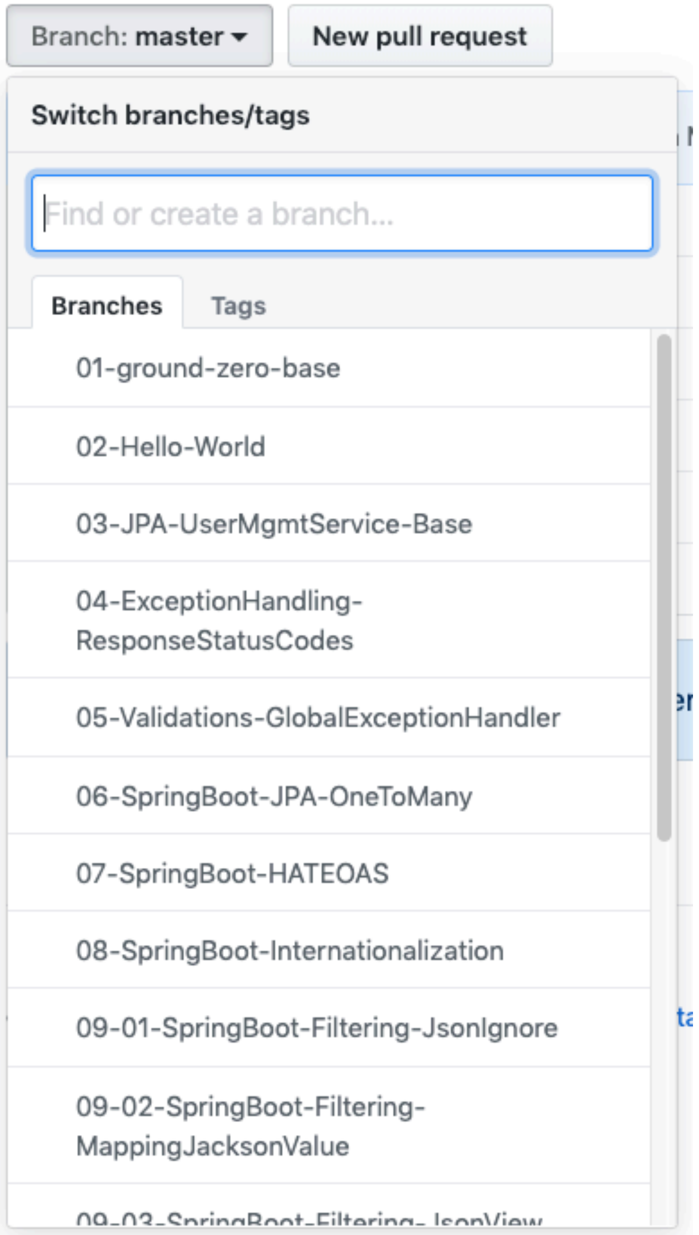
Master RESTful API using Spring Boot 2

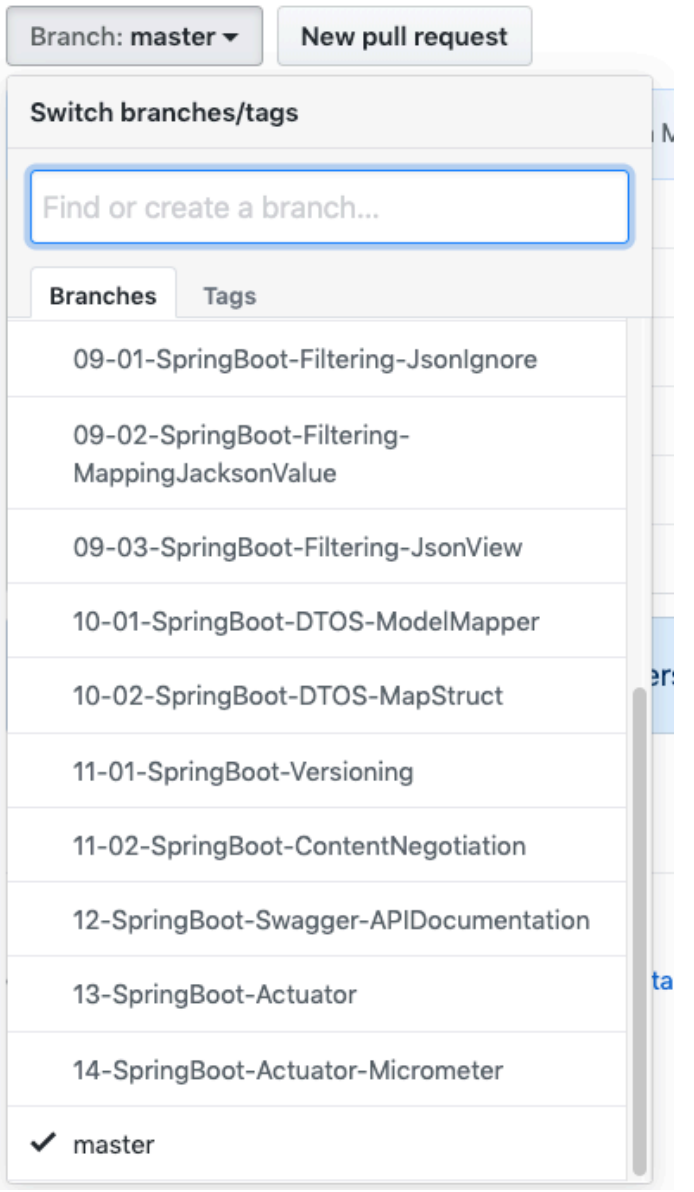
Kalyan Reddy Daida

Course Objectives

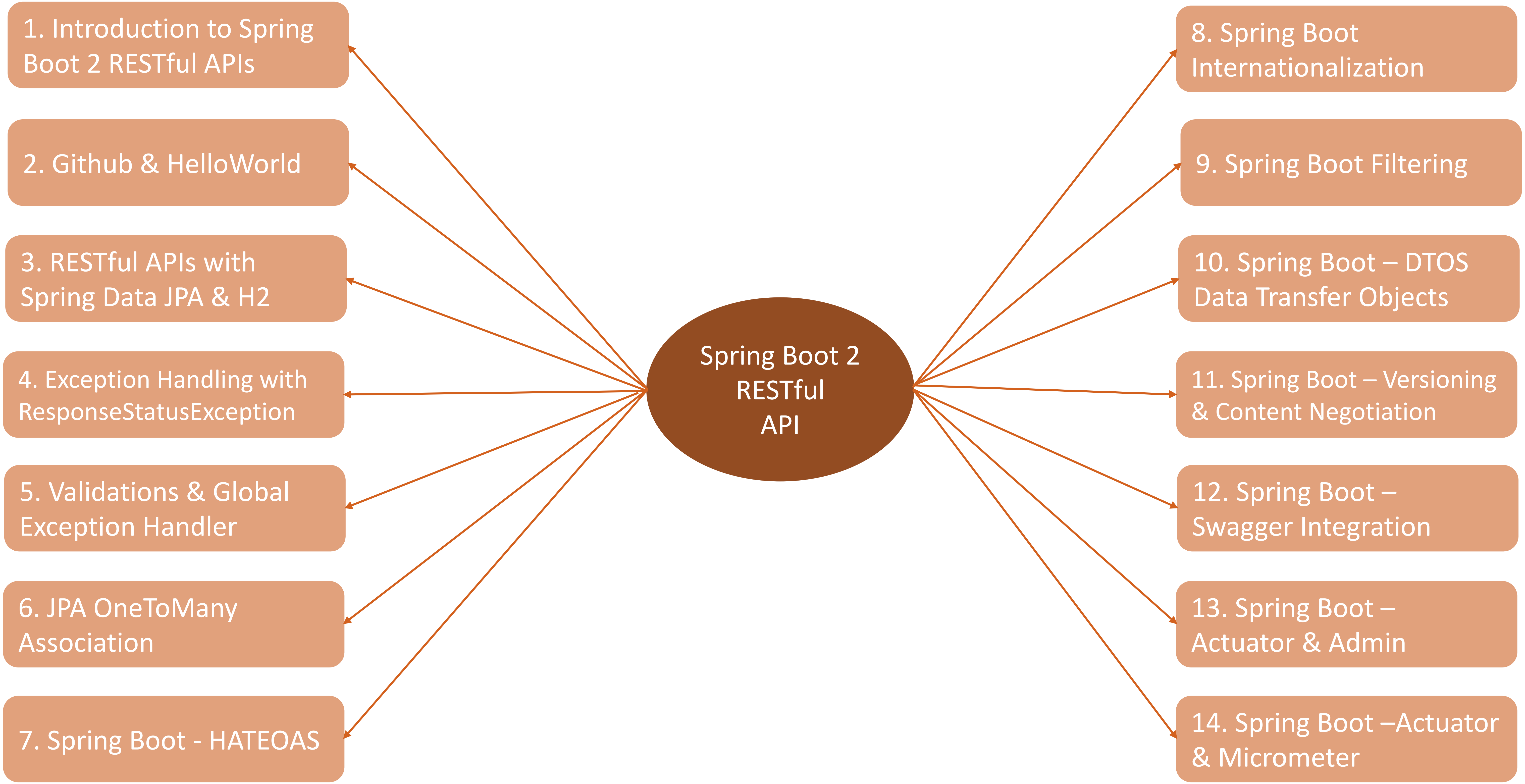
- Goal: We are going to use a single project and incrementally build it by adding each feature in a orderly manner by creating separate git branches for each feature.
- After the completion of course github is going to look this way.
- Project Link: https://github.com/stacksimplify/springboot-buildingblocks



01-ground-zero-base
02-Hello-World
03-JPA-UserMgmtService-Base
04-ExceptionHandling-ResponseStatusCodes
05-Validations-GlobalExceptionHandler
06-SpringBoot-JPA-OneToMany
07-SpringBoot-HATEOAS
08-SpringBoot-Internationalization
09-01-SpringBoot-Filtering-JsonIgnore
09-02-SpringBoot-Filtering-MappingJacksonValue
09-03-SpringBoot-Filtering-JsonView



09-01-SpringBoot-Filtering-JsonIgnore
09-02-SpringBoot-Filtering-MappingJacksonValue
09-03-SpringBoot-Filtering-JsonView
10-01-SpringBoot-DTOS-ModelMapper
10-02-SpringBoot-DTOS-MapStruct
11-01-SpringBoot-Versioning
11-02-SpringBoot-ContentNegotiation
12-SpringBoot-Swagger-APIDocumentation
13-SpringBoot-Actuator
14-SpringBoot-Actuator-Micrometer
✓ master



Spring Boot 2 RESTful API

1. Introduction to Spring Boot 2 RESTful APIs

2. Github & HelloWorld

3. RESTful APIs with Spring Data JPA & H2

4. Exception Handling with ResponseStatusException

5. Validations & Global Exception Handler

6. JPA OneToMany Association

7. Spring Boot - HATEOAS

8. Spring Boot Internationalization

9. Spring Boot Filtering

10. Spring Boot – DTOS Data Transfer Objects

11. Spring Boot – Versioning & Content Negotiation

12. Spring Boot – Swagger Integration

13. Spring Boot – Actuator & Admin

14. Spring Boot –Actuator & Micrometer

Spring Boot

Development Environment

GitHub: https://github.com/stacksimplify/springboot-buildingblocks

Spring Boot

RESTful API Introduction

RESTful Webservices

- REST stands for Representational State Transfer
- Restful Web Services is a **stateless client-server** architecture where web services are resources and can be identified by their URIs.
- REST Client applications can use HTTP GET/POST/PUT/DELETE .. methods to invoke Restful web services.
- Lightweight and doesn’t follow any standards unlike SOAP Webservices.

SOAP vs REST

- SOAP is a protocol
- SOAP server and client applications are tightly coupled and bind with the WSDL contract.
- Learning curve is little complex for SOAP web services.
- Rigid type checking, binds to a contract.
- SOAP works with XML only.

- REST is an architectural style.
- There is no contract in REST web services and client application consuming REST API.
- Learning curve is easy for REST when compared to SOAP.
- Human readable results.
- REST web services request and response types can be XML, JSON, text etc..

Spring Boot

Manage via Github
&
HelloWorld

Implementation Steps

- Step-01: Create Spring boot base project from start.spring.io
- Step-02: Introduction for managing spring boot projects via github
- Step-03: Github Base Setup
- Step-04: Add GIT Repository to Spring Tool Suite IDE - GIT Perspective
- Step-05: Create a Simple HelloWorld RESTful API which returns a String
- Step-06: Create a Simple Hello World REST Service which returns a Bean (JSON)
- Step-07: GIT Commit & Push Hello World RESTful service changes to Github

Spring Boot

RESTful APIs using Spring Data JPA & H2 Database

Implementation Steps

- Step-01: Usecase Introduction
- Step-02: Verify pom.xml for all Dependencies
- Step-03: Update application.properties required for JPA based RESTful Services
- Step-04-01: Create User Entity - Understand @Entity Annotation
- Step-04-02: Create User Entity - Understand @Table Annotation
- Step-04-03: Create User Entity - Define Variables, Getters & Setters
- Step-05: Understand and Implement changes related to H2 Database
- Step-06: Create User Repository - @Repository

Implementation Steps

- Step-07: Implement getAllUsers RESTful Service - @Service, @RestController
- Step-08: Test getAllUsers RESTful Service - Using REST Client POSTMAN
- Step-09: Implement createUser RESTful Service - @PostMapping
- Step-10: Implement getUserById RESTful Service - @GetMapping
- Step-11: Implement updateUserById RESTful service - @PutMapping
- Step-12: Implement deleteUserById RESTful Service - @DeleteMapping
- Step-13: Implement getUserByUsername RESTful Service - @GetMapping
- Step-14: GIT Commit, Push, Merge to Master and Push

Exception Handling
&
Response Status Codes

ResponseStatusException Class

- Spring5 introduces the ResponseStatusException class which is a fast way for basic error handling in our RESTful API's.
- It is an alternative to @ResponseStatus and is the base class for exceptions used for applying status code to an HTTP Response.
- We can create an instance of it providing an HttpStatus and optionally a reason and a cause.
- It's a RuntimeException.
- ResponseStatusException constructor arguments
  - status - an HTTP Status set to HTTP response
  - reason – a message explaining the exception set to that particular HTTP response
  - cause – a Throwable cause of the ResponseStatusException

ResponseStatusException Class

- Benefits
  - We can implement it quite fast.
  - There is no specific need for creating custom exception classes, unless we have a need because we can define HTTP Response Status code and Error message at a time.
  - As we are creating exceptions programmatically, we will have more control over exception handling.
- Downside
  - Code Duplication: As we are defining them programmatically, we find ourselves replicating code in multiple controllers.
  - Global Exception Handling: This approach will not look like a global approach like @ControllerAdvice. Its difficult to enforce application-wide conventions.
- Combine Approaches
  - We can implement @ControllerAdvice globally and ResponseStatusExceptions locally as and when required.

Implementation Steps

- Step-00: Create a git branch for Exception Handling.
- Step-01: Implement “ResponseStatusException” for getUserById service.
- Step-02: Implement “ResponseStatusException” for updateUserById service.
- Step-03: Implement “ResponseStatusException” for deleteUserById service directly at Service Layer.
- Step-04: Implement “ResponseStatusException” for createUser service.
- Step-05: Implement HTTP Status code – 201 created and Location header with user path for createUser Service.

Spring Boot

Validations
&
Global Exception Handling

Validations

- Validating user input is a very common & key requirement in today's world, Spring Boot provides strong support out of the box.
- Spring Boot supports seamless integration with custom validators but the de-facto for performing validation is Hibernate Validator (http://hibernate.org/validator/).
- JSR 380: JSR 380 is a specification of the Java API for bean validation, which ensures that properties of a bean meet specific criteria, using annotations such as *@NotNull, @Min,* and *@Max*.
- @Valid Annotation: When Spring Boot finds an argument annotated with @Valid, it automatically bootstraps the default JSR 380 implementation (Hibernate Validator) and validates the argument. When the target argument fails to pass the validation, Spring Boot throws a MethodArgumentNotValidException exception.
- Bean Validation 2.0 Specification - https://beanvalidation.org/2.0/

Commonly used - Validation Annotations

- @NotNull: Validates that the annotated property value is not null
- @Size: Validates that the annotated property value has a size between the attributes min and max
- @Min: Validates that the annotated property has a value no smaller than the value attribute
- @Max: Validates that the annotated property has a value no larger than the value attribute
- @Email: Validates that the annotated property is a valid email address
- @NotBlank: Validate that the property is not null or whitespace
- @NotEmpty: Validates that the property is not null or empty
- @AssertTrue: Validates that the annotated property value is true.

Global Exception Handling

- @ControllerAdvice
  - Allows us to write global code that can be applied to a wide range of controllers.
  - By default @ControllerAdvice annotation will be applicable to all classes that use @Controller which also applies for @RestController.
- @ExceptionHandler
  - Annotation for handling exceptions in specific handler classes and/or handler methods.
  - If used with controllers directly, we have the need to define it per controller but when used in combination with @ControllerAdvice it will be only used in Global Exception Handler class but applicable to all controllers due to @ControllerAdvice.
- @RestControllerAdvice
  - @RestControllerAdvice is the combination of both @ControllerAdvice and @ResponseBody.
  - We can use the @ControllerAdvice annotation for handling exceptions in the RESTful Services but we need to add @ResponseBody separately.

Usecase Combination

- @ControllerAdvice & ResponseEntityExceptionHandler class
  - MethodArgumentNotValidException
  - HttpRequestMethodNotSupportedException
- @ControllerAdvice & @ExceptionHandler
  - For pre-defined exceptions like ConstraintViolationException
  - For custom exceptions like UserNameNotFoundException
- @RestControllerAdvice & @ExceptionHandler
  - For custom exceptions like UserNameNotFoundException
  - For pre-defined exceptions like “Exception.class” (Applicable to all exceptions)

Implementation Steps

- Step-00: Create git branch for Validations & Global Exception Handler.
- Step-01: Implement Bean Validation
- Step-02: Implement Custom Global Exception Handler using @ControllerAdvice & ResponseEntityExceptionHandler
  - implement exception handler for MethodArgumentNotValidException.
- Step-03: Implement exception handler for HttpRequestMethodNotSupportedException.
- Step-04: Implement exception handler for custom exception like UserNameNotFoundException.
- Step-05: Path Variables Validation & implement exception handler for ConstraintViolationException.
- Step-06: Implement Global Exception Handling using @RestControllerAdvice

Spring Boot

JPA
@OneToMany
@ManyToOne

JPA - @OneToMany & @ManyToOne

- In JPA, one-to-many database association can be represented either through a @ManyToOne or @OneToMany association or both. All depends on our requirement and need.
- @ManyToOne annotation allows us to map the Foreign Key column in the child entity mapping so that child has an entity object reference to its parent entity. This is the most efficient way.
- We can perform the associations in below listed 3 ways.
  - Unidirectional @OneToMany association
  - Bidirectional @OneToMany association
  - Unidirectional @ManyToOne with JPQL query
- We are going to use Bidirectional association in our implementation.

Usecase Introduction

- Get All orders of a User
  - Method Name: getAllOrders
  - GET /users/{userid}/orders
- Create an order for a user
  - Method Name: createOrder
  - POST /users/{userid}/orders
- Get order details using orderid and userid
  - Method Name: getOrderByOrderId
  - GET /users/{userid}/orders/{orderid}

JPA - @OneToMany - Implementation Steps

- Step-01: Create GIT branch for JPA one-to-many association
- Step-02: Create Order entity and @ManyToOne association
- Step-03: Update User entity with @OneToMany association
- Step-04: Implement getAllOrders method
- Step-05: Implement createOrder method
- Step-06: Implement getOrderByOrderId method (Assignment)
- Step-07: GIT commit code, push to remote, merge to master and push to remote

Spring Boot

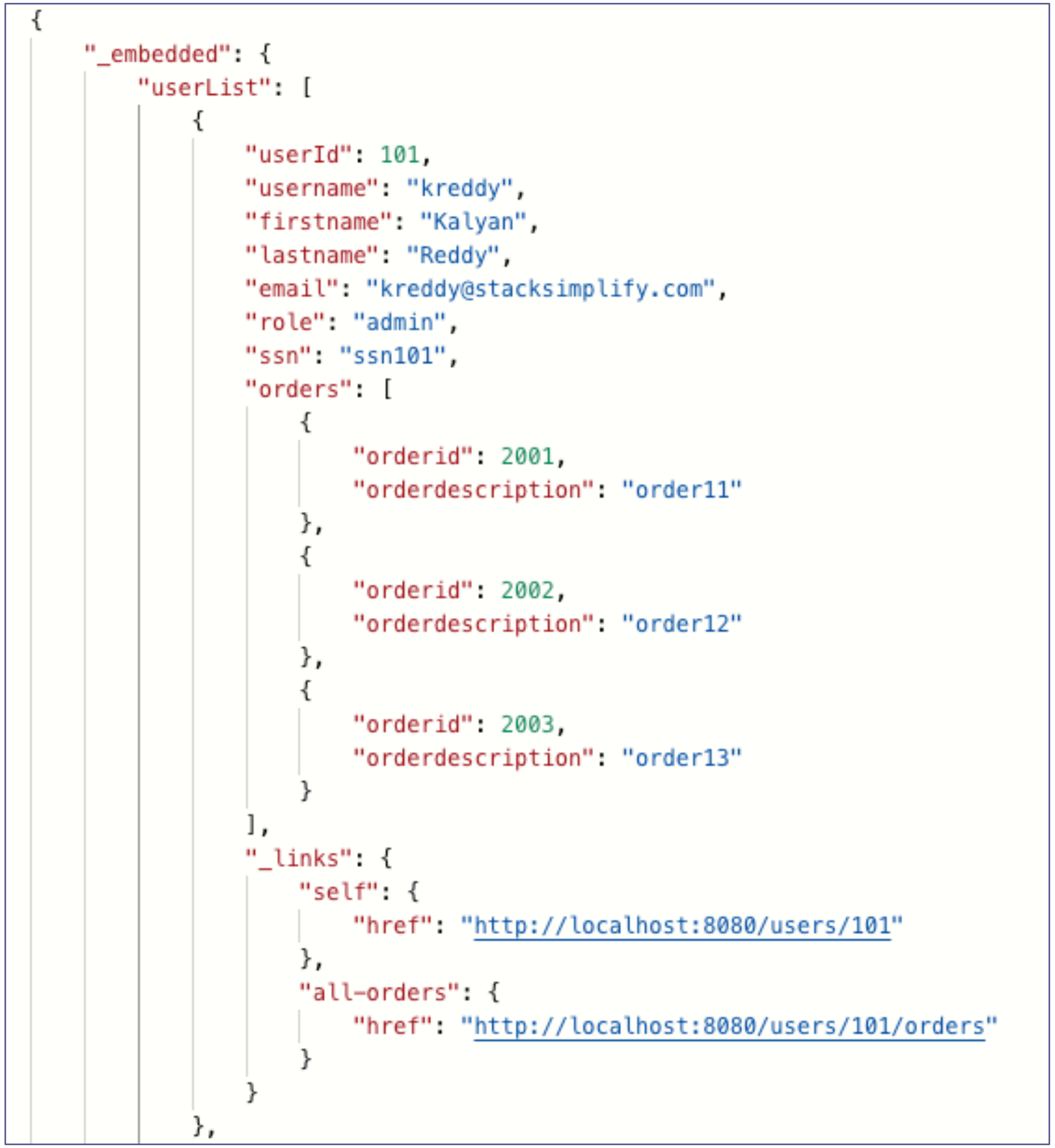
HATEOAS

Hypermedia As The Engine Of Application State

Spring Boot HATEOAS

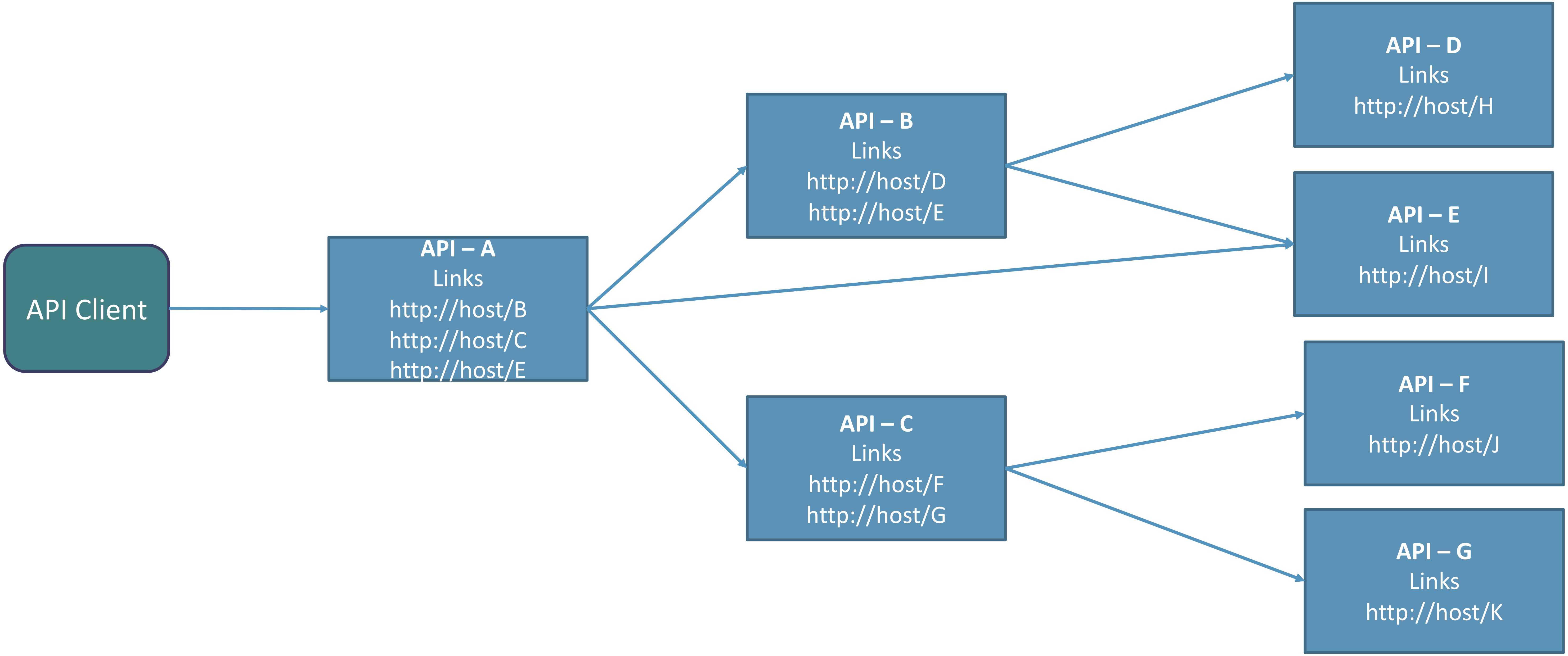
- HATEOAS is an extra level upon REST.
- It is used to present information about a REST API to a client without the need to bring up the API documentation.
- It includes links in a returned response and client can use those API links to further communicate with the server.
- Simplify the client by making the API discoverable.
- This reduces the likelihood of client breaking due to changes to the service. How?

Service: getAllUsersHateoas



```
{
    "_embedded": {
        "userList": [
            {
                "userId": 101,
                "username": "kreddy",
                "firstname": "Kalyan",
                "lastname": "Reddy",
                "email": "kreddy@stacksimplify.com",
                "role": "admin",
                "ssn": "ssn101",
                "orders": [
                    {
                        "orderid": 2001,
                        "orderdescription": "order11"
                    },
                    {
                        "orderid": 2002,
                        "orderdescription": "order12"
                    },
                    {
                        "orderid": 2003,
                        "orderdescription": "order13"
                    }
                ],
                "_links": {
                    "self": {
                        "href": "http://localhost:8080/users/101"
                    },
                    "all-orders": {
                        "href": "http://localhost:8080/users/101/orders"
                    }
                }
            },
```

Spring Boot HATEOAS – API Discovery



API Client

API – A
Links
http://host/B
http://host/C
http://host/E

API – B
Links
http://host/D
http://host/E

API – C
Links
http://host/F
http://host/G

API – D
Links
http://host/H

API – E
Links
http://host/I

API – F
Links
http://host/J

API – G
Links
http://host/K

Spring Boot HATEOAS

- Spring HATEOAS provides three abstractions for creating the URI
  - Resource Support
  - Link
  - ControllerLinkBuilder
- We can use these to build the API URL's and associate it to the resource.
- We extend entities (User, Order) from the Resource Support class to inherit the add() method.
- Once we create a link, we can easily associate that link to a resource representation without adding any new fields to the resource or without writing huge amount of manual boilerplate code.

Spring Boot – HATEOAS - Implementation Steps

- Step-00: Create git branch for Spring Boot HATEOAS
- Step-01: Add HATEOAS dependency in pom.xml
- Step-02: Extend both Entities to ResourceSupport
- Step-03: Create new User and Order Controllers for HATEOAS Implementation
  - UserHateoasController
  - OrderHateoasController
- Step-04: Implement self link in getUserById method
- Step-05: Implement self and relationship links in getAllUsers Method. Relationship link will be with getAllOrders method.
  - 5(A) - Self Link for each user
  - 5(B) - Relation Ship Link with getAllOrders
  - 5(C) - Self Link for getAllUsers
- Step-06: GIT commit code

Spring Boot

Internationalization

Spring Boot Internationalization

- Internationalization – i18n
- Internationalization is a process that makes our application adaptable to different languages without making any changes to our source code.
- In other words, Internationalization is a readiness of Localization.
- Spring Boot provides LocaleResolver & ResourceBundleMessageSource which is a foundation to the internationalization.

Internationalization - Implementation Steps

- Step-00: Create git branch for Spring Boot Internationalization
- Step-01: Create required beans and message property files per language
  - LocaleResolver
  - ResourceBundleMessageSource
  - messages.properties
  - messages_fr.properties
- Step-02: Create a simple rest service and convert it to support internationalization.
- Step-03: GIT commit code

Spring Boot

Static Filtering using @JsonIgnore & @JsonIgnoreProperties

Static Filtering

- Static Filtering
  - @JsonIgnore will be applied at field level in a model class (Entity).
  - @JsonIgnoreProperties will be applied class level in a model class and we can define list of fields that can be ignored.
  - Simply hides the field from the Jackson parser.
  - Cons
  - Create or Update requests will fail after applying these annotations (POST, PUT).

Spring Boot

Dynamic Filtering using MappingJacksonValue & @JsonFilter

Dynamic Filtering

- We are going to use MappingJacksonValue to implement dynamic filtering.
- @JsonFilter applied at Model class with filtername.
- Rest all logic related to filtering will be defined in service or controller layer.
- Usecase-1: We will first implement it with a basic hash set.
- Usecase-2: We will send fields using REST service query parameters to retrieve the data for those respective fields.

Spring Boot

Filtering / Creating Views using @JsonView

@JsonView

- @JsonView is used to customize views.
- Applied at field level in a model class to categorize which field belongs to which view.
- Applied at service level in a controller, so that for that respective REST service, view defined in @JsonView will be applicable.
- Will be very useful if we have a single entity or model which need to be provided with different views to different category of clients.

@JsonView

- Course Example: User & Order Management
  - We have a user entity defined with fields (userid, username, firstname, lastname, email, role, ssn, orders)
  - Consider we need to present data in 2 view patterns
    - External View: userid, username, firstname, lastname, email
    - Internal View: userid, username, firstname, lastname, email, role, ssn, orders
- Classic Example: Employee Management. (Assignment)
  - We have employee data (empid, name, department, loginTime, logoutTime, salary, lastPromotionDate)
  - Consider we need to present employee data in 3 views.
    - Normal View: empid, name, department
    - Manager View: empid, name, department, loginTime, logoutTime
    - HR View: empid, name, department, salary, lastPromotionDate

Spring Boot

Implement DTOs Using Model Mapper

ModelMapper

- DTOs stands for Data Transfer Objects
- Exposing entity objects through REST endpoints can mount security issues provided if we don't take enough care about which entity fields should be made available for publicly exposed REST API.
- ModelMapper is a library which supports to convert entity objects to DTOs and DTOs to entity objects.
- Intelligent
  - No manual mapping needed.
  - Automatically projects and flattens complex models.
- Refactoring Safe
  - It provides a simple fluent API for handling special usecases
  - The API is type safe and refactoring safe.

ModelMapper

- Convention Based:
  - ModelMapper provides predefined conventions and if user is in need can create custom conventions.
- Extensible
  - ModelMapper supports integrations with any type of data model. In short ModelMapper does the heavy lifting for us.
- Reference
  - http://modelmapper.org/
  - http://modelmapper.org/getting-started/

Model Mapper Implementation Steps

- Step-01: Create new GIT branch using IDE
- Step-02: Add Model Mapper dependency in pom.xml
- Step-03: Define Model Mapper bean in AppConfig
- Step-04: DTO Layer: Create a DTO with name as UserMmDTO.
- Step-05: Controller Layer: Create getUserDtoById method with Entity to DTO Conversion logic with Model Mapper in a new controller UserModelMapperController.
- Step-06: Commit & Push Code (using IDE)

Spring Boot

Implement DTOS Using MapStruct

MapStruct

- MapStruct is a code generator that simplifies bean mappings.
- Mapping classes are generated during compilation and no runtime processing or reflection is used.
- Mapping classes use simple method invocation, which makes them really easy to debug.
- We generally notice a lot of boilerplate code converting POJOs to other POJOs.
- Very common type of conversion we see regularly is in between persistence-backed entities and DTOs that go out to the client side.
- The problem that MapStruct solves is it can generate bean mapper classes automatically. If we go by implementing them manually, creating bean mappers is time-consuming.
- MapStruct also requires a processor plugin to be added to *pom.xml.* The *mapstruct-processor* is used to generate the mapper implementation during the build phase.

MapStruct Implementation Steps

- Step-01: Create new GIT branch using IDE
- Step-02: Update pom.xml with necessary dependencies for MapStruct
- Step-03: Create UserMsDTO class required for MapStruct Implementation.
- Step-04: Create the MapStruct Mapper Interface
- Step-05: Create the REST services by calling methods defined in MapStruct Mapper.
- Step-06: Commit & Push code via IDE

Spring Boot

API Versioning

API Versioning

- URI Versioning
- Request Parameter Versioning
- Custom Header Versioning
- Media Type or Mime Type or Accept Header Versioning

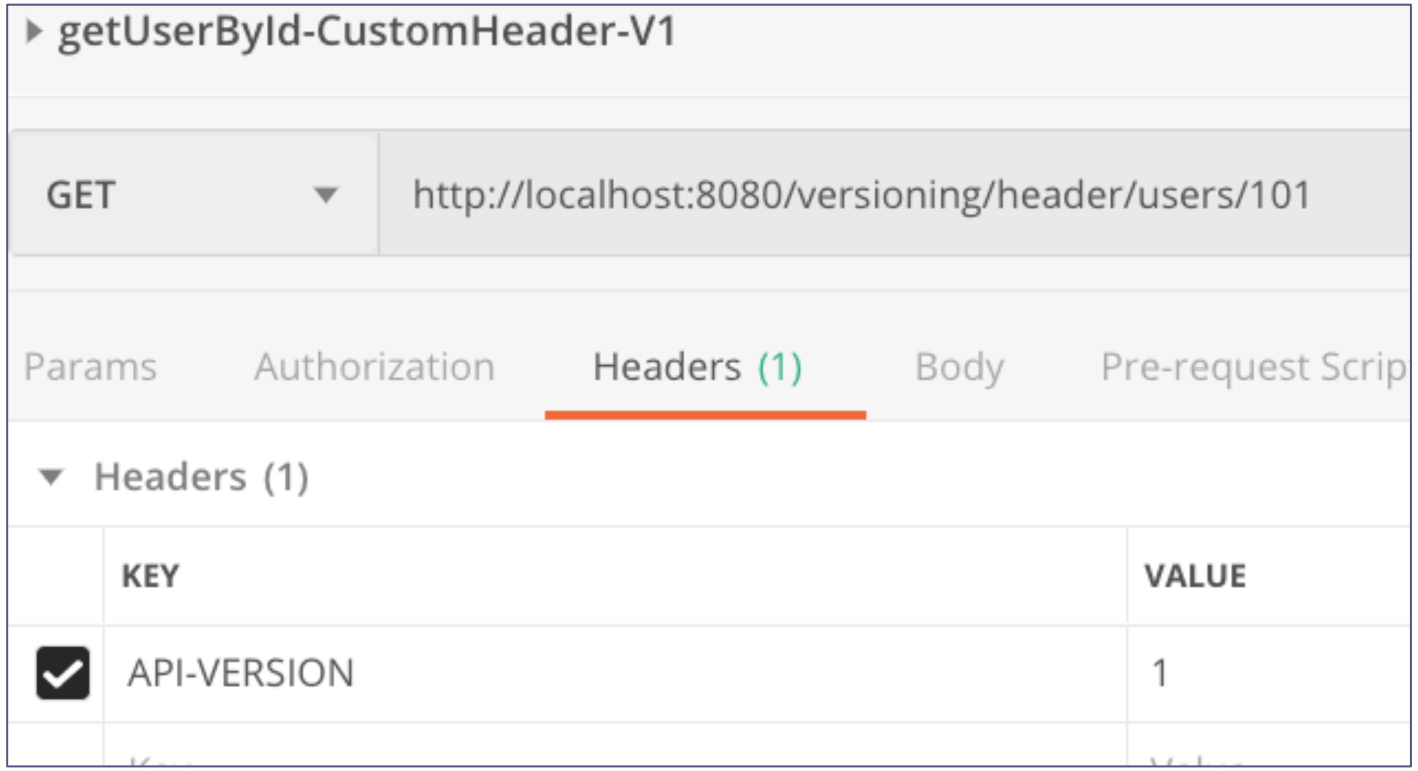
API Versioning

- URI Versioning
  - http://localhost:8080/versioning/uri/users/v1.0/101
  - http://localhost:8080/versioning/uri/users/v2.0/101

- Request Parameter Versioning
  - http://localhost:8080/versioning/params/users/101?version=1
  - http://localhost:8080/versioning/params/users/101?version=2

API Versioning

- Custom Header Versioning

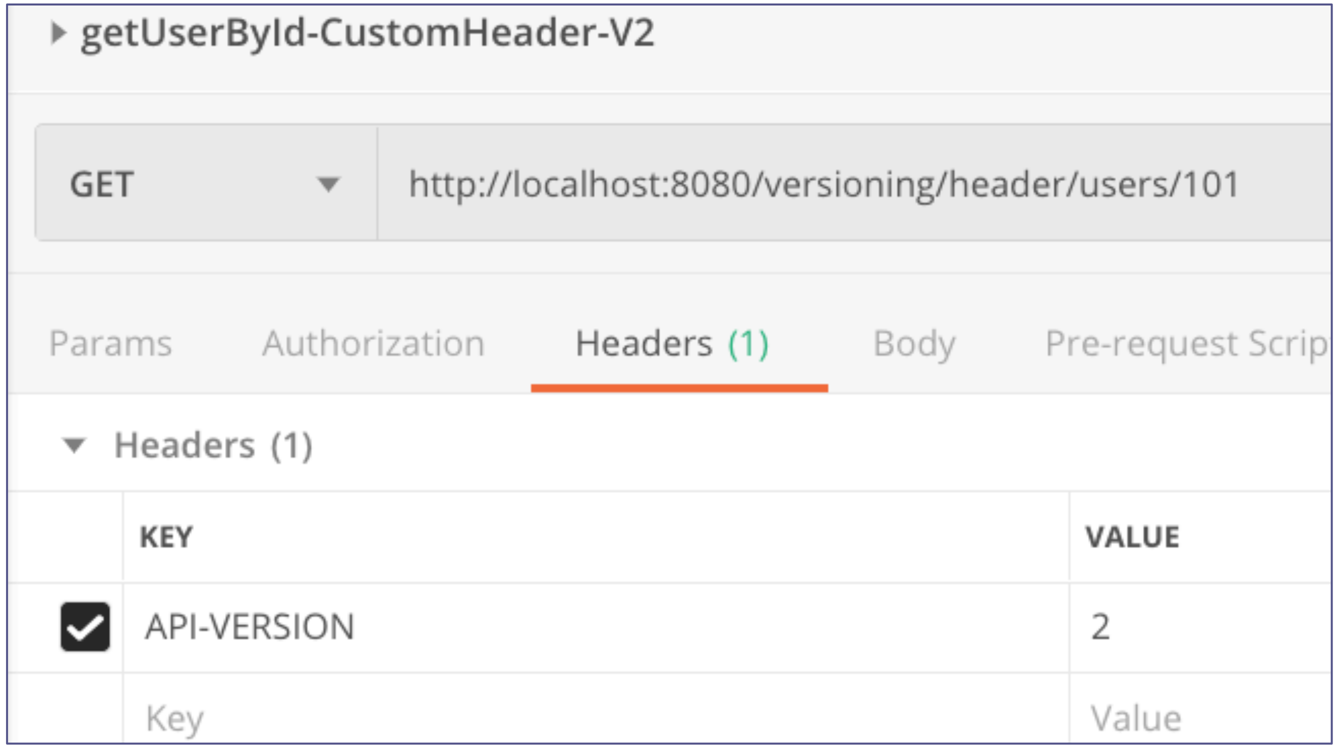


getUserById-CustomHeader-V1

GET http://localhost:8080/versioning/header/users/101

Headers (1)

	KEY	VALUE
☑	API-VERSION	1



getUserById-CustomHeader-V2

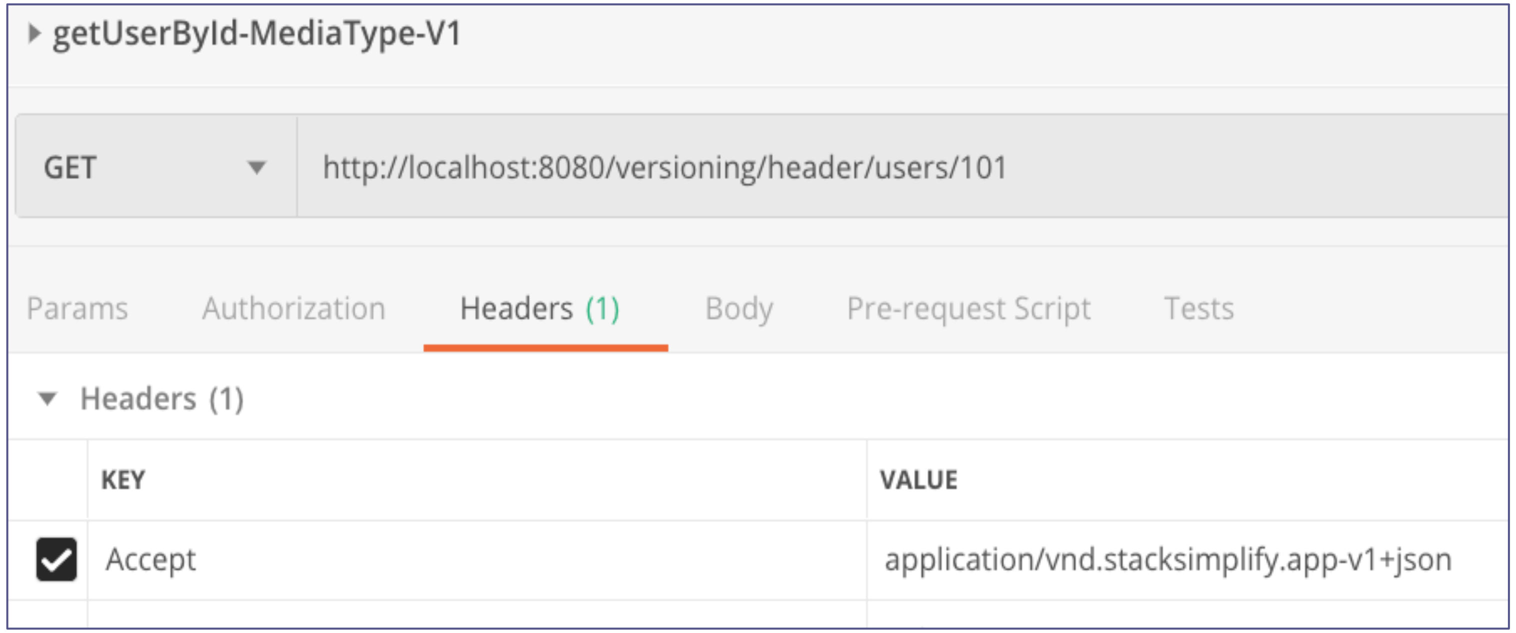
GET http://localhost:8080/versioning/header/users/101

Headers (1)

	KEY	VALUE
☑	API-VERSION	2
	Key	Value

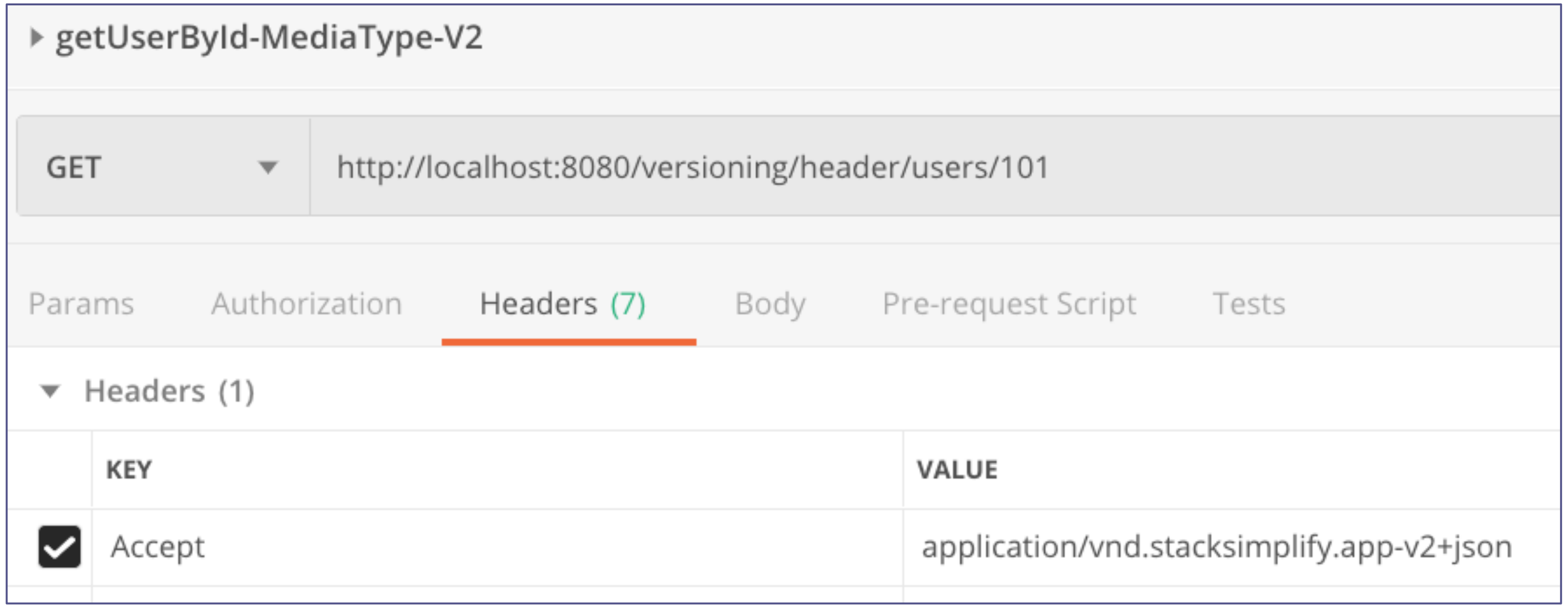
API Versioning

- Media Type or Mime Type or Accept Header Versioning



GET http://localhost:8080/versioning/header/users/101

	KEY	VALUE
☑	Accept	application/vnd.stacksimplify.app-v1+json



GET http://localhost:8080/versioning/header/users/101

	KEY	VALUE
☑	Accept	application/vnd.stacksimplify.app-v2+json

API Versioning - Implementation Steps

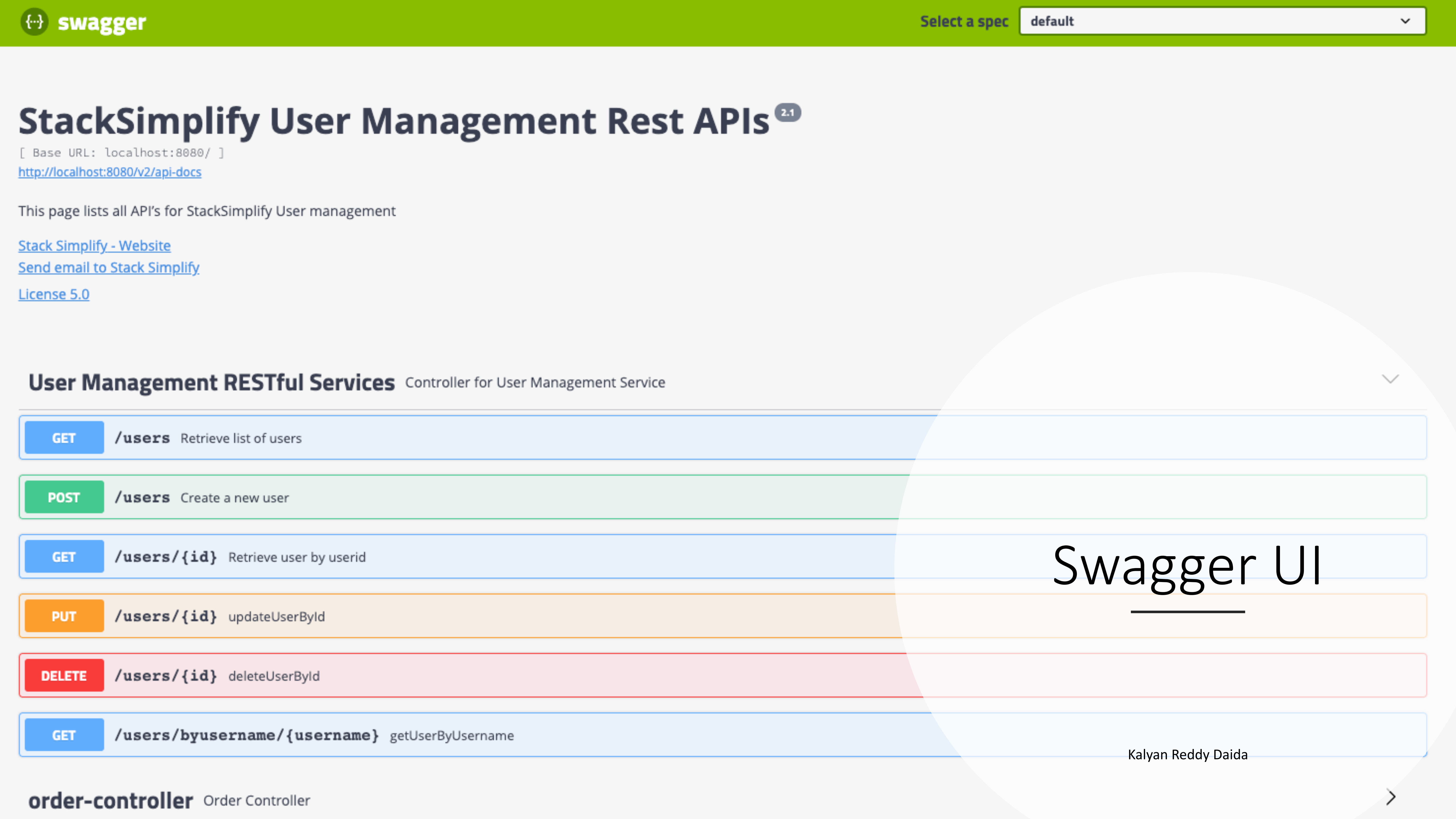
- Step-01: Create new GIT branch using IDE
- Step-02: Create two DTO’s and address field in User Entity.
- Step-03: Implement URI Versioning
- Step-04: Implement Request Parameter Versioning
- Step-05: Implement Custom Header Versioning
- Step-06: Implement Media Type Versioning
- Step-07: Commit & Push code via IDE

Spring Boot

Swagger Integration

Spring Boot - Swagger Integration

- Documenting REST API is very important primarily from API consumers point of view.
- API Documentation helps consumers to understand and implement their client applications without any confusion and also by avoiding costly mistakes.
- One of the most popular API documentation specifications is OpenApi, formerly known as Swagger.
- Swagger allows us to describe API properties either using JSON or YAML metadata.
- Swagger also provides a Web UI which transforms the JSON metadata to a nice HTML documentation.
- Swagger UI can also be used as a REST client.
- Swagger integration with Spring Framework can be implemented using SpringFox dependencies.



StackSimplify User Management Rest APIs 2.1

[Base URL: localhost:8080/]

http://localhost:8080/v2/api-docs

This page lists all API's for StackSimplify User management

Stack Simplify - Website

Send email to Stack Simplify

License 5.0

User Management RESTful Services Controller for User Management Service

GET /users Retrieve list of users

POST /users Create a new user

GET /users/{id} Retrieve user by userid

PUT /users/{id} updateUserById

DELETE /users/{id} deleteUserById

GET /users/byusername/{username} getUserByUsername

order-controller Order Controller

Swagger UI

Kalyan Reddy Daida

Swagger – Implementation Steps

- Step-01: New GIT branch (usign IDE)
- Step-02: Add Springfox Dependencies to pom.xml
- Step-03: Create SwaggerConfig file
- Step-04: Adding API Info to modify header part of our documentation
- Step-05: Restrict scope of swagger document generation using API Basepackages & Paths
- Step-06: Auto populate documentation for JSR-303 Validations
- Step-07: Adding Swagger Core Annotations to Model classes
- Step-08: Adding Swagger Core Annotations to Controller classes
- Step-09: Commit & Push code via IDE

Spring Boot

Actuator

Spring Boot Actuator

- Monitor and Manage Spring Boot Applications using REST/JMX Actuator endpoints.
- The endpoints offer
  - Health Check
  - Metrics Monitoring
  - Access To Logs
  - Thread Dumps
  - Heap Dumps
  - Environmental Info
  - and many more

Spring Boot Actuator Endpoints

auditevents	liquibase
beans	metrics
caches	mappings
conditions	scheduledtasks
configprops	sessions
env	shutdown
flyway	threaddump
health	**Spring MVC, Spring WebFlux, or Jersey**
httptrace	heapdump
info	jolokia
integrationgraph	logfile
loggers	prometheus

Spring Boot Actuator - Implementation Steps

- Step-01: New GIT branch using IDE
- Step-02: Add Spring Boot Actuator Dependency in pom.xml and verify actuator endpoints
- Step-03: Expose all Actuators endpoints and verify Health endpoint and discuss about all other endpoints.
- Step-04: Info Endpoint: Populate build-info on info endpoint.
- Step-05: Metrics Endpoint

Spring Boot

Admin

Spring Boot Admin

- Spring Boot Admin server is a web application used for managing and monitoring spring Boot Applications.
- It is available as a war packages so it can be deployed to any of the JVMs (example: tomcat).
- Each application is considered as a client and registers to Spring Boot Admin server.
- In the background, all the data displayed on Admin Server is using Spring Boot Actuator endpoints enabled on client application.

Spring Boot Admin Server Features

- Features
  - Dashboard with desktop notifications
  - View application health, info and details fetched using actuator endpoints.
  - Configure & View Metrics (Live only)
  - View Log files
  - Manage logback logger levels
  - *View and use JMX beans via jolokia*
  - View Thread dump
  - View HTTP request Traces (Live only)
  - *View history of registered applications*
  - Notifications is the top notch key feature (can notify in many ways)
  - All in all it’s a Live monitoring and alerting solution.
- Cons
  - Time series data is not available. It doesn’t store data.

Spring Boot Admin Server - Notifications

- Reminder Notifications
- Filtering Notifications
- Mail Notifications
- PageDuty Notifications
- OpsGenie Notifications
- HipChat Notifications
- Slack Notifications
- Let’s Chat Notifications
- Microsoft Teams Notifications
- Telegram Notifications
- Discord Notifications

Spring Boot Admin Server - Steps

- Step-01: Spring Boot Admin server - Base project setup
- Step-02: Point Spring Boot Client Application to Admin Server
- Step-03: Test the features in Spring Boot Admin Server

Spring Boot

Micrometer

Spring Boot - Micrometer

- Micrometer is the metrics collection facility included in Spring Boot 2’s Actuator.
- Micrometer is a *dimensional-first* metrics collection facade whose aim is to allow us to time, count, and gauge your code with a vendor neutral API.
- Through classpath and configuration, we can select one or several monitoring systems to export our metrics data.
- It has also been backported to Spring Boot 1.5, 1.4, and 1.3 with the addition of another dependency.

Spring Boot - Micrometer

- A single Micrometer Timer is capable of producing time series related to throughput, total time, maximum latency of recent samples, pre-computed percentiles, percentile histograms, and SLA boundary counts.
- The change to Micrometer arose out of a desire to better serve a wave of dimensional monitoring systems (think Prometheus, Datadog, Wavefront, SignalFx, Influx).
- Spring Boot is enabling us to choose one or more monitoring systems to use today, and change our mind later as our needs change without requiring a rewrite of our custom metrics instrumentation.
- https://micrometer.io/docs

Monitoring Systems - Supported

AppOptics	Instana
Atlas	JMX
Datadog	KairosDB
Dynatrace	New Relic
Elastic	Prometheus
Gangila	SignalFx
Graphite	StatsD
Humio	WaveFront
Influx	Simple (In memory backend used as fallback option)

Micrometer - Implementation Steps

- Step-01: New GIT branch using IDE
- Step-02: Add Micrometer dependency for Metrics and view metrics using simple in-memory backend.
- Step-03: Integrate with JMX and view metrics in JConsole using JMX
- Step-04: Integrate with AppOptics to export metrics and View metrics in AppOptics (Solarwinds product).
- Step-05: Perform Tests using POSTMAN Collection Runner
- Step-06: Commit & Push code via IDE

Thank You