**SOA 2.0**

**Microservices Development Cookbook**

Document Owner : SOA CoE, HKIT

Last Review : 6th, December, 2017

Version : 0.4

**Version History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date (DD/MM/YYYY)** | **Description of Change** | **Reviewed By** | **Approved By** |
| 0.1 | 03/10/2017 | Initial Draft | Chandi Prasad Ojha | Angus Chan |
| 0.2 | 08/11/2017 | Config Server Documentation | Chandi Prasad Ojha | Angus Chan |
| 0.3 | 10/11/2017 | Add concourse and New Relic document |  |  |
| 0.4 | 06/12/2017 | Changes made to REST Error Handling, Annotation based validation, JPA, JDBC, Accessing Hive related sections |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Table of Contents**

[1 Introduction 6](#_Toc501012941)

[1.1 Purpose 6](#_Toc501012942)

[1.2 Acronyms & Definitions 6](#_Toc501012943)

[2 Spring Boot Basics 7](#_Toc501012944)

[2.1 Basic Spring Boot Web App 7](#_Toc501012945)

[2.2 Managing configurations 9](#_Toc501012946)

[2.3 Decorating Java POJO using Lombok 11](#_Toc501012947)

[2.4 Annotations based validations 12](#_Toc501012948)

[2.5 Application Logging Standardization 16](#_Toc501012949)

[2.6 REST Error Handling 19](#_Toc501012950)

[2.7 Accessing Datastore using JPA 22](#_Toc501012951)

[2.8 Accessing database using JdbcTemplate 25](#_Toc501012952)

[3 Development 29](#_Toc501012953)

[3.1 Externalizing Configuration with Config Server 29](#_Toc501012954)

[3.2 Registering Services with Service Registry 32](#_Toc501012955)

[3.3 Load Balanced Client with Feign and Ribbon 33](#_Toc501012956)

[3.4 Fault Tolerant Services with Hystrix 34](#_Toc501012957)

[3.5 User provided Service Instance 35](#_Toc501012958)

[3.6 Single Sign On 37](#_Toc501012959)

[3.7 Generating Authorization Tokens 38](#_Toc501012960)

[3.8 Messaging with RabbitMQ 40](#_Toc501012961)

[3.9 Working with Pivotal Cloud Cache 41](#_Toc501012962)

[3.10 Working with Redis Cache 42](#_Toc501012963)

[3.11 Accessing Hive from Microservices 43](#_Toc501012964)

[3.12 Utilizing APIGEE and Edge Micro gateway 44](#_Toc501012965)

[4 Testing 45](#_Toc501012966)

[4.1 Unit Testing Micro Services 45](#_Toc501012967)

[4.2 Mock Services with Mockito 46](#_Toc501012968)

[5 Build and Deployment 47](#_Toc501012969)

[5.1 Manage project dependencies 47](#_Toc501012970)

[5.2 Deploying Micro Services to PCF 52](#_Toc501012971)

[6 Operational Monitoring 53](#_Toc501012972)

[6.1 Analyze Logs in Sumo Logic 53](#_Toc501012973)

[6.2 Monitoring Apps with New Relic 54](#_Toc501012974)

[7 Appendix 55](#_Toc501012975)

# Introduction

## Purpose

The purpose of the document is to provide quick knowledge on Spring Boot, Microservices based on several topics covering all stages of SDLC. The topics are categorized into multiple sections as follows

1. Spring Boot Basics
2. Microservices Development
3. Testing
4. Build, Package & Deployment
5. Monitoring

This would help the developers to familiarise themselves with microservices concepts, best practices, design and standardize the approach for specific scenario’s that frequently occur during implementation.

Each topic contains the following sections,

1. Topic overview - explaining the problem or best practice, the
2. Prerequisites - that are required before proceeding to implementation, a
3. Detailed Description on why we need this and how this helps provides with crucial implementation notes along with code, configuration snippets,
4. References - detailed documentation reference, source code repository reference if any
5. Next steps - guiding where to go from here and learn related concepts.

This is living document, new topics would be getting added as and when an approach or best practices is being introduced.

## Acronyms & Definitions

|  |  |
| --- | --- |
| **Term** | **Definition** |
| AMQP | Advanced Message Queuing Protocol |
| API | Application Programming Interface |
| CLI | Command Line Interface |
| IDE | Integrated Development Environment |
| SDLC | Software Development Life Cycle |
| STS | Spring Tool Suite |
| UR | User Requirements |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locator |
|  |  |

# Spring Boot Basics

## Basic Spring Boot Web App

|  |
| --- |
| **Overview**  Spring boot comes with several starter project templates to jump start creation of Spring Boot project and automatically manage library / jar dependencies based on the chosen template. The starter templates include connecting to database, NoSQL datastore, web apps, spring cloud related services etc. |
| **Pre-Requisites**   1. STS IDE 2. Browser / Rest Client 3. Java 8 |
| **Feature Description**  This helps the developer jump start development and familiarize with spring boot based web application exposing a simple REST service in STS using the Spring Initializer web service (http://start.spring.io) |
| **Implementation Notes**  Create a new Spring Boot project in STS using the Spring Boot -> Spring Starter Project. Provide the Group, Artifact, Version, Package details and select “Web” as the spring starter dependency.  The spring boot starter parent pom inherits dependency management from spring-boot-dependencies and helps to manage version, dependencies, plugin configuration etc. for all the child project and modules  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.7.RELEASE</version>  </parent>  spring-boot-starter-web inherits Tomcat, Hibernate Validation, Jackson, Spring Web and Spring WebMvc as dependencies (Run *mvn dependency:tree* to see the list of Jars)  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  Create a new class under src/main/java as shown below  @SpringBootApplication  public class HelloSpringBootApplication {  public static void main(String[] args) throws Exception {  SpringApplication.run(HelloSpringBootApplication.class, args);  }  }  @RestController  @EnableAutoConfiguration  public class HelloSpringBootController {  @RequestMapping("/")  String home() {  return "Hello Spring Boot!!!";  }  }  The *@RestController* stereotype annotation provides hints to consider it while handling web requests.  The *@EnableAutoConfiguration* annotation tells spring boot to look at the dependencies and configure. In this case it will assume that this is a web application based on tomcat and spring webmvc dependencies and would setup the project accordingly  The *@RequestMapping* annotation provides the route information that any requests to root path “/” should be handled by home method  Run the application by right clicking the project and selecting Run As -> Spring Boot App  Check the console for any errors and information about the Tomcat initialization, Ports utilized and request path mappings etc.  Now open the browser / rest client and point towards <http://localhost:8080/> and this should print the output as returned by the home method |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git> (Cookbook-RestSample) |
| **Next Steps**   1. Logging 2. Securing endpoints 3. Accessing Datastore 4. Documenting endpoints |

## Managing configurations

|  |
| --- |
| **Overview**  The @Value annotation can be used for more simplistic cases for injecting the configuration properties. But when we have to deal with multiple properties or hierarchical data @Value annotation may not be effective. Also we need to ensure that the application is supplied with required configurations in all environments and this should be validated so that any missing information can be easily identified and added. |
| **Pre-Requisites**   1. STS IDE 2. Browser / Rest Client 3. Java 8 |
| **Feature Description**  For dealing with multiple properties or hierarchical data, spring boot offers alternate approach which allows beans to handle the application configuration aspect. This helps to standardize the application configuration management and access in a uniform way across projects and also come up with maven project starter templates to auto generate the classes and configuration files. It also facilitates validation of configuration using JSR-303, javax.validation constraints annotation on the beans. This helps to reduce a lot of conditions in our code and makes it look much concise and clean |
| **Implementation Notes**  Add spring-boot-configuration-processor as dependency  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-configuration-processor</artifactId>  </dependency>  Utilize @ConfigurationProperties annotation with which we can isolate the configuration properties into a separate POJO class as shown below  @Component  @ConfigurationProperties(prefix="appconfig")  public class AppConfig {    private String ip;  private String port;  private ExtService extService;    public static class ExtService {  private String url;    }  //Ensure getters and setters are implemented  }  The application YML configuration should be as follows  appconfig:  extService:  url: http://localhost:8081/externalservice  ip: 10.10.20.20  port: 9081  Inject / Autowire the AppConfig object in required classes and obtain the value accordingly  @Autowired  **private** AppConfig appConfig;  logger.log(“IP Value is: ” + appConfig.getIp());  For enabling validation on the configuration please refer to [Annotation based validations](#_Annotations_based_validations)  This ensures that the application configuration is validated on the application start-up and the application would fail to start in case any of these validations fail  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  APPLICATION FAILED TO START  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Description:  Binding to target AppConfig [ip=10.10.20.20, port=9081, extservice=com.manulife.poc.sso.client.AppConfig$ExtService@191ae03f] failed:  Property: appconfig.from  Value: LKJHYU  Reason: length must be between 10 and 25  Action:  Update your application's configuration |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git>(Cookbook-Config project) |
| **Next Steps**   1. Configuration Validation 2. Config Server |

## Decorating Java POJO using Lombok

|  |
| --- |
| **Overview**  In Java, sometimes the coding is too verbose and repetitive. For example we rely on the IDE to auto generate getters, setters, toString, hashCode, no argument constructors, constructors with specific fields etc. Whenever there is a need to change, we need to regenerate the source code again. With Lombok annotations these scenarios are taken care automatically |
| **Pre-Requisites**   1. STS IDE with Maven 2. Project Lombok dependency 3. REST Client |
| **Feature Description**  Once project Lombok is added as dependency, it gets plugged into the build process and it generates bytecodes into the class files as per the annotations that we utilized as part of the code. Project Lombok also provides IDE plugins for local development as well |
| **Implementation Notes**  Yet To Start on the POC |
| **References**  NA |
| **Next Steps**   1. Understand using it in both IDE and Maven Builds to see any difficulties arising out of it |

## Annotations based validations

|  |
| --- |
| **Overview**  All services have the requirement to validate the input parameters before processing it. Writing validation logic is a cumbersome and tiring activity which involves more and more if..else condition logic. |
| **Pre-Requisites**   1. STS IDE with Maven 2. Browser / Rest Client 3. Java 8 |
| **Feature Description**  JSR-303 (javax.validation) provides constraints using annotation on the properties. This helps to reduce a lot of conditions in our code and makes it look much concise and clean. Hibernate Validator extends JSR-303 and provides much more sophisticated validation option for customizing validations, error messages, defining groups etc. |
| **Implementation Notes**  For web starter template, hibernate validator comes as dependency automatically and hence no need to provide explicit dependency, if not available include the dependency as follows  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-validation</artifactId>  </dependency>  Annotate the fields as follows  public class StoreRequest {  @NotNull(message = "Task name must not be blank!")  @Pattern(regexp = "^[A-Za-z]\*[A-Za-z-'. ]\*[A-Za-z]\*$")  @Size(min = 3, max = 30)  private String name;  @NotNull  @Size(min = 1, max = 1)  @Pattern(regexp = "^[MFU]$")  private String sex;  @NotNull  @Past  @DateTimeFormat(pattern = "dd/MM/yyyy")  private Date dateOfBirth;  @Email  private String email;  @ContactNumberConstraint  private String phone;  @NotEmpty  @Pattern(regexp = "\\d{1,10}")  private String int1To16Digit;  @Valid  private InnerStoreRequest innerStoreRequest;  // getters and setters  }  public class StoreListRequest {  @NotBlank  private String name;  @Valid  private List<InnerStoreRequest> innerStoreRequestList;  // getters and setters  }  Update the controller class to include the @Valid to invoke the validation:  @RestController  public class SampleController {  @PostMapping(value = "/people")  public ResponseEntity<StoreResponse> store(@Valid @RequestBody final StoreRequest request) {  StoreResponse response = new StoreResponse();  response.setSuccess(true);  return new ResponseEntity<>(response, HttpStatus.OK);  }  @PostMapping(value = "/peopleList")  public ResponseEntity<StoreResponse> storeList(@Valid @RequestBody final StoreListRequest request) {  StoreResponse response = new StoreResponse();  response.setSuccess(true);  return new ResponseEntity<>(response, HttpStatus.OK);  }  }  The @Valid ensures that the validations are run recursively including inner objects and in case the validations fail, the corresponding messages are thrown accordingly  There could be scenarios where we need to validate 2 or more parameters based on different criteria, creating our own custom validation annotations, validate Objects or validate a list of objects; For these cases we can also create our own customized validations using the constraint annotation and validator class.  The following example highlights Validating Objects  @Component  public class StoreListRequestValidator implements Validator {  @Override  public boolean supports(Class<?> clazz) {  return StoreListRequest.class.equals(clazz);  }  @Override  public void validate(Object target, Errors errors) {  StoreListRequest storeListRequest = (StoreListRequest) target;  for (int i = 0; i < storeListRequest.getInnerStoreRequestList().size(); i++) {  InnerStoreRequest innerStoreRequest = storeListRequest.getInnerStoreRequestList().get(i);  if (StringUtils.isEmpty(innerStoreRequest.getCheckInner())) {  errors.rejectValue("innerStoreRequest[" + i + "].checkInner", "Should not be empty.");  errors.popNestedPath();  }  }  }  }  The following showcases creating a custom validation annotation  Please refer the request class shown above “StoreRequest”.  @ContactNumberConstraint  private String phone;  Custom interface class for the above annotation: ContactNumberConstraint  @Constraint(validatedBy = ContactNumberValidator.class)  @Target({ ElementType.METHOD, ElementType.FIELD })  @Retention(RetentionPolicy.RUNTIME)  @Documented  public @interface ContactNumberConstraint {  String message() default "Invalid phone number";  Class<?>[] groups() default {};  Class<? extends Payload>[] payload() default {};  }  Custom validator class for custom validation annotation: ContactNumberConstraint  @Component  public class ContactNumberValidator implements ConstraintValidator<ContactNumberConstraint, String> {  @Override  public void initialize(ContactNumberConstraint contactNumberConstraint) {  // Overrided method.  }  @Override  public boolean isValid(String contactNumber, ConstraintValidatorContext cxt) {  return contactNumber != null && contactNumber.matches("[0-9]+") && contactNumber.length() == 10;  }  }  Please refer the REST Error Handling section to know more details about custom request validation error response. |
| **References** <https://git.ap.manulife.com/scm/hksoa/poc.git> (Spring Boot Validation) |
| **Next Steps**   1. Documenting endpoints 2. Custom request validation error response |

## Application Logging Standardization

|  |
| --- |
| **Overview**  Logging is a very important aspect for any application to understand and track the application internal information. It also helps the developer to debug the application during development. Sometimes the application might not behave as expected and logs would be very much helpful in such situations to analyze and understand the issue |
| **Pre-Requisites**   1. STS IDE with Maven 2. Java 1.8 3. Simple REST based application exposing some endpoints 4. Browser / REST Client |
| **Feature Description**  Spring Boot provides very good support for logging, default configurations are provided for JUL, JCL, Log4J2, Logback etc and supports customization as well. It uses commons logging internally but leaves the underlying log implementation open so that the team can decide on their respective framework for logging. |
| **Implementation Notes**  Always use SLF4J, it has become a de-facto standard since it provides an abstraction over all other frameworks and it’s really easy to switch between logging frameworks if required. The spring-boot-starter template includes spring-boot-starter-logging template which in turn includes dependencies for SLF4J and Logback dependencies with appropriate SLF4J wrappers for other logging libraries.  **import** org.slf4j.Logger;  **import** org.slf4j.LoggerFactory;  @RestController  **public** **class** CookbookRestSampleController {  **private** **static** **final** Logger ***logger*** = LoggerFactory.*getLogger*(CookbookRestSampleController.**class**);  @RequestMapping("/")  String home() {  ***logger***.error("Message logged at ERROR level");  ***logger***.warn("Message logged at WARN level");  ***logger***.info("Message logged at INFO level");  ***logger***.debug("Message logged at DEBUG level");  **return** "Hello Spring Boot!!!";  }  }  By running the application we can observe that even though we did not include any dependencies the logs started appearing (as shown below) but the logs does not include DEBUG statements since the default configuration LEVEL is INFO  [2m2017-10-02 12:34:25.480[0;39m [31mERROR[0;39m [35m1336[0;39m [2m---[0;39m [2m[nio-8080-exec-1][0;39m [36mc.m.p.b.CookbookRestSampleController [0;39m [2m:[0;39m Message logged at ERROR level  [2m2017-10-02 12:34:25.480[0;39m [33m WARN[0;39m [35m1336[0;39m [2m---[0;39m [2m[nio-8080-exec-1][0;39m [36mc.m.p.b.CookbookRestSampleController [0;39m [2m:[0;39m Message logged at WARN level  [2m2017-10-02 12:34:25.480[0;39m [32m INFO[0;39m [35m1336[0;39m [2m---[0;39m [2m[nio-8080-exec-1][0;39m [36mc.m.p.b.CookbookRestSampleController [0;39m [2m:[0;39m Message logged at INFO level  To customize the logging level just updating the application properties would be sufficient. Below have provided the Logback configuration to showcase updating the log pattern along with log level  <?xml version="1.0" encoding="UTF-8"?>  <configuration>  <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">  <!-- Log message format -->  <encoder>  <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n  </pattern>  </encoder>  </appender>  <!-- Setting the root level of logging to INFO -->  <root level="debug">  <appender-ref ref="CONSOLE" />  </root>  </configuration>  This results in the following output  12:53:34.254 [http-nio-8080-exec-3] ERROR c.m.p.b.CookbookRestSampleController - Message logged at ERROR level  12:53:34.254 [http-nio-8080-exec-3] WARN c.m.p.b.CookbookRestSampleController - Message logged at WARN level  12:53:34.254 [http-nio-8080-exec-3] INFO c.m.p.b.CookbookRestSampleController - Message logged at INFO level  12:53:34.254 [http-nio-8080-exec-3] DEBUG c.m.p.b.CookbookRestSampleController - Message logged at DEBUG level  Configure logging.level as part of the application properties / yml file to override the default level for packages as required  spring:    logging:      level:        com.manulife.hk: DEBUG  Following are some of the best practices for logging   1. Instead of just enabling DEBUG as default log level, apply different log levels for different packages as applicable 2. Log at one place and do not re throw exception after logging since this would result in multiple stack traces introducing lots of clutter in the overall logs 3. Log meaningful information, log method calls along with crucial method parameters 4. Do NOT log any sensitive information 5. Set the log levels appropriately (Default WARN, Service Layer INFO, Dao Layer DEBUG etc.) 6. Make use of log placeholders appropriately 7. Finalize one log pattern and utilize the same across all applications |
| **References** <https://git.ap.manulife.com/scm/hksoa/poc.git> |
| **Next Steps**   1. Analyze logs in Sumo Logic 2. Spring Sleuth, Analyze logs using span id and trace id |

## REST Error Handling

|  |
| --- |
| **Overview**  Handling exceptions, errors correctly and providing meaningful message helps the service client to respond properly. Returning stack traces and unneeded information would be hard to understand and also its not useful from API client perspective |
| **Pre-Requisites**   1. STS IDE with Maven 2. Java 1.8 3. RESTClient |
| **Feature Description**  Spring Boot provides several options for handling exception. We would come up with our own error response mechanism and utilize the same for different error scenario’s. |
| **Implementation Notes**  The error / exception handling mechanism is very much coupled with the HTTP Response mechanism. The below table lists down the most common error codes that needs to be handled by the micro services. It may not be necessary to handle all the error codes for all the services since these are very much scenario specific   |  |  |  | | --- | --- | --- | | **Code** | **Description** | **Comment** | | 200 | Ok | Request processed successfully | | 201 | Created | Resource created, response contains reference to new resource | | 204 | No Content | Request processed successfully but has not resulted in new resource creation | | 304 | Not Modified | Resource has not been modified since the version specified by the request headers | | 400 | Bad Request | Input s not as per expectation / data validations failed | | 401 | Unauthorized | Similar to 403 Forbidden, but specifically for use when authentication is required and has failed | | 403 | Forbidden | Need account or proper permission to access resource | | 404 | Not Found | Requested resource not found | | 405 | Method Not Allowed | Request method not supported | | 409 | Conflict | Edit conflict between multiple simultaneous updates | | 410 | Gone | Requested resource no longer available and will not be available again | | 415 | Unsupported Media Type | Request format not supported | | 500 | Internal Server Error | Backend Service / SQL / Queue / Cache / Unknown Application Error | | 503 | Service Unavailable | Service down or not available |   RestControllerAdvice would be used to handle all the exceptions at one place and handle the log and HTTP response mechanism accordingly. The below code snippet shows at high level the basic structure of the advice along with some sample for handling different scenario’s  @RestControllerAdvice  public class ExceptionHandlerAdvice {  private ResponseEntity<ApiErrorResponse> buildResponseEntity(HttpStatus httpStatus,  ApiErrorResponse errorResponse) {  return new ResponseEntity<>(errorResponse, httpStatus);  }  }  400 (Bad Request) sample code:  @ExceptionHandler(MethodArgumentNotValidException.class)  public ResponseEntity<ApiErrorResponse> methodArgumentNotValidException(MethodArgumentNotValidException ex) {  BindingResult bindingResult = ex.getBindingResult();  List<ApiValidationError> apiValidationErrorList = new ArrayList<>();  List<org.springframework.validation.FieldError> fieldErrors = bindingResult.getFieldErrors();  for (org.springframework.validation.FieldError fieldError : fieldErrors) {  ApiValidationError apiValidationError = new ApiValidationError(fieldError.getField(),  fieldError.getDefaultMessage());  apiValidationErrorList.add(apiValidationError);  }  List<ApiError> apiErrorList = new ArrayList<>();  for (ApiValidationError apiValidationErrorTemnp : apiValidationErrorList) {  ApiError apiError = apiValidationErrorTemnp;  apiErrorList.add(apiError);  }  ApiErrorResponse apiErrorResponse = new ApiErrorResponse("400", HttpStatus.BAD\_REQUEST.toString(),  apiErrorList);  return buildResponseEntity(HttpStatus.BAD\_REQUEST, apiErrorResponse);  }  @ExceptionHandler(value = { HttpMessageNotReadableException.class })  public ResponseEntity<ApiErrorResponse> httpMessageNotReadableException(HttpMessageNotReadableException ex) {  if (ex.getCause() != null && (ex.getCause() instanceof UnrecognizedPropertyException)) {  String exceptionStackTrace = ex.getCause().toString();  LOG.error(exceptionStackTrace);  ApiErrorResponse response = new ApiErrorResponse("400", "Bad Request", ex.getCause().getLocalizedMessage());  return buildResponseEntity(HttpStatus.BAD\_REQUEST, response);  } else {  String exceptionStackTrace = ex.toString();  LOG.error(exceptionStackTrace);  ApiErrorResponse response = new ApiErrorResponse("400", "Bad Request", ex.getLocalizedMessage());  return buildResponseEntity(HttpStatus.BAD\_REQUEST, response);  }  }  Please refer to the source code for all the other exceptions that are handled and the equivalent HTTP response that needs to be returned |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git> - (ExceptionHandlerDemo) |
| **Next Steps**   1. Improvise the source code to provide scenario to cover the remaining HTTP Status Codes |

## Accessing Datastore using JPA

|  |
| --- |
| **Overview**  Java Persistence API (JPA) is a data access technology that allows mapping objects to relational databases tables. Spring Data JPA is the implementation of JPA for simplifying operation like data accessing, querying, pagination and removes lots of boilerplate codes |
| **Pre-Requisites**   1. STS IDE with Maven 2. Java 1.8 3. RESTClient 4. Datastore |
| **Feature Description**  Spring data project provides the abstract repositories (CrudRepository, JPARepository etc.). We need to extend any of these repositories (based on specific requirement) and would just be providing the methods we need as part of the interface. These methods are implemented at run-time by the spring container to perform the appropriate database operations. This reduces the amount of boilerplate code required to write data access layers |
| **Implementation Notes**  To implement JPA Repository add following dependencies in pom.xml  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  Spring Data JPA focuses on using JPA to store data in a relational database. Its most compelling feature is the ability to create repository implementations automatically, at runtime, from a repository interface. Define JPA interface which extend JpaRepository. You can include your repository methods in this interface. By extending the interface we get the most relevant CRUD methods for standard data access available in a standard DAO out of the box.  public interface ModelJpaRepository extends JpaRepository<Model, String> {  List<Model> findByName(String value);  }  If the managed entity has a name field (the Java Bean standard getName /setName methods), we’ll define the findByNamemethod in the interface.this will automatically generate the correct query  @Entity  @Table(name = "MODEL")  public class Model {    @Id  @GeneratedValue  @Column(name = "ID", nullable = false)  private long id;    @Column(name = "VAL", nullable = false)  private String value;    public long getId() {  return id;  }  public void setId(long id) {  this.id = id;  }  public String getValue() {  return value;  }  public void setValue(String value) {  this.value = value;  }  }  The Model class is annotated with @Entity, indicating that it is a JPA entity. For lack of a @Table annotation, it is assumed that this entity will be mapped to a table named Model.  The Model id property is annotated with @Id so that JPA will recognize it as the object’s ID. The id property is also annotated with @GeneratedValue to indicate that the ID should be generated automatically.  By default, Spring Boot will enable JPA repository support and look in the package (and its subpackages) where @SpringBootApplication is located. If your configuration has JPA repository interface definitions located in a package not visible, you can point out alternate packages using @EnableJpaRepositories and its type-safe  **To Test Application**  @RunWith(SpringJUnit4ClassRunner.class)  @SpringApplicationConfiguration(SpringbootJpaDemoApplication.class)  public class SpringbootJPADemoApplicationTests {  @Autowired  private UserRepository userRepository;  @Autowired  private ModelJpaRepository modelJpaRepository;  @Test  public void SaveModel() {  Model m=new Model();  m.setValue("SAB");  modelJpaRepository.save(m);  }  @Test  public void findAllUsersByJpa() {  List<Model> model = modelJpaRepository.findAll();  assertNotNull(model);  assertTrue(!model.isEmpty());  }  }  **Note:** The sample project uses Embedded Database |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git> - springboot-jpa-poc |
| **Next Steps**   1. Connection Pooling 2. Transaction Management 3. Update the POC to use MySQL (PCF Managed Services) |

## Accessing database using JdbcTemplate

|  |
| --- |
| **Overview**  JdbcTemplate provides very flexible mechanism to connect to the database and execute SQL queries and thereby reduces the boilerplate code in DAO classes. The JDBC Template class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. |
| **Pre-Requisites**   1. STS IDE with Maven 2. Java 1.8 3. RESTClient |
| **Feature Description**  Spring Jdbc Template eliminates all the problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time. It is the central class in the Spring JDBC support classes. It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection. We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database. |
| **Implementation Notes**  To access database using Jdbc template we can either use Embedded database or we can use Non-Embedded databases like MySQL, Oracle or PostgreSQL etc.  In Embedded database the .sql files are used to handle the table creation and data manipulation.  For example we have schema.sql and data.sql files.  **Schema.sql**  CREATE TABLE users (  id int(11) NOT NULL AUTO\_INCREMENT,  name varchar(100) NOT NULL,  email varchar(100) DEFAULT NULL,  PRIMARY KEY (id)  );  CREATE TABLE MODEL  (  ID NUMBER(19) NOT NULL,  VAL VARCHAR2(50) NOT NULL,  );  **data.sql**  insert into users(id, name, email) values(1,'Sebitha','sb@gmail.com');  insert into users(id, name, email) values(2,'ALEE','alee@gmail.com');  insert into users(id, name, email) values(3,'Reddy','reddy@gmail.com');  In Non Embedded database you should specify the database details in application.yml file.  **application.yml file**  spring.datasource.url=  spring.datasource.username=  spring.datasource.password=  spring.datasource.driver-class-name=  By using SpringBoot we can take advantage of auto configuration feature and eliminate the need to configure beans by ourselves.  **Add following dependencies in pom.xml file**  <dependency>      <groupId>org.springframework.boot</groupId>      <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  If you are using emebedded database then add following dependencies  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  <exclusions>  <exclusion>  <groupId>org.apache.tomcat</groupId>  <artifactId>tomcat-jdbc</artifactId>  </exclusion>  </exclusions>  </dependency>  **Define a Model Class – User**  package com.manulife.hk.poc;  public class User  {  private Integer id;  private String name;  private String email;    public User() {  }  public User(Integer id, String name, String email) {  this.id = id;  this.name = name;  this.email = email;  }  public Integer getId() {  return id;  }  public void setId(Integer id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public String getEmail() {  return email;  }  public void setEmail(String email) {  this.email = email;  }    }  Define Repository class to inject **JdbcTemplate**into **UserRepository**as follows:  @Repository  public class UserRepository  {  @Autowired  private JdbcTemplate jdbcTemplate;    @Transactional(readOnly=true)  public List<User> findAll() {  return jdbcTemplate.query("select \* from users", new UserRowMapper());  }  @Transactional(readOnly=true)  public User findUserById(int id) {  return jdbcTemplate.queryForObject("select \* from users where id=?", new Object[]{id}, new UserRowMapper());  }  public User create(final User user) {  final String sql = "insert into users(name,email) values(?,?)";    KeyHolder holder = new GeneratedKeyHolder();  jdbcTemplate.update(new PreparedStatementCreator() {  @Override  public PreparedStatement createPreparedStatement(Connection connection)  throws SQLException {  PreparedStatement ps = connection.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);  ps.setString(1, user.getName());  ps.setString(2, user.getEmail());  return ps;  }  }, holder);  int newUserId = holder.getKey().intValue();  user.setId(newUserId);  return user;  }  }  class UserRowMapper implements RowMapper<User>  {  @Override  public User mapRow(ResultSet rs, int rowNum) throws SQLException {  User user = new User();  user.setId(rs.getInt("id"));  user.setName(rs.getString("name"));  user.setEmail(rs.getString("email"));    return user;  }    }  **Note:** The sample project used Embedded database for the implementation |
| **References :** <https://git.ap.manulife.com/scm/hksoa/poc.git> (springboot-jdbc-poc) |
| **Next Steps :**   1. Connection pooling 2. Transaction handling 3. Update the POC to use MySQL (PCF Managed Services) |

# Development

## Externalizing Configuration with Config Server

|  |
| --- |
| **Overview**  As per the general guidelines, microservices applications shouldexternalize the configuration related parameters that varies between deployment environments (Dev, Test, Prod etc.). This helps in creating immutable artifacts which can be deployed to different environments automatically with CI/CD pipelines |
| **Pre-Requisites**   1. STS IDE with Maven 2. PCF Config Server 3. Java 1.8 4. REST Client |
| **Feature Description**  As part of this exercise   1. Develop a local Config server in development environment backed by Git repository 2. Develop client application which connects to the local Config server and load the properties based on the profile / environment. 3. Create a Config server provided by PCF as part of market place in Sandbox environment 4. Update the sandbox configuration of the client application to work with PCF Config Server in Sandbox environment |
| **Implementation Notes**  There are three main parts to successfully configuring the config server in both local and cloud.  First Part – Setting up a local config server so developer can develop and test their application using this config server. Please follow the below steps:   1. Update the pom.xml with the below:    1. Add the dependency <dependency> <groupId>org.springframework.cloud</groupId> <artifactId>spring-cloud-config-server</artifactId> </dependency>    2. <dependency> <groupId>org.springframework.boot</groupId> <artifactId>spring-boot-starter-actuator</artifactId> </dependency> 2. In the main method class use the below annotation: @EnableConfigServer 3. Add application.yml file in the src/main/resources. Sample content of the file would be: server:  port: 8888 management:  security:  enabled: false security:  basic:  enabled: false spring:  application:  name: config-server  cloud:  config:  server:  git:  uri: <https://git.ap.manulife.com/scm/hksoa/poc.git>  username: Mazin M Ismail  password: <password>  search-paths: ConfigServer/config  encrypt:  enabled: false Please note that all the property file would reside in the above mentioned git repository. 4. You can test the local config server with the below URL: <http://localhost:8888/env>. 5. If you need specific property file that resides in this git repository, you can fetch the property file as shown below: http://localhost:8888/awdmanagement-service.yml   Second Part - Necessary changes to be made to the application so it can pick the property file from repository. Please follow the below steps:   1. Update the pom.xml of the <application name>-service with the below:    1. Update/Add the property <spring-cloud.version>Dalston.RELEASE</spring-cloud.version>    2. Add the dependency <dependency> <groupId>io.pivotal.spring.cloud</groupId> <artifactId>spring-cloud-services-starter-config-client</artifactId> </dependency> 2. Remove application.yml or any property file associate with <application name>-service as now we will not require the properties to be picked from within the application. 3. Update/Add bootstrap.yml file in the src/main/resources. Sample content of the file would be:  spring:  application:  name: awdmanagement-service  cloud:  services:  registrationMethod: route management:  security:  enabled: false security:  basic:  enabled: falsePlease not that profile could also be mentioned. It will look for the property file having the naming convention <application-name>-<profile-name>.yml An application deployed to PCF can have one or more URLs, or “routes,” bound to it. If you specify the route registration method, the application will be registered with the Service Registry instance using the first of these routes from the uris list in the application’s VCAP\_APPLICATION environment variable. 4. When you start your application you will notice logs containing “Fetching config from server at: <http://localhost:8888>”. 5. You should also see details of the application properties and other information from the below mentioned URL: http://localhost:<port>/env   Third Part – Setting up of the config server in PCF. Please find the steps below:   1. Navigate to Pivotal Apps Manager – Marketplace and select Config Server. Click on “SELECT THIS PLAN” 🡪 Fill the details (Make sure that the instance name is unique) 🡪 Click on “Add”. 2. Login to PCF CLI and use the update-service CF command to configure the git repository details with config server. Once service is updated, Click on manage link config server instance UI. The git configuration would be displayed along with Config Server Online message 3. Deploy your application in PCF, sample git command is shown below: *cf push onboard-service -d apps.eas.pcf.manulife.com --hostname onboard-service-mmisand -m 1024m -p "D:\Users\mazin m ismail\git\onboard-management\onboard-service\target\onboard-service.jar"* 4. Bind your application with the newly created Config Server using the Bind button and restart your application. |
| **References:** <https://git.ap.manulife.com/projects/HKSOA/repos/poc/browse/ConfigServer> |
| **Next Steps**   1. Enabling encryption and decryption (Both Server and Client Side) 2. Explain @RefreshScope to update the properties at runtime with example 3. Utilizing spring.cloud.config.label, spring.profiles.active 4. Connecting to Cloud Config Server Instance from Local IDE using Spring Cloud Connectors |

## Registering Services with Service Registry

|  |
| --- |
| **Overview**  Service registry is basically a server component which allows micro services to find and communicate with each other without hard coding hostname and port. For this to happen, all the service instances needs to register itself with Service Registry and deregister on shutdown |
| **Pre-Requisites**   1. STS IDE with Maven 2. Eureka (for local development) and PCF Service Registry (Sandbox) 3. Java 1.8 4. REST Client |
| **Feature Description**  As part of this exercise   1. Setup a local Eureka server 2. Develop client applications which would register themselves with local Eureka server instance 3. Create Service Registry provided by PCF as part of market place in Sandbox environment 4. Update the sandbox configuration of the client application to work with PCF Service Registry in Sandbox environment |
| **Implementation Notes**  The PCF GoRouter more or less does the same thing and Service Discovery may not be required for all the services  Yet to be started |
| **References**  NA |
| **Next Steps**   1. Identify the scenario’s where Service Discovery (Managed Services) needs to be used and where it is not required |

## Load Balanced Client with Feign and Ribbon

|  |
| --- |
| **Overview**  Micro services runs in a virtual / container based environments and the number of service instances change dynamically as new instances are created shutdown. Hence it would be difficult to make requests to dynamically changing services instances |
| **Pre-Requisites**   1. STS IDE with Maven 2. Eureka (for local development) and PCF Service Registry (Sandbox) 3. Client side service discovery with Feign client and Ribbon 4. Java 1.8 5. REST Client |
| **Feature Description**  As part of this exercise   1. Develop client application (A) which would register itself with Eureka server 2. Develop another client application (B) which would register itself with Eureka server and invoke the other client application (A) as well. This would be implemented with Feign client 3. Increase and decrease the number of instances of (A) to understand the load balancing aspect |
| **Implementation Notes**  The POC is to show case how to use Feign client along with Service Registry and not as how it is followed currently (Using it as component to reduce Rest Http implementation complexity  Yet to be started |
| **References**  NA |
| **Next Steps**  NA |

## Fault Tolerant Services with Hystrix

|  |
| --- |
| **Overview**  Micro services depend on many other micro services and 3rd party services. If the services are not isolated from dependency failures, the service itself is at risk of being taken down. When services faces with high volume requests, a single dependent service becoming slow can exhaust all application resources quickly. |
| **Pre-Requisites**   1. STS IDE with Maven 2. Eureka (for local development) and PCF Service Registry (Sandbox) 3. Client side service discovery with Feign client, Ribbon, Hystrix Fallback 4. Java 1.8 5. REST Client |
| **Feature Description**  Hystrix library is designed to control access to dependant and 3rd party services providing greater tolerance of latency and failure. Hystrix provides thread and semaphore isolation with fallbacks, circuit breakers, request caching, collapsing, monitoring and configuration |
| **Implementation Notes**  Yet to be started |
| **References**  **NA** |
| **Next Steps**   1. Understand more on how Hystrix monitoring can be utilized in the context of Sumo Logic |

## User provided Service Instance

|  |
| --- |
| **Overview**  PCF provides custom provisions to configure service instances for accessing external services. A common usage scenario is accessing on premise database. User provided service instances can be created by developers and any number of applications which want to connect to the same database can then bind itself to this service instance. |
| **Pre-Requisites**   1. PCF Environment Access 2. PCF CLI / Marketplace 3. STS IDE with Maven 4. Java 1.8 5. On premise database |
| **Feature Description**  PCF recommends to utilize User provided Service Instance as the mechanism to deliver the credentials at runtime automatically. The applications should just bind to this service instance and the credentials are delivered at runtime automatically by the platform |
| **Implementation Notes**  To create a User-Provided Service Instance with CF CLI use the CF CUPS command as follows  cf cups <service name> -p "<parameter>, <parameter>".  For example  cf cups ds-dev-hkpes3-db-service –p “jdbcUrl”  On execution of this command, the CF CLI would ask for input for the jdbcUrl parameter. Providing the same would result in creation of the ds-dev-hkpes3-db-service service instance  There are different ways of binding the application with the User Provided Service Instance   1. With PCF Platform UI. (Open the service and bind the application) 2. With PCF CLI   Use the bind service command as follows  cf bind-service <application> <service instance>  The restage command applies the changes to all running instances of the service  cf restage   1. Using the application manifest yml file (recommended option)   services:  - ds-dev-hkpes3-db-service  Spring Boot provides auto-configurations for external binding of data services. Configure the following dependencies as part of the application pom file  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-spring-service-connector</artifactId>  </dependency>  <!-- Cloud Foundry Connector -->  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-cloudfoundry-connector</artifactId>  </dependency>  The cloud connector is a platform-specific interface that identifies the presence of the platform and discovers any services bound to the application deployment. The service connector is an object which represents the runtime connection to a service (for example, a javax.sql.DataSource).  Having included the required dependencies, implement the data source configuration as follows  @Configuration  @ServiceScan  @Component(“DatasourceConfig”)  public class DatasourceConfig {  ..  ..  @Autowired  @Qualifier(“ds-dev-hkpes3-db-service”)  javax.sql.DataSource dataSource;  } |
| **References**   1. PCF User Provided Service Component Overview document 2. POC Source code repository |
| **Next Steps**   1. Identify and understand the other areas / scenario’s where User Provided Instances can be utilized |

## Single Sign On

|  |
| --- |
| **Overview**  The Single Sign On service provided by PCF helps securing the end points of the application and API. After authentication the Single Sign On service utilizes OAuth 2 to secure the resources. That way only users having appropriate access rights would be able to access the applications / services |
| **Pre-Requisites**   1. Access to PCF Sandbox 2. STS IDE with Maven 3. Microservice Application with Secured Endpoint |
| **Feature Description**   1. Explain PCF SSO Service Configuration 2. Roles and Scope configuration 3. Spring Method Level Authorization Annotations |
| **Implementation Notes**  Yet to start since now the SSO is handled at APIGEE level and hence not required at PCF |
| **References**  NA |
| **Next Steps**   1. Generate tokens 2. Passing token in header to run the service |

## Generating Authorization Tokens

|  |
| --- |
| **Overview**  Once the endpoints of the micro services are secured, it’s not possible for the developers to test their services independently without providing a token. To test the services the developers need to generate the Auth Token and add it as Authorization Bearer Token in the request header while testing the service. |
| **Pre-Requisites**   1. Microservice application with endpoints secured 2. SSO Service Client Id, Client Secret Details 3. Curl / Git Bash 4. Rest Client |
| **Feature Description**  Based on the Service Client Id and Client Secret we will generate the bearer token using the curl command and use it to test the secured endpoints |
| **Implementation Notes**  Open Git Bash command prompt and issue the curl command as follows  curl --data "grant\_type=client\_credentials&client\_id=b75d35ff-d33a-46d2-929a-629129193c0c&client\_secret=b0c893d0-339a-424d-bdc7-c285c2cc306e" <https://manulife-dev.login.sys.eas.pcf.manulife.com/oauth/token>  The above command results in creation of a new token and returns the response in JSON format as follows  {"**access\_token**":"*eyJhbGciOiJSUzI1NiIsImtpZCI6ImtleS0xIiwidHlwIjoiSldUIn0..PFhCvaKQySQyMgIuEjVNmLxJTzqDx5HaaXzWablNspLjmuhppGHcrcRS9W4HJeYOuqnFFUjYgSEoPq3pVkDrVN215rTjZHNb6vi18Szkw\_QZnRhb8xAaUNr9FAbPFVi8boHbZwP12TcCk8Zai\_-CMozTOOwsCo7Z9CcQ5wLtqxyD\_DPMgpJubzYN5UN5SC92HZSjCgm78lHt7E3KwjcG7BGibFJykX9AXhK\_jM5lNJQW0HIDJhbKDvzmMI2eNjybsTIsLiJq06JfswKqTjTHUHJdWZJtLU8DAnjGkh4oDwYoIOk3HOjoV536wYiCteZbgM3jW7a9BB\_Bct3U65\_3Xg*","token\_type":"bearer","expires\_in":43199,"**scope**":"*HK-on*  *board.read HK-eligibility-check.read HK-transaction-management.read HK-notification.write HK-document-management.write HK-transaction-management.write HK-document-management.read HK-claim.read HK-claim.write*","jti":"1ac39bb2013f4d7ba146a7df382b7738"}  The returned JSON has the following attributes   1. Access token – For using as part of the request header 2. Token-type – Bearer token 3. Expires – Number of seconds the token is valid 4. Scope – The resources valid for this token   We need to use the Access token as part of the request header as shown below to test the secured endpoint with ARC Rest Client |
| **References**   1. PCF Oauth document |
| **Next Steps**   1. Testing secured API documentation end points |

## Messaging with RabbitMQ

|  |
| --- |
| **Overview**  Messaging technique is used for inter-service asynchronous communication using message The producer and consumer of messages are decoupled by messaging layer known as the message broker. Message broker provides features like persistent storage of messages, message filtering, message transformation etc. RabbitMQ is a message broke. It supports Advanced Message Queuing Protocol (AMQP) a performant messaging protocol standard. |
| **Pre-Requisites**   1. Access to PCF Sandbox (Rabbit MQ as managed services) 2. STS IDE with Maven 3. Java 1.8 |
| **Feature Description**  Spring framework provides Spring AMQP project which simplifies access to AMQP based message brokers with AMQP Templates. Spring Rabbit project provides features which are specific to Rabbit MQ like connection management, Async message consumption etc. |
| **Implementation Notes**  POC Completed, documentation is yet to be updated |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git> |
| **Next Steps**   1. Arrive at the general guidelines / scenario’s on where to use RabbitMQ |

## Working with Pivotal Cloud Cache

|  |
| --- |
| **Overview**  Pivotal Cloud Cache (PCC) is a high-performance, highly available caching layer for Pivotal Cloud Foundry (PCF). PCC offers an in-memory key-value store. The product delivers low-latency responses to a large number of concurrent data access requests |
| **Pre-Requisites**   1. Access to PCF in Sandbox (PCC as managed services) 2. STS IDE with Maven 3. Java 1.8 |
| **Feature Description**  PCF provides PCC as managed services and we would be creating a cluster and region for storing the data in cache. PCC uses Pivotal GemFire and using the Pivotal Gemfire client library we would be able to store and retrieve data from cache |
| **Implementation Notes**  POC Completed, documentation is yet to be updated |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git> |
| **Next Steps**  Arrive at the general guidelines / scenario’s on where to use PCC |

## Working with Redis Cache

|  |
| --- |
| **Overview**  *Redis* is an easy to use, high speed key-value store that can be used in variety of ways like database, cache, and message broker based on the scenario. It supports a wide range of data structures which includes strings, lists, hashes, sets, bitmaps and geospatial indexes |
| **Pre-Requisites**   1. Access to Redis in Sandbox (Redis as managed services) 2. STS IDE with Maven 3. Java 1.8 |
| **Feature Description**  Use Spring Cloud Connectors and Spring Cloud Foundry Connectors to connect to Redis services in PCF. Use Spring Data Redis Template and Jedis client to connect and access Redis to store and retrieve data from the Redis Cache |
| **Implementation Notes**  POC Completed, documentation is yet to be updated |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git> |
| **Next Steps**  Arrive at the general guidelines / scenario’s on where to use Redis Cache |

## Accessing Hive from Microservices

|  |
| --- |
| **Overview**  As per the Manulife DataLake Architecture data from Source Systems / Databases would be periodically pushed to Hadoop and would be loaded on Hive, which would then be utilized by all the other applications |
| **Pre-Requisites** |
| **Feature Description**  Use Spring Data Hadoop and Apache Hive libraries to access Hive database. Uses Kerberos for authentication mechanism. Also provided references to setting up Toad Client for accessing the Hive database |
| **Implementation Notes**  POC Completed, documentation is yet to be updated |
| **References**  <https://git.ap.manulife.com/scm/hksoa/poc.git> |
| **Next Steps**   1. Java Serialization Vs PDX Serialization 2. Performance |

## Utilizing APIGEE and Edge Micro gateway

|  |
| --- |
| **Overview**  Apigee provides comprehensive API management capabilities and scalable delivery of apps on the powerful Pivotal Cloud Foundry platform. Apigee Edge Service Broker for Pivotal Cloud Foundry enables developers to manage APIs for their PCF apps through the Apigee Edge management console |
| **Pre-Requisites**   1. Apigee Edge account 2. Apigee Edge Service Broker for PCF 3. STS IDE with Maven 4. Java 1.8 |
| **Feature Description**  Configuring the API’s Proxy, Configuration of scopes, routes, Client Id and Client Secret. Generation of tokens for testing purpose and validating the requests for authorization and appropriate scopes. Generating documentation with Swagger and accessing it using OAuth |
| **Implementation Notes**  For detailed implementation notes please refer the below References section |
| **References**  Apigee setup steps:  <https://git.ap.manulife.com/projects/RSF/repos/starter-template/browse/apigee/README2.md>  Enable SSO with Apigee in PCF App:  <https://git.ap.manulife.com/projects/HKCOM/repos/sample-pcf-micro-service/browse/security_checklist.md>  Modification to be done for enabling Swagger UI  <https://git.ap.manulife.com/projects/HKCOM/repos/sample-pcf-micro-service/browse/swagger-ui.md>  Example:  <https://apigee-mg-key-service-hk-shared-dev.apps.eas.pcf.manulife.com/manulife-development-dev/publicKey>  <https://apigee-mg-key-service-hk-shared-dev.apps.eas.pcf.manulife.com/manulife-operation-preprod-ext/publicKey>  <https://apigee-mg-key-service-hk-shared-dev.apps.eas.pcf.manulife.com/manulife-operation-prod-ext/publicKey>  <https://git.ap.manulife.com/projects/HKSOA/repos/apigee-mg-key-service/browse> |
| **Next Steps**   1. Creating API Documentation in APIGEE |

# Testing

## Unit Testing Micro Services

|  |
| --- |
| **Overview**  Unit tests cases are written for testing a specific piece of code. The testing for micro services is slightly different from testing a normal web application since there are different type of tests that needs to be performed. Spring boot provides extensive support to for writing Unit test cases. As the name suggests, with unit testing we test only a small functionality / fragment of code. |
| **Pre-Requisites**   1. STS IDE with Maven 2. Java 1.8 3. Database Server 4. REST Client |
| **Feature Description**  Unit tests can be run quickly. Whenever there is any change to the code, rerunning the test cases written earlier would ensure that we have not broken any existing functionality. As part of this exercise we will look at the below features provided by Spring Boot for writing test cases   1. DataJpaTest 2. TestEntityManager 3. MockBean 4. TestConfiguration 5. WebMvcTest 6. MockMvc 7. SpringBootTest 8. TestPropertySource |
| **Implementation Notes**  POC In Progress |
| **References**  NA |
| **Next Steps**  NA |

## Mock Services with Mockito

|  |
| --- |
| **Overview**  Unit tests should always test the functionality in isolation. Side effects from other classes or the system should be eliminated for a unit test. This can be done with test replacements (Mock Objects) for the real dependencies. A mock objectis a dummy implementation for an interface or a class in which outputs for method calls are defined. |
| **Pre-Requisites**   1. STS IDE with Maven 2. Java 1.8 3. REST Client |
| **Feature Description**  Mockito is a popular mock framework which can be used in conjunction with Junit. It provides various annotations like @Mock, @InjectMock with which we would be able to easily mock objects and execute the test case |
| **Implementation Notes**  POC In Progress |
| **References**  NA |
| **Next Steps**  NA |

# Build and Deployment

## Manage project dependencies

|  |
| --- |
| **Overview**  Maven is the most prominent build tool utilized widely. Maven provides a consistent build structure and also provides lots of plugins to simplify the overall build and packaging process. Managing application Jars dependencies, versions and dealing with older versions of Jar etc. can be a challenging task. |
| **Pre-Requisites**   1. STS IDE with Maven 2. Java 8 |
| **Feature Description**  We will look at the provisions provided by Maven to manage version, manage dependencies and also have a look at use of plugins to enforce constraints and rules that needs to be complied with when selecting application Jar files for build and packaging. |
| **Implementation Notes**  **Version Numbers:**  Maven provides the <properties /> section to deal with library, tools, plugins related version numbers  Utilize this section to mention the version numbers and use it in other sections to make it easier to upgrade and test new versions. Also include the version numbers for all the plugins utilized.  <properties>  <java.version>1.8</java.version>  <maven.enforcer.plugin>3.0.0-M1</maven.enforcer.plugin>  <maven.version.range>[3.1.0,3.5.0)</maven.version.range>  <spring-cloud.version>Dalston.SR3</spring-cloud.version>  </properties>  **Dependency Management:**  Maven provides <dependencyManagement /> section to mention all the dependencies of the project. Utilize this section to manage all the dependencies and to control implicit, explicit dependencies. Transitive dependencies would be resolved for libraries included in this section  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  **Enforcer Plugin:**  The enforcer plugin helps to enhance the application stability. For example we would be able to perform   1. Ban circular dependencies 2. Ban duplicate classes 3. Prohibit direct or indirect inclusion of incompatible jars 4. Exclude older versions of libraries 5. Manage dependency convergence (If 2 libraries depends on 2 different versions of the same jar) 6. Etc...   <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-enforcer-plugin</artifactId>  <version>${maven.enforcer.plugin}</version>  <executions>  <execution>  <id>enforce-banned-dependencies</id>  <goals>  <goal>enforce</goal>  </goals>  <configuration>  <rules>  <dependencyConvergence>  <uniqueVersions>false</uniqueVersions>  </dependencyConvergence>  <bannedDependencies>  <searchTransitive>true</searchTransitive>  <excludes>  <exclude>….</exclude>  <exclude>….</exclude>  <exclude>….</exclude>  </excludes>  </bannedDependencies>  <requireMavenVersion>  <version>${maven.version.range}</version>  </requireMavenVersion>  <requireJavaVersion>  <version>${java.version}</version>  </requireJavaVersion>  </rules>  <fail>true</fail>  </configuration>  </execution>  <execution>  <id>enforce-ban-circular-dependencies</id>  <goals>  <goal>enforce</goal>  </goals>  <configuration>  <rules>  <banCircularDependencies />  </rules>  <fail>true</fail>  </configuration>  </execution>  <execution>  <id>enforce-bytecode-version</id>  <goals>  <goal>enforce</goal>  </goals>  <configuration>  <rules>  <enforceBytecodeVersion>  <ignoredScopes>  <scope>test</scope>  </ignoredScopes>  <maxJdkVersion>${java.version}</maxJdkVersion>  </enforceBytecodeVersion>  </rules>  <fail>true</fail>  </configuration>  </execution>  <execution>  <id>enforce-ban-duplicate-classes</id>  <goals>  <goal>enforce</goal>  </goals>  <configuration>  <rules>  <banDuplicateClasses>  <ignoreClasses>  <ignoreClass>…..ignoreClass>  <ignoreClass>…..ignoreClass>  </ignoreClasses>  <findAllDuplicates>true</findAllDuplicates>  </banDuplicateClasses>  </rules>  <fail>true</fail>  </configuration>  </execution>  </executions>  </plugin> |
| **References** |
| **Next Steps**   1. Come up with recommendations for constraints to be enforced which can be utilized by all projects |

## Deploying Micro Services to PCF

|  |
| --- |
| **Overview**  Deploying applications / micro services to PCF can be done in following ways   1. Using PCF CLI (Local Development) 2. Using PCF STS Plugin (Local Development) 3. Using Concourse |
| **Pre-Requisites**   1. PCF CLI 2. STS with Maven and PCF Plugin |
| **Feature Description**  As part of this exercise we would be deploying the application using the first 2 methods and using Concourse would be covered separately.  The Concourse pipeline setup for microservice document describes the implementation of concourse pipeline for microservices. |
| **Implementation Notes**  Summarize the details from different documents to provide an overview as part of the section - Yet to start |
| **References**   1. PCF Deployment Basics document 2. Concourse pipeline setup for microservice |
| **Next Steps**   1. Making the Concourse pipeline more flexible |

# Operational Monitoring

## Analyze Logs in Sumo Logic

|  |
| --- |
| **Overview**  The platform / global team have already configured the routing of log messages from different applications in PCF to Sumo Logic. As part of this exercise we would be looking at the support provided by Sumo Logic to analyze the logs, faults, application metrics, container metrics etc. |
| **Pre-Requisites**   1. Access to Sumo Logic 2. PCF Configured to route logs to Sumo Logic |
| **Feature Description**  The log messages are exported in JSON format to Sumo Logic. We need to understand the JSON format in order to setup queries to parse the JSON objects and derive the specific log messages. We would be able to parse the HTTP logs to parse and analyze the traffic, analyze the performance the applications deployed in different spaces, application usage trends, service response times, container metrics etc. Also we would be able to analyze the errors / faults at space level and application level |
| **Implementation Notes**  The Sumo Logic Analysis Guidelines document describes in detail the implementation of custom queries, analyzing application usage trends, service response times, container related metrics, analyze error / faults etc. |
| **References**   1. Sumo Logic Analysis Guidelines document |
| **Next Steps**   1. Need to analyze various other scenario’s (Spring Sleuth) and see if any custom queries needs to be implemented 2. Need to also validate the capabilities of Sumo Logic to raise alerts based on pre-configured threshold |

## Monitoring Apps with New Relic

|  |
| --- |
| **Overview**  New Relic is the all-in-one web app performance tool that lets analyze performance from the end user experience, through servers, and down to the line of code |
| **Pre-Requisites**   1. PCF New Relic Services |
| **Feature Description**  Configuring the New Relic Instance, binding the application to New Relic services and analyzing applications with New Relic on different sets of parameters |
| **Implementation Notes**  The Features exploration of New Relic document describes the features of New Relic and the implementation of Error logging and custom logging for performance. |
| **References**   1. Features exploration of New Relic |
| **Next Steps**   1. Need to analye best practise on monitoring using New Relic 2. Some features of New Relic are not available yet such as monitoring infrastructure and API availability alert. Need to analye them when they are ready to use. |

# Appendix