the API e.g. body, path, header, query etc.
- tags - these are optional but help in the documentation of the API.
- responses - the responses specify how the API responds and what's in the response body.

- definitions - Define the models in the definitions - this section is typically at the end of the API spec file.

```
definitions:
  Transaction:
    type: object
    properties:
      transactionId:
        type: string
        description: Unique identifier representing a specific transaction.
      status:
        type: string
        description: The status of the transaction.
      transactionType:
        type: string
        description: The activity for the User Intent
      accountNumber:
        $ref: '#/definitions/Account'
      agency:
        $ref: '#/definitions/Agency'
      lobRequests:
        type: array
        items:
          $ref: '#/definitions/LOBRequest'

  LOBRequest:
    type: object
    properties:
      lobCd:
        type: string
        description: Line of business code. This is unique per transaction.
      status:
        type: string
        description: Status of the LOBRequest.
      startDate:
        type: string
        format: date
        description: The date when the request was started.
      completionDate:
        type: string
        format: date
        description: The date when the request was completed.

      policy:
        $ref: '#/definitions/Policy'
      underwritinginstructions:
        type: array
        items:
          $ref: '#/definitions/UnderwritingInstruction'

  Policy:
    type: object
    properties:
      policy_id:
        type: string
        description: This is a unique identified (GUID) that represents the
policy aggregate.
      lobCd:
        type: string
        description: Line of business code.
      policyTerm:
        type: number
        description: Represents the policy term number.

  UnderwritingInstruction:
    type: object
    properties:
      instructionType:
        type: string
        description: The instruction type.
      instructionDate:
        type: string
        format: date-time
        description: Represents the instruction date-time.
      instructionRemark:
        type: string
        description: Represents the instruction date-time.

  Account:
      type: object
      properties:
        accountNumber:
          type: string
          description: The account number for the account.
  Agency:
      type: object
      properties:
        agencyNumber:
          type: string
          description: The agency number for the account.

  Error:
    type: object
    properties:
      code:
        type: integer
        format: int32
      message:
        type: string
      fields:
        type: string
```

You can start the Open API 2.0 Editor **here** It will start up with an example. Alternatively, you can import another YAML specification - see the YAML specifications in the references section.

Step 2: Evolve the API (Collaborate)

Step 2: Evolve the API (Collaborate)

As the API is developing - the specification file can and should be shared with other collaborators for review. The main idea is to ensure that the paths and models are good enough. In other words, you don't want to strive for perfection rather a few brief meetings with the stakeholders to ensure that there is a common understanding and consensus. The idea is that API-first allows for rapid iterations without changing the code/plumbing in the initial phases.

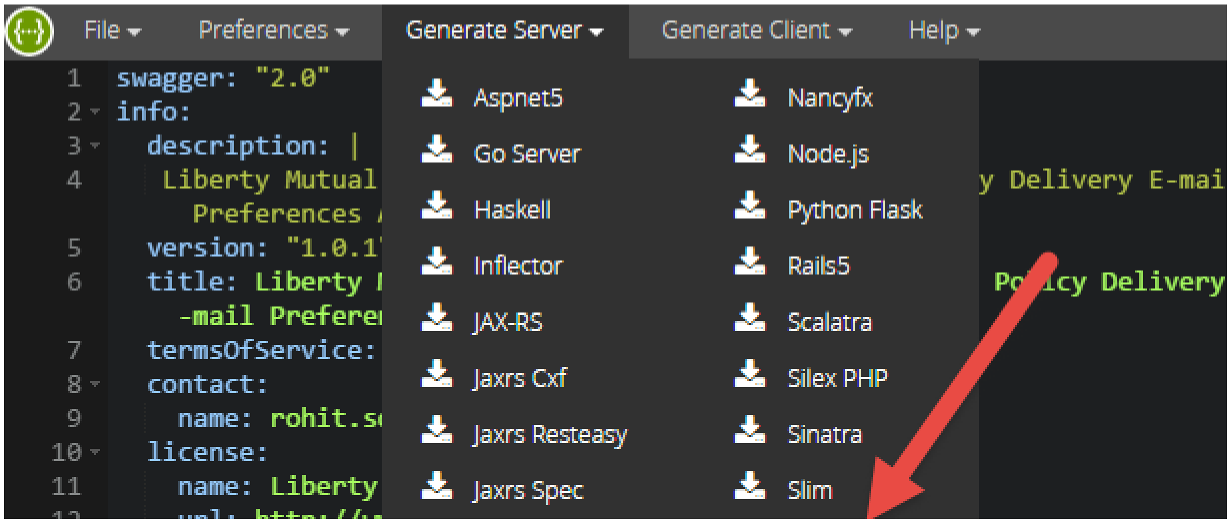
The first-cut of the API spec should minimally have the RESTful paths, responses and high level response objects. The API should be small-enough that it fits within a person’s head. Of course, good understanding of domain driven design, API design, REST design and Open API spec is required.

This design spec can save you hours of implementation and refactoring effort in the plumbing of the code. This allows subject matter experts to discuss the API and make changes. Quality engineers can get involved and begin early test design and development. Architects can begin to think about system architecture implications on sourcing of data. Data analysts can begin thinking about data fields and standards alignment where appropriate. This spec should become the camp fire across which multiple teams can gather to discuss the API and make changes.

Step 3: Generate Server Stub (Spring Boot Scaffolding)

To further accelerate the API development process, go ahead and use the API Editor to Generate the Server. Alternatively a command line tool can be used.

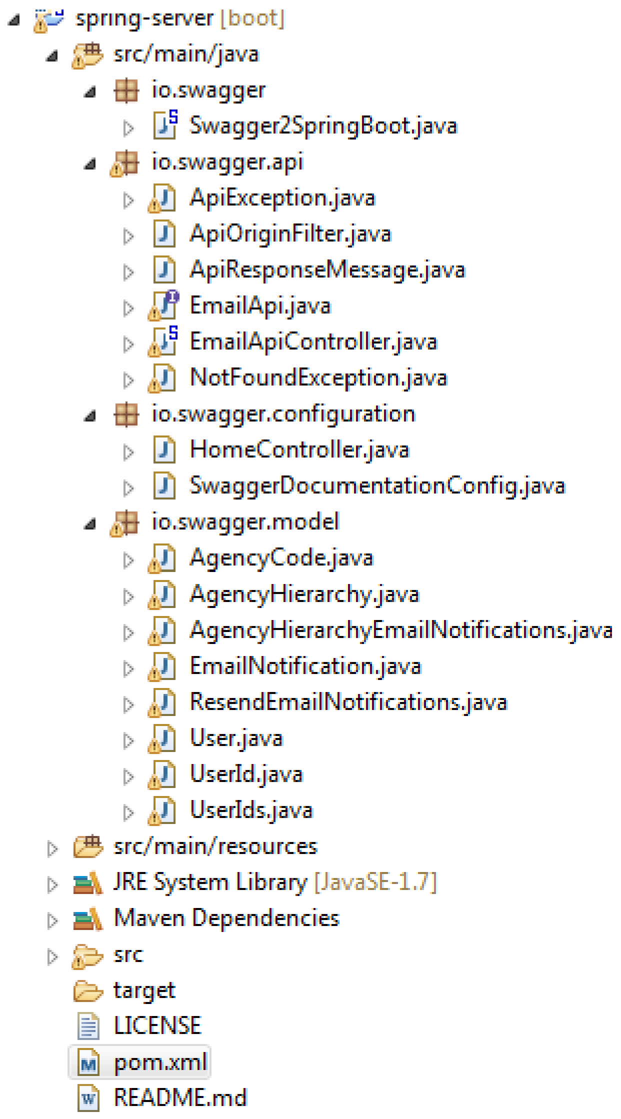
A 'Spring Boot' server gets generated along with a Maven pom file and it's dependencies. A compressed zip file with all the scaffolding code for the API is downloaded.



```
swagger: "2.0"
info:
  description: |
   Liberty Mutual                                          y Delivery E-mai
     Preferences /
  version: "1.0.1'
  title: Liberty I                                         Policy Delivery
    -mail Prefere
  termsOfService:
  contact:
    name: rohit.s
  license:
    name: Liberty
```

```
host: ci-dist-api
basePath: /v1
```

This can be expanded and imported as a Maven project. The parent pom should be updated. Package names should be renamed to match the needs of the service.

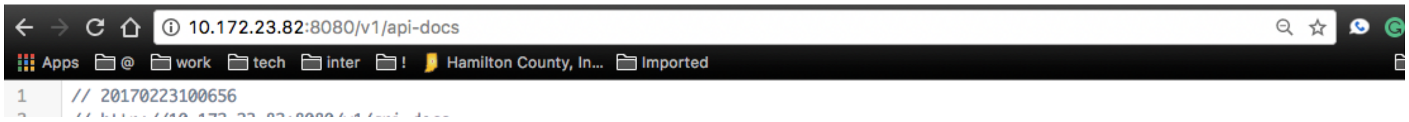


```
spring-server [boot]
  src/main/java
    io.swagger
      Swagger2SpringBoot.java
    io.swagger.api
      ApiException.java
      ApiOriginFilter.java
      ApiResponseMessage.java
      EmailApi.java
      EmailApiController.java
      NotFoundException.java
    io.swagger.configuration
      HomeController.java
      SwaggerDocumentationConfig.java
    io.swagger.model
      AgencyCode.java
      AgencyHierarchy.java
      AgencyHierarchyEmailNotifications.java
      EmailNotification.java
      ResendEmailNotifications.java
      User.java
      UserId.java
      UserIds.java
  src/main/resources
  JRE System Library [JavaSE-1.7]
  Maven Dependencies
  src
  target
  LICENSE
  pom.xml
  README.md
```

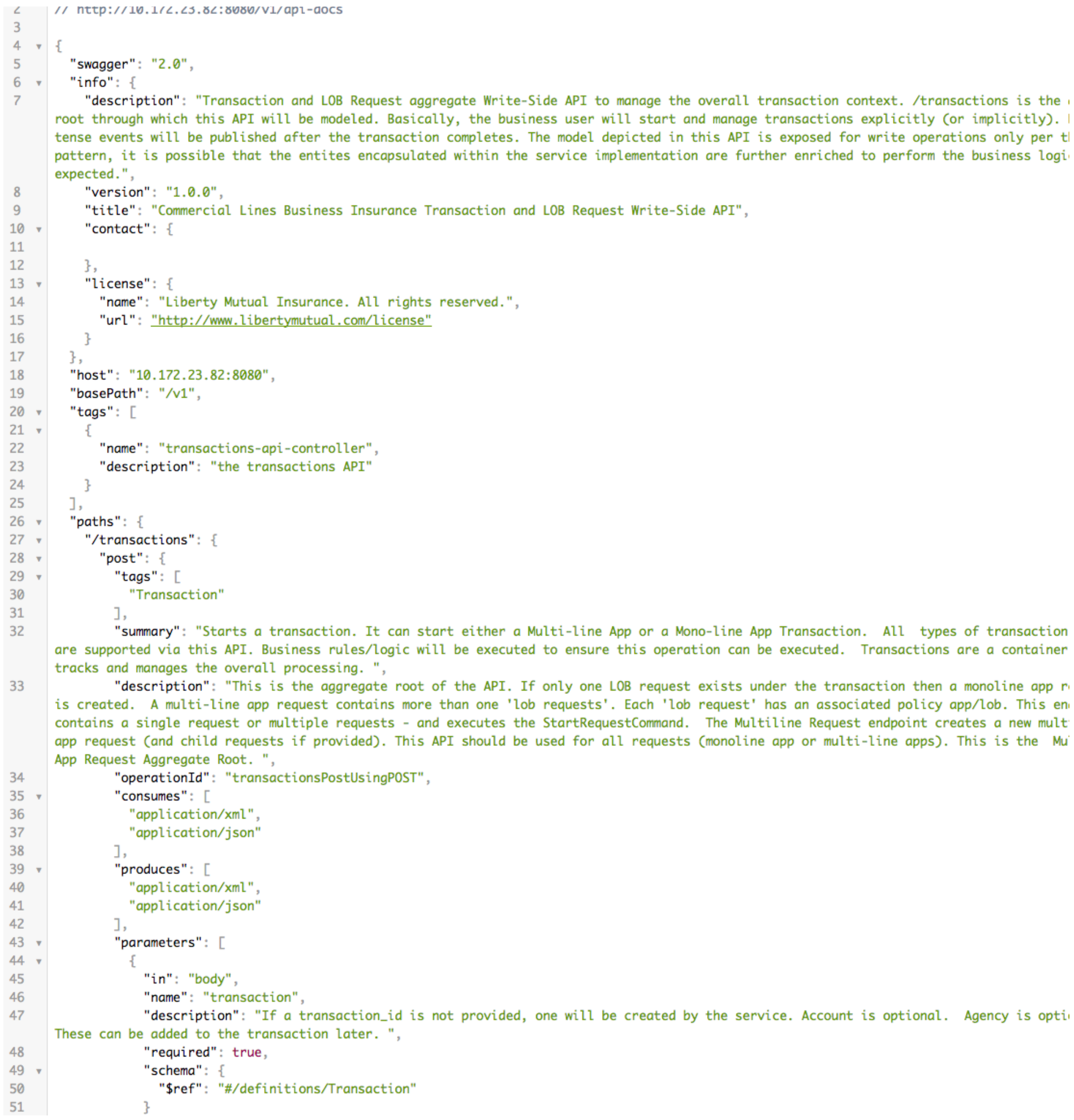
The project already contains a Maven pom file and a Spring Boot project with dependencies. it can be run as-is and deployed - it just works. Open a command terminal `mvn spring-boot:run` .

Step 4: Implement and Deploy

When you run the Spring Boot API - you will see the Annotations have been added to the code - and they show up via the api-docs and also the swagger ui.

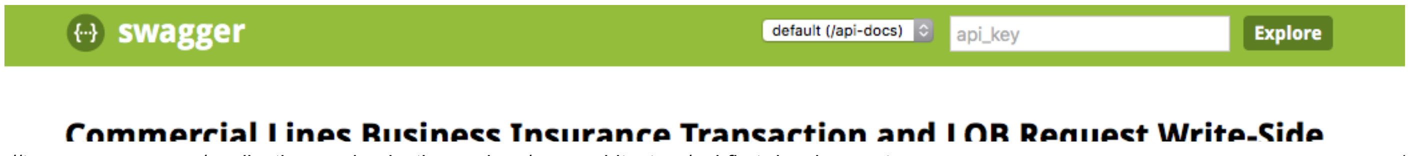


```
// 20170223100656
```

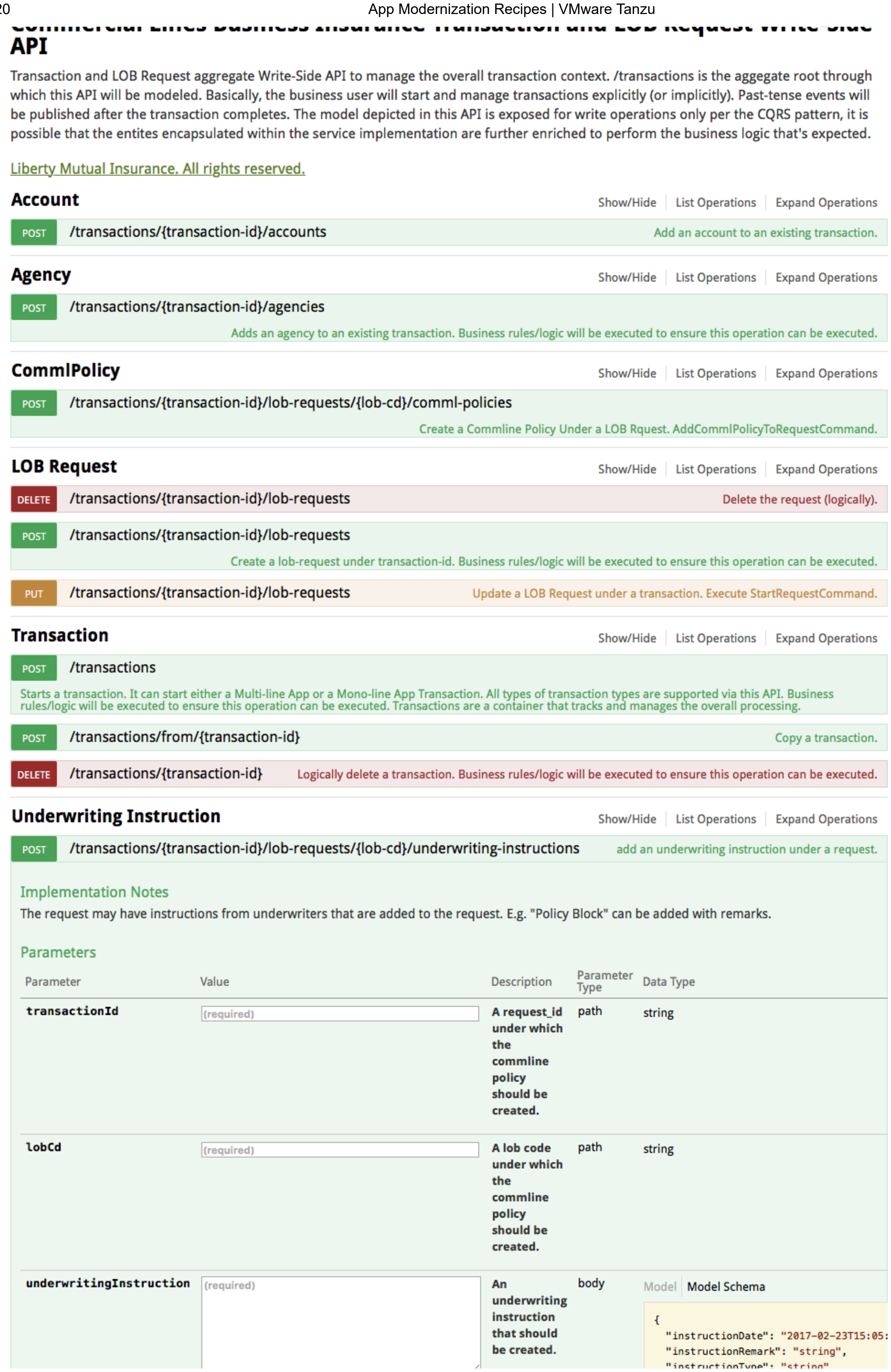


```
// http://10.172.23.82:8080/v1/api-docs

{
  "swagger": "2.0",
  "info": {
    "description": "Transaction and LOB Request aggregate Write-Side API to manage the overall transaction context. /transactions is the
root through which this API will be modeled. Basically, the business user will start and manage transactions explicitly (or implicitly).
tense events will be published after the transaction completes. The model depicted in this API is exposed for write operations only per t
pattern, it is possible that the entites encapsulated within the service implementation are further enriched to perform the business logi
expected.",
    "version": "1.0.0",
    "title": "Commercial Lines Business Insurance Transaction and LOB Request Write-Side API",
    "contact": {

    },
    "license": {
      "name": "Liberty Mutual Insurance. All rights reserved.",
      "url": "http://www.libertymutual.com/license"
    }
  },
  "host": "10.172.23.82:8080",
  "basePath": "/v1",
  "tags": [
    {
      "name": "transactions-api-controller",
      "description": "the transactions API"
    }
  ],
  "paths": {
    "/transactions": {
      "post": {
        "tags": [
          "Transaction"
        ],
        "summary": "Starts a transaction. It can start either a Multi-line App or a Mono-line App Transaction.  All  types of transaction
are supported via this API. Business rules/logic will be executed to ensure this operation can be executed.  Transactions are a container
tracks and manages the overall processing. ",
        "description": "This is the aggregate root of the API. If only one LOB request exists under the transaction then a monoline app r
is created.  A multi-line app request contains more than one 'lob requests'. Each 'lob request' has an associated policy app/lob. This en
contains a single request or multiple requests - and executes the StartRequestCommand.  The Multiline Request endpoint creates a new mult
app request (and child requests if provided). This API should be used for all requests (monoline app or multi-line apps). This is the  Mu
App Request Aggregate Root. ",
        "operationId": "transactionsPostUsingPOST",
        "consumes": [
          "application/xml",
          "application/json"
        ],
        "produces": [
          "application/xml",
          "application/json"
        ],
        "parameters": [
          {
            "in": "body",
            "name": "transaction",
            "description": "If a transaction_id is not provided, one will be created by the service. Account is optional.  Agency is opti
These can be added to the transaction later. ",
            "required": true,
            "schema": {
              "$ref": "#/definitions/Transaction"
            }
```

This api is a JSON format that represents the exact YAML file that was written. Going forward as the code is changed to evolve the shape of the API and it's implementation, SpringFox annotations in the code will always keep the API in sync. If new API end-points are added to this or new verbs - the developer is expected to use the annotations. This should be simple because the API interface class already has annotations and following the same conventions over time will be much easier.



Commercial Lines Business Insurance Transaction and LOB Request Write-Side



API

Transaction and LOB Request aggregate Write-Side API to manage the overall transaction context. /transactions is the aggegate root through which this API will be modeled. Basically, the business user will start and manage transactions explicitly (or implicitly). Past-tense events will be published after the transaction completes. The model depicted in this API is exposed for write operations only per the CQRS pattern, it is possible that the entites encapsulated within the service implementation are further enriched to perform the business logic that's expected.

Liberty Mutual Insurance. All rights reserved.

Account Show/Hide | List Operations | Expand Operations

POST /transactions/{transaction-id}/accounts — Add an account to an existing transaction.

Agency Show/Hide | List Operations | Expand Operations

POST /transactions/{transaction-id}/agencies — Adds an agency to an existing transaction. Business rules/logic will be executed to ensure this operation can be executed.

CommlPolicy Show/Hide | List Operations | Expand Operations

POST /transactions/{transaction-id}/lob-requests/{lob-cd}/comml-policies — Create a Commline Policy Under a LOB Rquest. AddCommlPolicyToRequestCommand.

LOB Request Show/Hide | List Operations | Expand Operations

DELETE /transactions/{transaction-id}/lob-requests — Delete the request (logically).

POST /transactions/{transaction-id}/lob-requests — Create a lob-request under transaction-id. Business rules/logic will be executed to ensure this operation can be executed.

PUT /transactions/{transaction-id}/lob-requests — Update a LOB Request under a transaction. Execute StartRequestCommand.

Transaction Show/Hide | List Operations | Expand Operations

POST /transactions — Starts a transaction. It can start either a Multi-line App or a Mono-line App Transaction. All types of transaction types are supported via this API. Business rules/logic will be executed to ensure this operation can be executed. Transactions are a container that tracks and manages the overall processing.

POST /transactions/from/{transaction-id} — Copy a transaction.

DELETE /transactions/{transaction-id} — Logically delete a transaction. Business rules/logic will be executed to ensure this operation can be executed.

Underwriting Instruction Show/Hide | List Operations | Expand Operations

POST /transactions/{transaction-id}/lob-requests/{lob-cd}/underwriting-instructions — add an underwriting instruction under a request.

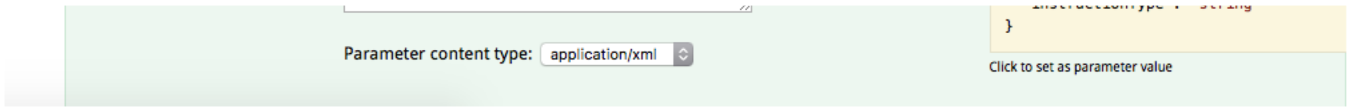
Implementation Notes

The request may have instructions from underwriters that are added to the request. E.g. "Policy Block" can be added with remarks.

Parameters

Parameter	Value	Description	Parameter Type	Data Type
`transactionId`	(required)	**A request_id under which the commline policy should be created.**	path	string
`lobCd`	(required)	**A lob code under which the commline policy should be created.**	path	string
`underwritingInstruction`	(required)	**An underwriting instruction that should be created.**	body	Model | Model Schema

```
{
  "instructionDate": "2017-02-23T15:05:
  "instructionRemark": "string",
  "instructionType": "string"
```


Parameter content type: application/xml
Click to set as parameter value

A user-friendly and interactive Web-Page is generated automatically (Swagger UI). This allows for a quick execution of the API - the web-form allows for data to be entered directly and this allows the API to be tested quickly by any user without any special tools or technology skillsets.

Conclusion

There are many permutations of tools, technologies, languages and methods that try to approach API design and development. Weeks of microservices coding can save you hours of API designing. The goal is to provide a clear path to achieving API-First development and being effective at it. API-First Development practice as outlined in this document is effective and economical.

References

Here's an (opinionated) list of things you need to know (kind of):

1. **Open API 2.0 Specification**
2. **JSON Spec**
3. **YAML Spec**
4. **CI API Editor**
5. **The full API first specifications**